

Введение

Microsoft Windows — сложная операционная система. Она включает в себя столько всего и делает так много, что одному человеку просто не под силу полностью разобраться в этой системе. Более того, из-за такой сложности и комплексности Windows трудно решить, с чего начать ее изучение. Лично я всегда начинаю с самого низкого уровня, стремясь получить четкое представление о базовых сервисах операционной системы. Разобравшись в основах, дальше двигаться проще. С этого момента я шаг за шагом, по мере необходимости, изучаю сервисы более высокого уровня, построенные именно на этом базисе.

Например, вопросы, относящиеся к компонентной модели объектов (Component Object Model, COM), в моей книге прямо не затрагиваются. Но COM — это архитектура, где используются процессы, потоки, механизмы управления памятью, DLL, локальная память потоков, Unicode и многое другое. Если Вы знаете, как устроены и работают эти фундаментальные сервисы операционной системы, то для освоения COM достаточно понять, как они применяются в этой архитектуре. Мне очень жаль тех, кто пытается перепрыгнуть через все это и сразу же взяться за изучение архитектуры COM. Впереди у них долгий и тернистый путь; в их знаниях неизбежны пробелы, которые непременно будут мешать им в работе.

И вот тут мы подходим к тому, о чем же моя книга. А она — о строительных кирпичиках Windows, базовых сервисах, в которых (по крайней мере, на мой взгляд) должен досконально разбираться каждый разработчик Windows-приложений. Рассматривая тот или иной сервис, я буду рассказывать, как им пользуется система и как им должно пользоваться Ваше приложение. Во многих главах я буду показывать, как на основе базовых сервисов Windows создавать собственные строительные кирпичики. Реализуя их в виде универсальных функций и классов C++ и комбинируя в них те или иные базовые сервисы Windows, Вы получите нечто большее суммы отдельных частей.

Сегодняшние Windows-платформы

Сейчас Microsoft поставляет операционные системы Windows с тремя ядрами. Каждое ядро оптимизировано под свои виды вычислительных задач. Microsoft пытается переманить разработчиков программного обеспечения на Windows-платформы, утверждая, что интерфейс прикладного программирования (application programming interface, API) у каждой из них одинаков. Это означает лишь то, что, научившись писать Windows-приложения для одного ядра, Вы поймете, как сделать то же самое для остальных.

Поскольку я объясняю, как писать Windows-приложения на основе Windows API, то теоретически все, о чем Вы узнаете из моей книги, применимо ко всем трем ядрам. На самом деле они сильно отличаются друг от друга, и поэтому одни и те же функции соответствующих операционных систем реализованы по-разному. Скажем так: базовые концепции одинаковы, но детали могут различаться.

Начнем с того, что представляют собой эти три ядра Windows.

Ядро Windows 2000

Windows 2000 — это операционная система Microsoft класса «high-end». Список ее возможностей и особенностей займет не одну страницу. Вот лишь некоторые из них (в совершенно произвольном порядке).

- Windows 2000 рассчитана на рабочие станции и серверы, а также на применение в центрах обработки данных.
- Отказоустойчива — плохо написанные программы не могут привести к краху системы.
- Защищена — несанкционированный доступ к ресурсам (например, файлам или принтерам), управляемым этой системой, невозможен.
- Богатый набор средств и утилит для администрирования системы в масштабах организации.
- Ядро Windows 2000 написано в основном на С и С++, поэтому система легко портируется (переносится) на процессоры с другими архитектурами.
- Полностью поддерживает Unicode, что упрощает локализацию и работу с использованием различных языков.
- Имеет высокоэффективную подсистему управления памятью с чрезвычайно широкими возможностями.
- Поддерживает структурную обработку исключений (structured exception handling, SEH), облегчая восстановление после ошибок.
- Позволяет расширять функциональность за счет динамически подключаемых библиотек (DLL).
- Поддерживает многопоточность и мультипроцессорную обработку, обеспечивая высокую масштабируемость системы.
- Файловая система Windows 2000 дает возможность отслеживать, как пользователи манипулируют с данными на своих машинах.

Ядро Windows 98

Windows 98 — операционная система потребительского класса. Она обладает многими возможностями Windows 2000, но некоторые ключевые из них не поддерживает. Так, Windows 98 не отнесешь к числу отказоустойчивых (приложение вполне способно привести к краху системы), она менее защищена, работает только с одним процессором (что ограничивает ее масштабируемость) и поддерживает Unicode лишь частично.

Microsoft намерена ликвидировать ядро Windows 98, поскольку его доработка до уровня ядра Windows 2000 потребовала бы слишком много усилий. Да и кому нужно еще одно ядро Windows 2000? Так что Windows 2000 — это вроде бы надолго, а Windows 98 проживет года два-три, если не меньше.

Но почему вообще существует ядро Windows 98? Ответ очень прост: Windows 98 более дружественна к пользователю, чем Windows 2000. Потребители не любят регистрироваться на своих компьютерах, не хотят заниматься администрированием и т. д. Плюс ко всему в компьютерные игры они играют чаще, чем сотрудники корпораций в рабочее время (впрочем, это спорно). Многие старые игровые программы обращаются к оборудованию напрямую, что может приводить к зависанию компьютера. Windows 2000 — операционная система с отказоустойчивым ядром — такого не по-

зволяет никому. Любая программа, которая пытается напрямую обратиться к оборудованию, немедленно завершается, не успев навредить ни себе, ни другим.

По этим причинам Windows 98 все еще с нами, и ее доля на рынке операционных систем весьма велика. Microsoft активно работает над тем, чтобы Windows 2000 стала дружественнее к пользователю, — очень скоро появится потребительская версия ее ядра. Поскольку ядра Windows 98 и Windows 2000 имеют сходные наборы функциональных возможностей и поскольку они наиболее популярны, я решил сосредоточиться в этой книге именно на них.

Готовя книгу, я старался обращать внимание на отличия реализаций Win32 API в Windows 98 и Windows 2000. Материалы такого рода я обводил рамками и, как показано ниже, помечал соответствующими значками — чтобы привлечь внимание читателей к каким-то деталям, характерным для той или иной платформы.

WINDOWS 98 Здесь рассказывается об особенностях реализации на платформе Windows 98.

WINDOWS 2000 А тут — об особенностях реализации на платформе Windows 2000.

Windows 95 я особо не рассматриваю, но все, что относится к Windows 98, применимо и к ней, так как ядра у них совершенно одинаковые.

Ядро Windows CE

Windows CE — самое новое ядро Windows от Microsoft. Оно рассчитано главным образом на карманные и автомобильные компьютеры, «интеллектуальные» терминалы, тостеры, микроволновые печи и торговые автоматы. Большинство таких устройств должно потреблять минимум электроэнергии, у них очень мало памяти, а дисков чаще всего просто нет. Из-за столь жестких ограничений Microsoft пришлось создать совершенно новое ядро операционной системы, намного менее требовательное к памяти, чем ядро Windows 98 или Windows 2000.

Как ни странно, Windows CE довольно мощная операционная система. Устройства, которыми она управляет, предназначены только для индивидуального использования, поэтому ее ядро не поддерживает администрирование, масштабирование и т. д. Тем не менее практически все концепции Win32 применимы и к данной платформе. Различия обычно проявляются там, где Windows CE накладывает ограничения на те или иные Win32-функции.

Завтрашние Windows-платформы (64-разрядная Windows 2000)

Будущее уже совсем близко. Когда я пишу эти строки, Microsoft напряженно трудится над переносом ядра Windows 2000 на 64-разрядную платформу. Предполагается, что эта истинно 64-разрядная операционная система получит название *64-bit Windows 2000* (64-разрядная Windows 2000). На первых порах она будет работать на процессорах Alpha (архитектура AXP64) от Compaq, а чуть позже и на новых процессорах Itanium (архитектура IA-64) от Intel.

Процессоры Alpha всегда были 64-разрядными. Так что, если у Вас есть машина с одним из этих процессоров, Вы просто установите 64-разрядную Windows 2000 и получите полноценную 64-разрядную программно-аппаратную платформу. Процес-

соры Intel серии Pentium (и более ранние) имеют 32-разрядную архитектуру (IA-32). Машины с такими процессорами не смогут работать с 64-разрядной Windows 2000. Intel сравнительно недавно закончил разработку новой 64-разрядной архитектуры процессоров и сейчас готовит к выпуску процессор Itanium (его кодовое название было Merced). Поставка машин на базе Itanium ожидается уже в 2000 году.

Меня очень интересует 64-разрядная Windows 2000, и я давно готовлюсь к ее появлению. Сейчас на Web-узле Microsoft можно найти много статей о 64-разрядной Windows 2000 и о том, какие изменения она принесет разработчикам программного обеспечения. С радостью сообщаю Вам следующее.

- Ядро 64-разрядной Windows 2000 получено в результате портирования ядра 32-разрядной версии. А значит, все, что Вы узнали о 32-разрядной Windows 2000, применимо и к 64-разрядной. В сущности, Microsoft так модифицировала исходный код 32-разрядной Windows, что из него можно получить как 32-, так и 64-разрядную систему. Таким образом, у них теперь одна база исходного кода, и любые новшества или исправления будут вноситься в обе системы одновременно.
- Поскольку эти ядра построены на одном коде и одинаковых концепциях, Windows API идентичен на обеих платформах. Следовательно, Ваши приложения потребуют лишь минимальной модификации.
- Если перенос 32-разрядных приложений так легок, то вскоре появится масса инструментальных средств (вроде Microsoft Developer Studio), поддерживающих разработку 64-разрядного программного обеспечения.
- Конечно, 64-разрядная Windows сможет выполнять и 32-разрядные приложения. Но, судя по обещаниям, истинно 64-разрядные приложения будут работать в ней гораздо быстрее.
- Вам не придется учиться заново. Вы обрадуетесь, узнав, что большинство типов данных осталось 32-разрядным. Это относится к целым типам, DWORD, LONG, BOOL и т. д. По сути, беспокоиться следует лишь об указателях и некоторых описателях, так как теперь они являются 64-разрядными.

Сведений о том, как подготовить исходный код к выполнению на 64-разрядной платформе, вполне хватает и на Web-узле Microsoft, так что я в эти детали вдаваться не буду. Но, что бы я ни писал в каждой главе, я все время помнил о 64-разрядной Windows и, где это было нужно, включал специфическую для нее информацию. Кроме того, все приведенные в этой книге программы-примеры я компилировал с использованием 64-разрядного компилятора, что позволило мне протестировать их на очень ранней версии 64-разрядной Windows 2000 для процессоров Alpha. Если Вы будете следовать тем же правилам, что и я, Вам не составит труда создать единую базу исходного кода своих приложений для 32- и 64-разрядной Windows.

Что нового в четвертом издании

Четвертое издание является практически новой книгой. Я решил разбить материал на большее количество глав для более четкой структуризации и изменил порядок его изложения. Надеюсь, так будет легче изучать его и усваивать. Например, глава по Unicode теперь находится в начале книги, поскольку с ним так или иначе связаны многие другие темы.

Более того, все темы рассматриваются гораздо глубже, чем в предыдущих изданиях. В частности, я подробнее, чем раньше, объясняю внутреннее устройство Windows,

чтобы Вы точно знали, что происходит за кулисами этой системы. Намного детальнее я рассказываю и о том, как взаимодействует с системой библиотека C/C++ (C/C++ run-time library) — особенно при создании и уничтожении процессов и потоков. Динамически подключаемым библиотекам я также уделяю больше внимания.

Помимо этих изменений, в книге появилась целая тонна нового содержания. Упомяну лишь самое главное.

- **Новшества Windows 2000.** Книгу было бы нельзя считать действительно переработанной, не будь в ней отражены новшества Windows 2000: объект ядра «задание» (job kernel object), функции для создания пула потоков (thread pooling functions), изменения в механизме планирования потоков (thread scheduling), расширения Address Windowing, вспомогательные информационные функции (toolhelp functions), разреженные файлы (sparse files) и многое другое.
- **Поддержка 64-разрядной Windows.** В книге приводится информация, специфическая для 64-разрядной Windows; все программы-примеры построены с учетом специфики этой версии Windows и протестированы в ней.
- **Практичность программ-примеров.** Я заменил многие старые примеры новыми, более полезными в повседневной работе; они иллюстрируют решение не абстрактных, а реальных проблем программирования.
- **Применение C++.** По требованию читателей примеры теперь написаны на C++. В итоге они стали компактнее и легче для понимания.
- **Повторно используемый код.** Я старался писать по возможности универсальный и повторно используемый код. Это позволит Вам брать из него отдельные функции или целые C++-классы без изменений (незначительная модификация может понадобиться лишь в отдельных случаях). Код на C++ гораздо проще для повторного использования.
- **Утилита VMMap.** Эта программа-пример из предыдущих изданий серьезно усовершенствована. Ее новая версия дает возможность исследовать адресное пространство любого процесса, выяснить полные имена (вместе с путями) любых файлов данных, спроектированных в адресное пространство процесса, копировать информацию из памяти в буфер обмена и (если Вы пожелаете) просматривать только регионы или блоки памяти внутри регионов.
- **Утилита ProcessInfo.** Это новая утилита. Она показывает, какие процессы выполняются в системе и какие DLL используются тем или иным модулем. Как только Вы выбираете конкретный процесс, ProcessInfo может запустить утилиту VMMap для просмотра всего адресного пространства этого процесса. ProcessInfo позволяет также узнать, какие модули загружены в системе и какие исполняемые файлы используют определенный модуль. Кроме того, Вы сможете увидеть, у каких модулей были изменены базовые адреса из-за неподходящих значений.
- **Утилита LISWatch.** Тоже новая утилита. Она отслеживает общесистемные и специфические для конкретного потока изменения в локальном состоянии ввода. Эта утилита поможет Вам разобраться в проблемах, связанных с перемещением фокуса ввода в пользовательском интерфейсе.
- **Информация по оптимизации кода.** В этом издании я даю гораздо больше информации о том, как повысить быстродействие кода и сделать его компактнее. В частности, я подробно рассказываю о выравнивании данных (data alignment), привязке к процессорам (processor affinity), кэш-линиях процессо-

ра (CPU cache lines), модификации базовых адресов (rebasing), связывании модулей (module binding), отложенной загрузке DLL (delay-loading DLLs) и др.

- **Существенно переработанный материал по синхронизации потоков.** Я полностью переписал и перестроил весь материал по синхронизации потоков. Теперь я сначала рассказываю о самых эффективных способах синхронизации, а наименее эффективные обсуждаю в конце. Попутно я добавил новую главу, посвященную набору инструментальных средств, которые помогают решать наиболее распространенные проблемы синхронизации потоков.
- **Детальная информация о форматах исполняемых файлов.** Форматы файлов EXE- и DLL-модулей рассматриваются намного подробнее. Я рассказываю о различных разделах этих модулей и некоторых специфических параметрах компоновщика, которые позволяют делать с модулями весьма интересные вещи.
- **Более подробные сведения о DLL.** Главы по DLL тоже полностью переписаны и перестроены. Первая из них отвечает на два основных вопроса: «Что такое DLL?» и «Как ее создать?» Остальные главы по DLL посвящены весьма продвинутым и отчасти новым темам — явному связыванию (explicit linking), отложенной загрузке, переадресации вызова функций (function forwarding), перенаправлению DLL (DLL redirection) (новая возможность, появившаяся в Windows 2000), модификации базового адреса модуля (module rebasing) и связыванию.
- **Перехват API-вызовов.** Да, это правда. За последние годы я получил столько почты с вопросами по перехвату API-вызовов (API hooking), что в конце концов решил включить эту тему в свою книгу. Я представлю Вам несколько C++-классов, которые сделают перехват API-вызовов в одном или всех модулях процесса тривиальной задачей. Вы сможете перехватывать даже вызовы *LoadLibrary* и *GetProcAddress* от библиотеки C/C++!
- **Более подробные сведения о структурной обработке исключений.** Эту часть я тоже переписал и во многом перестроил. Вы найдете здесь больше информации о необрабатываемых исключениях и увидите C++-класс — оболочку кода, управляющего виртуальной памятью за счет структурной обработки исключений (structured exception handling). Я также добавил сведения о соответствующих приемах отладки и о том, как обработка исключений в C++ соотносится со структурной обработкой исключений.
- **Обработка ошибок.** Это новая глава. В ней показывается, как правильно перехватывать ошибки при вызове API-функций. Здесь же представлены некоторые приемы отладки и ряд других сведений.
- **Windows Installer.** Чуть не забыл: программы-примеры (все они содержатся на прилагаемом компакт-диске) используют преимущества нового Windows Installer, встроенного в Windows 2000. Это позволит полностью контролировать состав устанавливаемого программного обеспечения и легко удалять больше не нужные его части через апллет Add/Remove Programs в Control Panel. Если Вы используете Windows 95/98 или Windows NT 4.0, программа Setup с моего компакт-диска сначала установит Windows Installer. Но, разумеется, Вы можете и сами скопировать с компакт-диска любые интересующие Вас файлы с исходным или исполняемым кодом.

В этой книге нет ошибок

Этот заголовок отражает лишь то, что я хотел бы сказать. Но все мы знаем: это полное вранье. Мои редакторы и я очень старались без ошибок донести до Вас новую, точную и глубокую информацию в простом для понимания виде. Увы, даже собрав самую фантастическую команду, никто не застрахован от проколов. Найдете какую-нибудь ошибку в этой книге, сообщите мне на <http://www.JeffreyRichter.com> — буду крайне признателен.

Содержимое компакт-диска и требования к системе

Компакт-диск, прилагаемый к книге, содержит исходный код и исполняемые файлы всех программ-примеров. Эти программы написаны и скомпилированы с использованием Microsoft Visual C++ 6.0. Большая их часть будет работать в Windows 95, Windows 98, Windows NT 4.0 и Windows 2000, но некоторые программы требуют такую функциональность, которая поддерживается только Windows NT 4.0 и Windows 2000. Если Вы захотите самостоятельно скомпилировать какие-то примеры, Вам понадобится Microsoft Visual C++ 6.0.

В корневом каталоге компакт-диска находится общий заголовочный файл (Cmnhdr.h) и около трех десятков каталогов, в которых хранятся соответствующие программы-примеры. В каталогах x86 и Alpha32 содержатся отладочные версии тех же программ — их можно запускать прямо с компакт-диска.

Вставив компакт-диск в привод CD-ROM, Вы увидите окно Welcome. Если оно не появится, перейдите в каталог Setup на компакт-диске и запустите файл PressCDx86.exe или PressCDAlpha32.exe (в зависимости от того, какой процессор в Вашем компьютере).

Техническая поддержка

Microsoft Press публикует исправления на <http://mspress.microsoft.com/support>.

Если у Вас есть какие-нибудь комментарии, вопросы или идеи, касающиеся моей книги, пожалуйста, направляйте их в Microsoft Press по обычной или электронной почте:

Microsoft Press

Attn: *Programming Applications for Microsoft Windows*, 4th ed., editor

One Microsoft Way

Redmond, WA 98052-6399

mspinput@microsoft.com

Спасибо всем за помощь

Я не смог бы написать эту книгу без помощи и содействия многих людей. Вот кого хотелось бы поблагодарить особо.

Членов редакторской группы Microsoft Press: Джека Бьюодри (Jack Beaudry), Донни Камерон (Donnie Cameron), Айни Чэнга (Ina Chang), Карла Дилтца (Carl Diltz), Стефена Гьюти (Stephen Guty), Роберта Лайена (Robert Lyon), Ребекку Мак-Кэй (Rebecca McKay), Роба Нэнса (Rob Nance), Джослин Пол (Jocelyn Paul), Шона Пека (Shawn Peck), Джона Пиэрса (John Pierce), Барб Раньян (Barb Runyan), Бена Райена (Ben Ryan), Эрика Стру (Eric Stroo) и Уильяма Тила (William Teel).

Членов группы разработчиков Windows 2000: Асмуса Фрейтага (Asmus Freytag), Дэйва Харта (Dave Hart), Ли Харт (Lee Hart), Джеффа Хейвнса (Jeff Havens), Локеша Сриниваса Копполу (Lokesh Srinivas Koppolu), Он Ли (On Lee), Скотта Людвига (Scott Ludwig), Лью Перазоли (Lou Perazzoli), Марка Луковски (Mark Lucovsky), Лэнди Уэнга (Landy Wang) и Стива Вуда (Steve Wood).

Членов группы разработчиков Windows 95 и Windows 98: Брайана Смита (Brian Smith), Джона Томасона (Jon Thomason) и Майкла Тутуньи (Michael Toutonghi).

Членов группы разработчиков Visual C++: Джонатана Марка (Jonathan Mark), Чака Митчела (Chuck Mitchell), Стива Солсбери (Steve Salisbury) и Дэна Спэлдинга (Dan Spalding).

Членов группы разработчиков IA-64 из корпорации Intel: Джэфа Мюррея (Geoff Murray), Хуана Родригеса (Juan Rodriguez), Джейсона Уэксмена (Jason Waxman), Коичи Ямады (Koichi Yamada), Кита Йедлина (Keith Yedlin) и Уилфреда Ю (Wilfred Yu).

Членов группы разработчиков AXP64 из корпорации Compaq: Тома ван Баака (Tom Van Baak), Билла Бакстера (Bill Baxter), Джима Лэйна (Lim Lane), Рича Петерсона (Rich Peterson), Энни По (Annie Poh) и Джозефа Сиримарко (Joseph Sirimarco).

Членов группы разработчиков InstallShield's Installer: Боба Бэйкера (Bob Baker), Кевина Фута (Kevin Foote) и Тайлера Робинсона (Tyler Robinson).

Участников всевозможных вечеринок: Джеффа Куперстайна (Jeff Cooperstein) и Стефани (Stephanie), Кита Плиса (Keith Pleas) и Сьюзан (Susan), Сьюзан Рейми (Susan Ramee) и Санджив Сурати (Sanjeev Surati), Скотта Людвига (Scott Ludwig) и Вал Хорвач (Val Horvath) с их сыном Николасом (Nicholas), Даррен (Darren) и Шаула Массену (Shaula Massena), Дэвида Соломона (David Solomon), Джеффа Просиза (Jeff Prosise), Джима Харкинса (Jim Harkins), Тони Спика (Tony Spika).

Членов Братства Рихтеров: Рона (Ron), Марию (Maria), Джоуи (Joeу) и Брэнди (Brandi).

Основателей семьи Рихтеров: Арлин (Arlene) и Сильвэна (Sylvan).

Члена фракции косматых: Макса (Max).

Члена группы поддержки: Кристин Трейс (Kristin Trace).

ЧАСТЬ I

МАТЕРИАЛЫ ДЛЯ ОБЯЗАТЕЛЬНОГО ЧТЕНИЯ



Обработка ошибок

Прежде чем изучать функции, предлагаемые Microsoft Windows, посмотрим, как в них устроена обработка ошибок.

Когда Вы вызываете функцию Windows, она проверяет переданные ей параметры, а затем пытается выполнить свою работу. Если Вы передали недопустимый параметр или если данную операцию нельзя выполнить по какой-то другой причине, она возвращает значение, свидетельствующее об ошибке. В таблице 1-1 показаны типы данных для возвращаемых значений большинства функций Windows.

Тип данных	Значение, свидетельствующее об ошибке
VOID	Функция всегда (или почти всегда) выполняется успешно. Таких функций в Windows очень мало.
BOOL	Если вызов функции заканчивается неудачно, возвращается 0; в остальных случаях возвращаемое значение отлично от 0. (Не пытайтесь проверять его на соответствие TRUE или FALSE.)
HANDLE	Если вызов функции заканчивается неудачно, то обычно возвращается NULL; в остальных случаях HANDLE идентифицирует объект, которым Вы можете манипулировать. Будьте осторожны: некоторые функции возвращают HANDLE со значением INVALID_HANDLE_VALUE, равным –1. В документации Platform SDK для каждой функции четко указывается, что именно она возвращает при ошибке –NULL или INVALID_HANDLE_VALUE.
PVOID	Если вызов функции заканчивается неудачно, возвращается NULL; в остальных случаях PVOID сообщает адрес блока данных в памяти.
LONG или DWORD	Это значение — «крепкий орешек». Функции, которые сообщают значения каких-либо счетчиков, обычно возвращают LONG или DWORD. Если по какой-то причине функция не сумела сосчитать то, что Вы хотели, она обычно возвращает 0 или –1 (все зависит от конкретной функции). Если Вы используете одну из таких функций, проверьте по документации Platform SDK, каким именно значением она уведомляет об ошибке.

Таблица 1-1. Стандартные типы значений, возвращаемых функциями Windows

При возникновении ошибки Вы должны разобраться, почему вызов данной функции оказался неудачен. За каждой ошибкой закреплен свой код — 32-битное число.

Функция Windows, обнаружив ошибку, через механизм локальной памяти потока сопоставляет соответствующий код ошибки с вызывающим потоком. (Локальная память потока рассматривается в главе 21.) Это позволяет потокам работать независимо друг от друга, не вмешиваясь в чужие ошибки. Когда функция вернет Вам управление, ее возвращаемое значение будет указывать на то, что произошла какая-то ошибка. Какая именно — Вы узнаете, вызвав функцию *GetLastError*:

```
DWORD GetLastError();
```

Она просто возвращает 32-битный код ошибки для данного потока.

Теперь, когда у Вас есть код ошибки, Вам нужно обменять его на что-нибудь более приятное. Список кодов ошибок, определенных Microsoft, содержится в заголовочном файле WinError.h. Я приведу здесь его небольшую часть, чтобы Вы представляли, на что он похож.

```
// MessageId: ERROR_SUCCESS
//
// MessageText:
//
// The operation completed successfully.
//
#define ERROR_SUCCESS           0L

#define NO_ERROR                // dderror
#define SEC_E_OK                 ((HRESULT)0x00000000L)

//
// MessageId: ERROR_INVALID_FUNCTION
//
// MessageText:
//
// Incorrect function.
//
#define ERROR_INVALID_FUNCTION   1L    // dderror

//
// MessageId: ERROR_FILE_NOT_FOUND
//
// MessageText:
//
// The system cannot find the file specified.
//
#define ERROR_FILE_NOT_FOUND     2L

//
// MessageId: ERROR_PATH_NOT_FOUND
//
// MessageText:
//
// The system cannot find the path specified.
//
#define ERROR_PATH_NOT_FOUND     3L

//
// MessageId: ERROR_TOO_MANY_OPEN_FILES
//
// MessageText:
//
// The system cannot open the file.

//
#define ERROR_TOO_MANY_OPEN_FILES 4L
```

см. след. стр.

```
//  
// MessageId: ERROR_ACCESS_DENIED  
//  
// MessageText:  
//  
// Access is denied.  
//  
#define ERROR_ACCESS_DENIED 5L
```

Как видите, с каждой ошибкой связаны идентификатор сообщения (его можно использовать в исходном коде для сравнения со значением, возвращаемым *GetLastError*), текст сообщения (описание ошибки на нормальном языке) и номер (вместо него лучше использовать индентификатор). Учтите, что я показал лишь крошечную часть файла WinError.h; на самом деле в нем более 21 000 строк!

Функцию *GetLastError* нужно вызывать сразу же после неудачного вызова функции Windows, иначе код ошибки может быть потерян.



GetLastError возвращает последнюю ошибку, возникшую в потоке. Если этот поток вызывает другую функцию Windows и все проходит успешно, код последней ошибки не перезаписывается и не используется как индикатор благополучного вызова функции. Лишь несколько функций Windows нарушают это правило и все же изменяют код последней ошибки. Однако в документации Platform SDK утверждается обратное: якобы после успешного выполнения API-функции обычно изменяют код последней ошибки.

WINDOWS 98

Многие функции Windows 98 на самом деле реализованы в 16-разрядном коде, унаследованном от операционной системы Windows 3.1. В нем не было механизма, сообщающего об ошибках через некую функцию наподобие *GetLastError*, и Microsoft не стала «исправлять» 16-разрядный код в Windows 98 для поддержки обработки ошибок. На практике это означает, что многие Win32-функции в Windows 98 не устанавливают код последней ошибки после неудачного завершения, а просто возвращают значение, которое свидетельствует об ошибке. Поэтому Вам не удастся определить причину ошибки.

Некоторые функции Windows всегда завершаются успешно, но по разным причинам. Например, попытка создать объект ядра «событие» с определенным именем может быть успешна либо потому, что Вы действительно создали его, либо потому, что такой объект уже есть. Но иногда нужно знать причину успеха. Для возврата этой информации Microsoft предпочла использовать механизм установки кода последней ошибки. Так что и при успешном выполнении некоторых функций Вы можете вызывать *GetLastError* и получать дополнительную информацию. К числу таких функций относится, например, *CreateEvent*. О других функциях см. Platform SDK.

На мой взгляд, особенно полезно отслеживать код последней ошибки в процессе отладки. Кстати, отладчик в Microsoft Visual Studio 6.0 позволяет настраивать окно Watch так, чтобы оно всегда показывало код и описание последней ошибки в текущем потоке. Для этого надо выбрать какую-нибудь строку в окне Watch и ввести «@err,hr». Теперь посмотрите на рис. 1-1. Видите, я вызвал функцию *CreateFile*. Она вернула значение INVALID_HANDLE_VALUE (-1) типа HANDLE, свидетельствующее о том, что ей не удалось открыть заданный файл. Но окно Watch показывает нам код последней ошибки (который вернула бы функция *GetLastError*, если бы я ее вызвал),

равный 0x00000002, и описание «The system cannot find the file specified» («Система не может найти указанный файл»). Именно эта строка и определена в заголовочном файле WinError.h для ошибки с кодом 2.

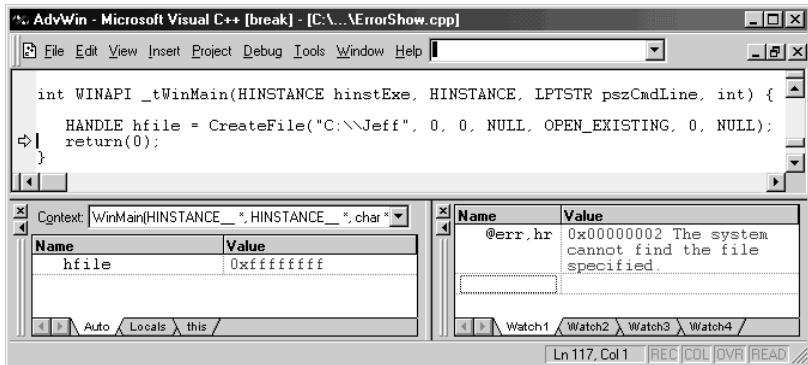
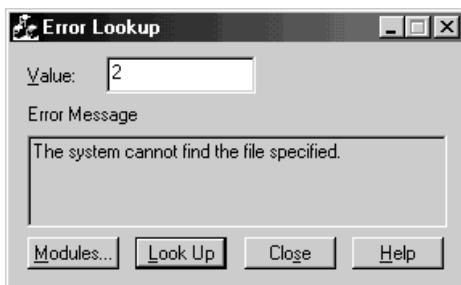


Рис. 1-1. Используя «@err,hr» в окне Watch среды Visual Studio 6.0, Вы можете просматривать код последней ошибки в текущем потоке

С Visual Studio поставляется небольшая утилита Error Lookup, которая позволяет получать описание ошибки по ее коду.



Если приложение обнаруживает какую-нибудь ошибку, то, как правило, сообщает о ней пользователю, выводя на экран ее описание. В Windows для этого есть специальная функция, которая «конвертирует» код ошибки в ее описание, — *FormatMessage*.

```
DWORD FormatMessage(
    DWORD dwFlags,
    LPCVOID pSource,
    DWORD dwMessageId,
    DWORD dwLanguageId,
    PTSTR pszBuffer,
    DWORD nSize,
    va_list *Arguments);
```

FormatMessage — весьма богатая по своим возможностям функция, и именно ее желательно применять при формировании всех строк, показываемых пользователю. Дело в том, что она позволяет легко работать со множеством языков. *FormatMessage* определяет, какой язык выбран в системе в качестве основного (этот параметр задается через апллет Regional Settings в Control Panel), и возвращает текст на соответствующем языке. Разумеется, сначала Вы должны перевести строки на нужные языки и встроить этот ресурс в свой EXE- или DLL-модуль, зато потом функция будет автоматически выбирать требуемый язык. Программа-пример ErrorShow, приведенная в кон-

це главы, демонстрирует, как вызывать эту функцию для получения текстового описания ошибки по ее коду, определенному Microsoft.

Время от времени меня кто-нибудь да спрашивает, составит ли Microsoft полный список кодов всех ошибок, возможных в каждой функции Windows. Ответ: увы, нет. Скажу больше, такого списка никогда не будет — слишком уж сложно его составлять и поддерживать для все новых и новых версий системы.

Проблема с подобным списком еще и в том, что Вы вызываете одну API-функцию, а она может обратиться к другой, та — к третьей и т. д. Любая из этих функций может завершиться неудачно (и по самым разным причинам). Иногда функция более высокого уровня сама справляется с ошибкой в одной из вызванных ею функций и в конечном счете выполняет то, что Вы от нее хотели. В общем, для создания такого списка Microsoft пришлось бы проследить цепочки вызовов в каждой функции, что очень трудно. А с появлением новой версии системы эти цепочки нужно было бы пересматривать заново.

Вы тоже можете это сделать

О'кэй, я показал, как функции Windows сообщают об ошибках. Microsoft позволяет Вам использовать этот механизм и в собственных функциях. Допустим, Вы пишете функцию, к которой будут обращаться другие программы. Вызов этой функции может по какой-либо причине завершиться неудачно, и Вам тоже нужно сообщать об ошибках. С этой целью Вы просто устанавливаете код последней ошибки в потоке и возвращаete значение FALSE, INVALID_HANDLE_VALUE, NULL или что-то другое, более подходящее в Вашем случае. Чтобы установить код последней ошибки в потоке, Вы вызываете *SetLastError*:

```
VOID SetLastError(DWORD dwErrCode);
```

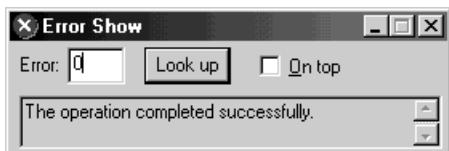
и передаете ей нужное 32-битное число. Я стараюсь использовать коды, уже определенные в WinError.h, — при условии, что они подходят под те ошибки, о которых могут сообщать мои функции. Если Вы считаете, что ни один из кодов в WinError.h не годится для ошибки, возможной в Вашей функции, определите свой код. Он представляет собой 32-битное значение, которое разбито на поля, показанные в следующей таблице.

Биты	31–30	29	28	27–16	15–0
Содержимое:	Код степени «тяжести» (severity)	Кем определен — Microsoft или пользователем	Зарезервирован	Код подсистемы (facility code)	Код исключения
Значение:	0 = успех 1 = информация 2 = предупреждение 3 = ошибка	0 = Microsoft 1 = пользователь	Должен быть 0	Определяется Microsoft	Определяется Microsoft или пользователем

Подробнее об этих полях я рассказываю в главе 24. На данный момент единственное важное для Вас поле — бит 29. Microsoft обещает, что все коды ошибок, генерируемые ее функциями, будут содержать 0 в этом бите. Если Вы определяете собственный код ошибки, запишите сюда 1. Тогда у Вас будет гарантия, что Ваш код ошибки не войдет в конфликт с кодом, определенным Microsoft, — ни сейчас, ни в будущем.

Программа-пример ErrorShow

Эта программа, «01 ErrorShow.exe» (см. листинг на рис. 1-2), демонстрирует, как получить текстовое описание ошибки по ее коду. Файлы исходного кода и ресурсов программы находятся в каталоге 01-ErrorShow на компакт-диске, прилагаемом к книге. Программа ErrorShow в основном предназначена для того, чтобы Вы увидели, как работают окно Watch отладчика и утилита Error Lookup. После запуска ErrorShow открывается следующее окно.



В поле Error можно ввести любой код ошибки. Когда Вы щелкнете кнопку Look Up, внизу, в прокручиваемом окне появится текст с описанием данной ошибки. Единственная интересная особенность программы заключается в том, как она обращается к функции *FormatMessage*. Я использую эту функцию так:

```
// получаем код ошибки
DWORD dwError = GetDlgItemInt(hwnd, IDC_ERRORCODE, NULL, FALSE);

HLOCAL hlocal = NULL; // буфер для строки с описанием ошибки

// получаем текстовое описание ошибки
BOOL fOk = FormatMessage(
    FORMAT_MESSAGE_FROM_SYSTEM | FORMAT_MESSAGE_ALLOCATE_BUFFER,
    NULL, dwError, MAKELANGID(LANG_ENGLISH, SUBLANG_ENGLISH_US),
    (LPTSTR) &hlocal, 0, NULL);

:

if (hlocal != NULL) {
    SetDlgItemText(hwnd, IDC_ERRORTEXT, (PCTSTR) LocalLock(hlocal));
    LocalFree(hlocal);
} else {
    SetDlgItemText(hwnd, IDC_ERRORTEXT, TEXT("Error number not found."));
}
```

Первая строка считывает код ошибки из текстового поля. Далее я создаю экземпляр описателя (handle) блока памяти и инициализирую его значением NULL. Функция *FormatMessage* сама выделяет нужный блок памяти и возвращает нам его описатель.

Вызывая *FormatMessage*, я передаю флаг *FORMAT_MESSAGE_FROM_SYSTEM*. Он сообщает функции, что мне нужна строка, соответствующая коду ошибки, определенному в системе. Кроме того, я передаю флаг *FORMAT_MESSAGE_ALLOCATE_BUFFER*, чтобы функция выделила соответствующий блок памяти для хранения текста. Описатель этого блока будет возвращен в переменной *hlocal*. Третий параметр указывает код интересующей нас ошибки, а четвертый — язык, на котором мы хотим увидеть ее описание.

Если выполнение *FormatMessage* заканчивается успешно, описание ошибки помещается в блок памяти, и я копирую его в прокручиваемое окно, расположенное в нижней части окна программы. А если вызов *FormatMessage* оказывается неудачным,

я пытаюсь найти код сообщения в модуле NetMsg.dll, чтобы выяснить, не связана ли ошибка с сетью. Используя описатель NetMsg.dll, я вновь вызываю *FormatMessage*. Дело в том, что у каждого DLL или EXE-модуля может быть собственный набор кодов ошибок, который включается в модуль с помощью Message Compiler (MC.exe). Как раз это и позволяет делать утилита Error Lookup через свое диалоговое окно Modules.



ErrorShow.cpp

```
/*
Модуль: ErrorShow.cpp
Автор: Copyright (c) 2000, Джейфри Рихтер (Jeffrey Richter)
*****
```

```
#include "..\CmnHdr.h"      /* см. приложение A */
#include <Windowsx.h>
#include <tchar.h>
#include "Resource.h"

///////////

#define ESM_POKECODEANDLOOKUP    (WM_USER + 100)
const TCHAR g_szAppName[] = TEXT("Error Show");

///////////

BOOL Dlg_OnInitDialog(HWND hwnd, HWND hwndFocus, LPARAM lParam) {

    chSETDLGICONS(hwnd, IDI_ERRORSHOW);

    // не принимаем коды ошибок, состоящие более чем из 5 цифр
    Edit_LimitText(GetDlgItem(hwnd, IDC_ERRORCODE), 5);

    // проверяем, не передан ли код ошибки через командную строку
    SendMessage(hwnd, ESM_POKECODEANDLOOKUP, lParam, 0);
    return(TRUE);
}

///////////

void Dlg_OnCommand(HWND hwnd, int id, HWND hwndCtl, UINT codeNotify) {

    switch (id) {

        case IDCANCEL:
            EndDialog(hwnd, id);
            break;

        case IDC_ALWAYSONTOP:
            SetWindowPos(hwnd, IsDlgButtonChecked(hwnd, IDC_ALWAYSONTOP)
                ? HWND_TOPMOST : HWND_NOTOPMOST, 0, 0, 0, 0, SWP_NOMOVE | SWP_NOSIZE);
            break;
    }
}
```

Рис. 1-2. Программа-пример ErrorShow

Рис. 1-2. продолжение

```

case IDC_ERRORCODE:
    EnableWindow(GetDlgItem(hwnd, IDOK), Edit_GetTextLength(hwndCtl) > 0);
    break;

case IDOK:
    // получаем код ошибки
    DWORD dwError = GetDlgItemInt(hwnd, IDC_ERRORCODE, NULL, FALSE);

    HLOCAL hlocal = NULL;    // буфер для строки с описанием ошибки

    // получаем текстовое описание ошибки
    BOOL fOk = FormatMessage(
        FORMAT_MESSAGE_FROM_SYSTEM | FORMAT_MESSAGE_ALLOCATE_BUFFER,
        NULL, dwError, MAKELANGID(LANG_ENGLISH, SUBLANG_ENGLISH_US),
        (PTSTR) &hlocal, 0, NULL);

    if (!fOk) {
        // не связана ли ошибка с сетью?
        HMODULE hDll = LoadLibraryEx(TEXT("netmsg.dll"), NULL,
            DONT_RESOLVE_DLL_REFERENCES);

        if (hDll != NULL) {
            FormatMessage(
                FORMAT_MESSAGE_FROM_HMODULE | FORMAT_MESSAGE_FROM_SYSTEM
                | FORMAT_MESSAGE_ALLOCATE_BUFFER, hDll, dwError,
                MAKELANGID(LANG_ENGLISH, SUBLANG_ENGLISH_US), (PTSTR) &hlocal, 0, NULL);
            FreeLibrary(hDll);
        }
    }

    if (hlocal != NULL) {
        SetDlgItemText(hwnd, IDC_ERRORTEXT, (PCTSTR) LocalLock(hlocal));
        LocalFree(hlocal);
    } else {
        SetDlgItemText(hwnd, IDC_ERRORTEXT, TEXT("Error number not found."));
    }
    break;
}

///////////////////////////////
INT_PTR WINAPI Dlg_Proc(HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam) {

switch (uMsg) {
    chHANDLE_DLMSG(hwnd, WM_INITDIALOG, Dlg_OnInitDialog);
    chHANDLE_DLMSG(hwnd, WM_COMMAND, Dlg_OnCommand);

case ESM_POKECODEANDLOOKUP:
    SetDlgItemInt(hwnd, IDC_ERRORCODE, (UINT) wParam, FALSE);
    FORWARD_WM_COMMAND(hwnd, IDOK, GetDlgItem(hwnd, IDOK), BN_CLICKED,

```

см. след. стр.

Рис. 1-2. продолжение

```
        PostMessage();
        SetForegroundWindow(hwnd);
        break;
    }

    return(FALSE);
}

/////////////////////////////// Конец файла ///////////////////////////////
```

```
int WINAPI _tWinMain(HINSTANCE hinstExe, HINSTANCE, PTSTR pszCmdLine, int) {

    HWND hwnd = FindWindow(TEXT("#32770"), TEXT("Error Show"));
    if (IsWindow(hwnd)) {
        // экземпляр уже выполняется, активизируем его и посыпаем ему новый номер
        SendMessage(hwnd, ESM_POKECODEANDLOOKUP, _ttoi(pszCmdLine), 0);
    } else {
        DialogBoxParam(hinstExe, MAKEINTRESOURCE(IDD_ERRORSHOW),
            NULL, Dlg_Proc, _ttoi(pszCmdLine));
    }
    return(0);
}

/////////////////////////////// Конец файла ///////////////////////////////
```

Unicode

Microsoft Windows становится все популярнее, и нам, разработчикам, надо больше ориентироваться на международные рынки. Раньше считалось нормальным, что локализованные версии программных продуктов выходят спустя полгода после их появления в США. Но расширение поддержки в операционной системе множества самых разных языков упрощает выпуск программ, рассчитанных на международные рынки, и тем самым сокращает задержки с началом их дистрибуции.

В Windows всегда были средства, помогающие разработчикам локализовать свои приложения. Программа получает специфичную для конкретной страны информацию (региональные стандарты), вызывая различные функции Windows, и узнает предпочтения пользователя, анализируя параметры, заданные в Control Panel. Кроме того, Windows поддерживает массу всевозможных шрифтов.

Я решил переместить эту главу в начало книги, потому что вопрос о поддержке Unicode стал одним из основных при разработке любого приложения. Проблемы, связанные с Unicode, обсуждаются почти в каждой главе; все программы-примеры в моей книге «готовы к Unicode». Тот, кто пишет программы для Microsoft Windows 2000 или Microsoft Windows CE, просто обязан использовать Unicode, и точка. Но если Вы разрабатываете приложения для Microsoft Windows 98, у Вас еще есть выбор. В этой главе мы поговорим и о применении Unicode в Windows 98.

Наборы символов

Настоящей проблемой при локализации всегда были операции с различными наборами символов. Годами, кодируя текстовые строки как последовательности однобайтовых символов с нулем в конце, большинство программистов так к этому привыкло, что это стало чуть ли не второй их натурой. Вызываемая нами функция *strlen* возвращает количество символов в заканчивающемся нулем массиве однобайтовых символов. Но существуют такие языки и системы письменности (классический пример — японские иероглифы), в которых столько знаков, что одного байта, позволяющего кодировать не более 256 символов, просто недостаточно. Для поддержки подобных языков были созданы двухбайтовые наборы символов (double-byte character sets, DBCS).

Одно- и двухбайтовые наборы символов

В двухбайтовом наборе символ представляется либо одним, либо двумя байтами. Так, для японской кандзи, если значение первого байта находится между 0x81 и 0x9F или между 0xE0 и 0xFC, надо проверить значение следующего байта в строке, чтобы определить полный символ. Работа с двухбайтовыми наборами символов — просто кошмар для программиста, так как часть их состоит из одного байта, а часть — из двух.

Простой вызов функции *strlen* не дает количества символов в строке — она возвращает только число байтов. В ANSI-библиотеке С нет функций, работающих с двухбайтовыми наборами символов. Но в аналогичную библиотеку Visual C++ включено множество функций (типа *_mbslen*), способных оперировать со строками мультибайтовых (как одно-, так и двухбайтовых) символов.

Для работы с DBCS-строками в Windows предусмотрен целый набор вспомогательных функций:

Функция	Описание
<i>PTSTR CharNext (PCTSTR pszCurrentChar);</i>	Возвращает адрес следующего символа в строке
<i>PTSTR CharPrev (PCTSTR pszStart, PCTSTR pszCurrentChar);</i>	Возвращает адрес предыдущего символа в строке
<i>BOOL IsDBCSLeadByte (BYTE bTestChar);</i>	Возвращает TRUE, если данный байт — первый в DBCS-символе

Функции *CharNext* и *CharPrev* позволяют «перемещаться» по двухбайтовой строке единовременно на 1 символ вперед или назад, а *IsDBCSLeadByte* возвращает TRUE, если переданный ей байт — первый в двухбайтовом символе.

Хотя эти функции несколько облегчают работу с DBCS-строками, необходимость в ином подходе очевидна. Перейдем к Unicode.

Unicode: набор «широких» символов

Unicode — стандарт, первоначально разработанный Apple и Xerox в 1988 г. В 1991 г. был создан консорциум для совершенствования и внедрения Unicode. В него вошли компании Apple, Compaq, Hewlett-Packard, IBM, Microsoft, Oracle, Silicon Graphics, Sybase, Unisys и Xerox. (Полный список компаний — членов консорциума см. на www.Unicode.org.) Эта группа компаний наблюдает за соблюдением стандарта Unicode, описание которого Вы найдете в книге *The Unicode Standard* издательства Addison-Wesley (ее электронный вариант можно получить на том же www.Unicode.org).

Строки в Unicode просты и логичны. Все символы в них представлены 16-битными значениями (по 2 байта на каждый). В них нет особых байтов, указывающих, чем является следующий байт — частью того же символа или новым символом. Это значит, что прохождение по строке реализуется простым увеличением или уменьшением значения указателя. Функции *CharNext*, *CharPrev* и *IsDBCSLeadByte* больше не нужны.

Так как каждый символ — 16-битное число, Unicode позволяет кодировать 65 536 символов, что более чем достаточно для работы с любым языком. Разительное отличие от 256 знаков, доступных в однобайтовом наборе!

В настоящее время кодовые позиции¹ определены для арабского, китайского, греческого, еврейского, латинского (английского) алфавитов, а также для кириллицы (русского), японской каны, корейского хангыль и некоторых других алфавитов. Кроме того, в набор символов включено большое количество знаков препинания, математических и технических символов, стрелок, диакритических и других знаков. Все вместе они занимают около 35 000 кодовых позиций, оставляя простор для будущих расширений.

Эти 65 536 символов разбиты на отдельные группы. Некоторые группы, а также включенные в них символы показаны в таблице.

¹ Кодовая позиция (code point) — позиция знака в наборе символов.

16-битный код	Символы	16-битный код	Символы
0000–007F	ASCII	0300–036F	Общие диакритические
0080–00FF	Символы Latin1	0400–04FF	Кириллица
0100–017F	Европейские латинские	0530–058F	Армянский
0180–01FF	Расширенные латинские	0590–05FF	Еврейский
0250–02AF	Стандартные фонетические	0600–06FF	Арабский
02B0–02FF	Модифицированные литеры	0900–097F	Деванагари

Около 29 000 кодовых позиций пока не заняты, но зарезервированы на будущее. Примерно 6 000 позиций оставлено специально для программистов (на их усмотрение).

Почему Unicode?

Разрабатывая приложение, Вы определенно должны использовать преимущества Unicode. Даже если Вы пока не собираетесь локализовать программный продукт, разработка с прицелом на Unicode упростит эту задачу в будущем. Unicode также позволяет:

- легко обмениваться данными на разных языках;
- распространять единственный двоичный EXE- или DLL-файл, поддерживающий все языки;
- увеличить эффективность приложений (об этом мы поговорим чуть позже).

Windows 2000 и Unicode

Windows 2000 — операционная система, целиком и полностью построенная на Unicode. Все базовые функции для создания окон, вывода текста, операций со строками и т. д. ожидают передачи Unicode-строк. Если какой-то функции Windows передается ANSI-строка, она сначала преобразуется в Unicode и лишь потом передается операционной системе. Если Вы ждете результата функции в виде ANSI-строки, операционная система преобразует строку — перед возвратом в приложение — из Unicode в ANSI. Все эти преобразования протекают скрытно от Вас, но, конечно, на них тратятся и лишнее время, и лишняя память.

Например, функция *CreateWindowEx*, вызываемая с ANSI-строками для имени класса и заголовка окна, должна, выделив дополнительные блоки памяти (в стандартной куче Вашего процесса), преобразовать эти строки в Unicode и, сохранив результат в выделенных блоках памяти, вызвать Unicode-версию *CreateWindowEx*.

Для функций, заполняющих строки выделенные буферы, системе — прежде чем программа сможет их обрабатывать — нужно преобразовать строки из Unicode в ANSI. Из-за этого Ваше приложение потребует больше памяти и будет работать медленнее. Поэтому гораздо эффективнее разрабатывать программу, с самого начала ориентируясь на Unicode.

Windows 98 и Unicode

Windows 98 — не совсем новая операционная система. У нее «16-разрядное наследство», которое не было рассчитано на Unicode. Введение поддержки Unicode в Windows 98 было бы слишком трудоемкой задачей, и при разработке этой операционной системы от нее отказались. По этой причине вся внутренняя обработка строк в Windows 98, как и у ее предшественниц, построена на применении ANSI.

И все же Windows 98 допускает работу с приложениями, обрабатывающими символы и строки в Unicode, хотя вызов функций Windows при этом заметно усложняется. Например, если Вы, обращаясь к *CreateWindowEx*, передаете ей ANSI-строки, вызов проходит очень быстро — не требуется ни выделения буферов, ни преобразования строк. Но для вызова *CreateWindowEx* с Unicode-строками Вам придется самому выделять буфера, явно вызывать функции, преобразующие строки из Unicode в ANSI, обращаться к *CreateWindowEx*, снова вызывать функции, преобразующие строки — на этот раз из ANSI в Unicode, и освобождать временные буфера. Так что в Windows 98 работать с Unicode не столь удобно, как в Windows 2000. Подробнее о преобразованиях строк в Windows 98 я расскажу в конце главы.

Хотя большинство Unicode-функций в Windows 98 ничего не делает, некоторые все же реализованы. Вот они:

- `EnumResourceLanguagesW`
- `EnumResourceNamesW`
- `EnumResourceTypesW`
- `ExtTextOutW`
- `FindResourceW`
- `FindResourceExW`
- `GetCharWidthW`
- `GetCommandLineW`
- `GetTextExtentPoint32W`
- `GetTextExtentPointW`
- `lstrlenW`
- `MessageBoxExW`
- `MessageBoxW`
- `TextOutW`
- `WideCharToMultiByte`
- `MultiByteToWideChar`

К сожалению, многие из этих функций в Windows 98 работают из рук вон плохо. Одни не поддерживают определенные шрифты, другие повреждают область динамически распределяемой памяти (кучу), третьи нарушают работу принтерных драйверов и т. д. С этими функциями Вам придется здорово потрудиться при отладке программы. И даже это еще не значит, что Вы сможете устранить все проблемы.

Windows CE и Unicode

Операционная система Windows CE создана для небольших вычислительных устройств — бездисковых и с малым объемом памяти. Вы вполне могли бы подумать, что Microsoft, раз уж эту систему нужно было сделать предельно компактной, в качестве «родного» набора символов выберет ANSI. Но Microsoft поступила дальновиднее. Зная, что вычислительные устройства с Windows CE будут продаваться по всему миру, там решили сократить затраты на разработку программ, упростив их локализацию. Поэтому Windows CE полностью поддерживает Unicode.

Чтобы не увеличивать ядро Windows CE, Microsoft вообще отказалась от поддержки ANSI-функций Windows. Так что, если Вы пишете для Windows CE, то просто обязаны разбираться в Unicode и использовать его во всех частях своей программы.

В чью пользу счет?

Для тех, кто ведет счет в борьбе Unicode против ANSI, я решил сделать краткий обзор «История Unicode в Microsoft»:

- Windows 2000 поддерживает Unicode и ANSI — Вы можете использовать любой стандарт;
- Windows 98 поддерживает только ANSI — Вы обязаны программировать в расчете на ANSI;

- Windows CE поддерживает только Unicode — Вы обязаны программировать в расчете на Unicode.

Несмотря на то что Microsoft пытается облегчить написание программ, способных работать на всех трех платформах, различия между Unicode и ANSI все равно создают проблемы, и я сам не раз с ними сталкивался. Не поймите меня неправильно, но Microsoft твердо поддерживает Unicode, поэтому я настоятельно рекомендую переходить именно на этот стандарт. Только имейте в виду, что Вас ждут трудности, на преодоление которых потребуется время. Я бы посоветовал применять Unicode и, если Вы работаете в Windows 98, преобразовывать строки в ANSI лишь там, где без этого не обойтись.

Увы, есть еще одна маленькая проблема, о которой Вы должны знать, — COM.

Unicode и COM

Когда Microsoft переносила COM из 16-разрядной Windows на платформу Win32, руководство этой компании решило, что все методы COM-интерфейсов, работающие со строками, должны принимать их только в Unicode. Это было удачное решение, так как COM обычно используется для того, чтобы компоненты могли общаться друг с другом, а Unicode позволяет легко локализовать строки.

Если Вы разрабатываете программу для Windows 2000 или Windows CE и при этом используете COM, то выбора у Вас просто нет. Применяя Unicode во всех частях программы, Вам будет гораздо проще обращаться и к операционной системе, и к COM-объектам.

Если Вы пишете для Windows 98 и тоже используете COM, то попадаете в затруднительное положение. COM требует строк в Unicode, а большинство функций операционной системы — строк в ANSI. Это просто кошмар! Я работал над несколькими такими проектами, и мне приходилось писать прорыв кода только для того, чтобы гонять строки из одного формата в другой.

Как писать программу с использованием Unicode

Microsoft разработала Windows API так, чтобы как можно меньше влиять на Ваш код. В самом деле, у Вас появилась возможность создать единственный файл с исходным кодом, компилируемый как с применением Unicode, так и без него, — достаточно определить два макроса (`UNICODE` и `_UNICODE`), которые отвечают за нужные изменения.

Unicode и библиотека С

Для использования Unicode-строк были введены некоторые новые типы данных. Стандартный заголовочный файл `String.h` модифицирован: в нем определен `wchar_t` — тип данных для Unicode-символа:

```
typedef unsigned short wchar_t;
```

Если Вы хотите, скажем, создать буфер для хранения Unicode-строки длиной до 99 символов с нулевым символом в конце, поставьте оператор:

```
wchar_t szBuffer[100];
```

Он создает массив из ста 16-битных значений. Конечно, стандартные функции библиотеки С для работы со строками вроде `strcpy`, `strchr` и `strcat` оперируют только с ANSI-строками — они не способны корректно обрабатывать Unicode-строки. Поэтому

му в ANSI C имеется дополнительный набор функций. На рис. 2-1 приведен список строковых функций ANSI C и эквивалентных им Unicode-функций.

```
char * strcat(char *, const char *);
wchar_t * wcscat(wchar_t *, const wchar_t *);

char * strchr(const char *, int);
wchar_t * wcschr(const wchar_t *, wchar_t *);

int strcmp(const char *, const char *);
int wcscmp(const wchar_t *, const wchar_t *);

char * strcpy(char *, const char *);
wchar_t * wcscpy(wchar_t *, const wchar_t *);

size_t strlen(const char *);
size_t wcslen(const wchar_t *);
```

Рис. 2-1. Строковые функции ANSI C и их Unicode-аналоги

Обратите внимание, что имена всех новых функций начинаются с *wcs* — это аббревиатура *wide character set* (набор широких символов). Таким образом, имена Unicode-функций образуются простой заменой префикса *str* соответствующих ANSI-функций на *wcs*.



Один очень важный момент, о котором многие забывают, заключается в том, что библиотека C, предоставляемая Microsoft, отвечает стандарту ANSI. А он требует, чтобы библиотека C поддерживала символы и строки в Unicode. Это значит, что Вы можете вызывать функции C для работы с Unicode-символами и строками даже при работе в Windows 98. Иными словами, функции *wcscat*, *wcslen*, *wcstok* и т. д. прекрасно работают и в Windows 98; беспокоиться нужно за функции операционной системы.

Код, содержащий явные вызовы *str*- или *wcs*-функций, просто так компилировать с использованием и ANSI, и Unicode нельзя. Чтобы реализовать возможность компиляции «двойного назначения», замените в своей программе заголовочный файл String.h на TChar.h. Он помогает создавать универсальный исходный код, способный задействовать как ANSI, так и Unicode, — и это единственное, для чего нужен файл TChar.h. Он состоит из макросов, заменяющих явные вызовы *str*- или *wcs*-функций. Если при компиляции исходного кода Вы определяете _UNICODE, макросы ссылаются на *wcs*-функции, а в его отсутствие — на *str*-функции.

Например, в TChar.h есть макрос *_tcscpy*. Если Вы включили этот заголовочный файл, но _UNICODE не определен, *_tcscpy* раскрывается в ANSI-функцию *strcpy*, а если _UNICODE определен — в Unicode-функцию *wcscpy*. В файле TChar.h определены универсальные макросы для всех стандартных строковых функций С. При использовании этих макросов вместо конкретных имен ANSI- или Unicode-функций Ваш код можно будет компилировать в расчете как на Unicode, так и на ANSI.

Но, к сожалению, это еще не все. В файле TChar.h есть дополнительные макросы.

Чтобы объявить символьный массив «универсального назначения» (ANSI/Unicode), применяется тип данных TCHAR. Если _UNICODE определен, TCHAR объявляется так:

```
typedef wchar_t TCHAR;
```

А если _UNICODE не определен, то:

```
typedef char TCHAR;
```

Используя этот тип данных, можно объявить строку символов как:

```
TCHAR szString[100];
```

Можно также определять указатели на строки:

```
TCHAR *szError = "Error";
```

Правда, в этом операторе есть одна проблема. По умолчанию компилятор Microsoft C++ транслирует строки как состоящие из символов ANSI, а не Unicode. В итоге этот оператор normally компилируется, если `_UNICODE` не определен, но в ином случае дает ошибку. Чтобы компилятор сгенерировал Unicode-, а не ANSI-строку, оператор надо переписать так:

```
TCHAR *szError = L"Error";
```

Заглавная буква `L` перед строковым литералом указывает компилятору, что его надо обрабатывать как Unicode-строку. Тогда, размещая строку в области данных программы, компилятор вставит между всеми символами нулевые байты. Но возникает другая проблема — программа компилируется, только если `_UNICODE` определен. Следовательно, нужен макрос, способный избирательно ставить `L` перед строковым литералом. Эту работу выполняет макрос `_TEXT`, также содержащийся в `Tchar.h`. Если `_UNICODE` определен, `_TEXT` определяется как:

```
#define _TEXT(x) L ## x
```

В ином случае `_TEXT` определяется следующим образом:

```
#define _TEXT(x) x
```

Используя этот макрос, перепишем злополучный оператор так, чтобы его можно было корректно компилировать независимо от того, определен `_UNICODE` или нет:

```
TCHAR *szError = _TEXT("Error");
```

Макрос `_TEXT` применяется и для символьных литералов. Например, чтобы проверить, является ли первый символ строки заглавной буквой `J`:

```
if (szError[0] == _TEXT('J')) {
    // первый символ - J
    :
} else {
    // первый символ - не J
    :
}
```

Типы данных, определенные в Windows для Unicode

В заголовочных файлах Windows определены следующие типы данных.

Тип данных	Описание
WCHAR	Unicode-символ
PWSTR	Указатель на Unicode-строку
PCWSTR	Указатель на строковую константу в Unicode

Эти типы данных относятся исключительно к символам и строкам в кодировке Unicode. В заголовочных файлах Windows определены также универсальные (ANSI/

Unicode) типы данных PTSTR и PCTSTR, указывающие — в зависимости от того, определен ли при компиляции макрос UNICODE, — на ANSI- или на Unicode-строку.

Кстати, на этот раз имя макроя UNICODE не предваряется знаком подчеркивания. Дело в том, что макроя _UNICODE используется в заголовочных файлах библиотеки C, а макроя UNICODE — в заголовочных файлах Windows. Для компиляции модулей исходного кода обычно приходится определять оба макроя.

Unicode- и ANSI-функции в Windows

Я уже говорил, что существует две функции *CreateWindowEx*: одна принимает строки в Unicode, другая — в ANSI. Все так, но в действительности прототипы этих функций чуть-чуть отличаются:

```
HWND WINAPI CreateWindowExW(
    DWORD dwExStyle,
    PCWSTR pClassName,
    PCWSTR pWindowName,
    DWORD dwStyle,
    int X,
    int Y,
    int nWidth,
    int nHeight,
    HWND hWndParent,
    HMENU hMenu,
    HINSTANCE hInstance,
    PVOID pParam);
```

```
HWND WINAPI CreateWindowExA(
    DWORD dwExStyle,
    PCSTR pClassName,
    PCSTR pWindowName,
    DWORD dwStyle,
    int X,
    int Y,
    int nWidth,
    int nHeight,
    HWND hWndParent,
    HMENU hMenu,
    HINSTANCE hInstance,
    PVOID pParam);
```

CreateWindowExW — это Unicode-версия. Буква *W* в конце имени функции — аббревиатура слова *wide* (широкий). Символы Unicode занимают по 16 битов каждый, поэтому их иногда называют широкими символами (wide characters). Буква *A* в конце имени *CreateWindowExA* указывает, что данная версия функции принимает ANSI-строки.

Но обычно *CreateWindowExW* или *CreateWindowExA* напрямую не вызывают, а обращаются к *CreateWindowEx* — макроя, определенному в файле WinUser.h:

```
#ifdef UNICODE
#define CreateWindowEx CreateWindowExW
#else
#define CreateWindowEx CreateWindowExA
#endif // !UNICODE
```

Какая именно версия *CreateWindowEx* будет вызвана, зависит от того, определен ли UNICODE в период компиляции. Перенося 16-разрядное Windows-приложение на платформу Win32, Вы, вероятно, не станете определять UNICODE. Тогда все вызовы *CreateWindowEx* будут преобразованы в вызовы *CreateWindowExA* — ANSI-версии функций. И перенос приложения упростится, ведь 16-разрядная Windows работает только с ANSI-версией *CreateWindowEx*.

В Windows 2000 функция *CreateWindowExA* — просто шлюз (транслятор), который выделяет память для преобразования строк из ANSI в Unicode и вызывает *CreateWindowExW*, передавая ей преобразованные строки. Когда *CreateWindowExW* вернет управление, *CreateWindowExA* освободит буферы и передаст Вам описатель окна.

Разрабатывая DLL, которую будут использовать и другие программисты, предусматривайте в ней по две версии каждой функции — для ANSI и для Unicode. В ANSI-версии просто выделяйте память, преобразуйте строки и вызывайте Unicode-версию той же функции. (Этот процесс я продемонстрирую позже.)

В Windows 98 основную работу выполняет *CreateWindowExA*. В этой операционной системе предусмотрены точки входа для всех функций Windows, принимающих Unicode-строки, но функции не транслируют их в ANSI, а просто сообщают об ошибке. Последующий вызов *GetLastError* дает ERROR_CALL_NOT_IMPLEMENTED. Должным образом действуют только ANSI-версии функций. Ваше приложение не будет работать в Windows 98, если в скомпилированном коде присутствуют вызовы «широкосимвольных» функций.

Некоторые функции Windows API (например, *WinExec* или *OpenFile*) существуют только для совместимости с 16-разрядными программами, и их надо избегать. Лучше заменить все вызовы *WinExec* и *OpenFile* вызовами *CreateProcess* и *CreateFile* соответственно. Тем более, что старые функции просто обращаются к новым. Самая серьезная проблема с ними в том, что они не принимают строки в Unicode, — при их вызове Вы должны передавать строки в ANSI. С другой стороны, в Windows 2000 у всех новых или пока не устаревших функций обязательно есть как ANSI-, так и Unicode-версия.

Строковые функции Windows

Windows предлагает внушительный набор функций, работающих со строками. Они похожи на строковые функции из библиотеки C, например на *strcpy* и *wcscpy*. Однако функции Windows являются частью операционной системы, и многие ее компоненты используют именно их, а не аналоги из библиотеки C. Я советую отдать предпочтение функциям операционной системы. Это немножко повысит быстродействие Вашего приложения. Дело в том, что к ним часто обращаются такие тяжеловесные процессы, как оболочка операционной системы (Explorer.exe), и скорее всего эти функции будут загружены в память еще до запуска Вашего приложения.

Данные функции доступны в Windows 2000 и Windows 98. Но Вы сможете вызывать их и в более ранних версиях Windows, если установите Internet Explorer версии 4.0 или выше.

По классической схеме именования функций в операционных системах их имена состоят из символов нижнего и верхнего регистра и выглядят так: *StrCat*, *StrChr*, *StrCmp*, *StrCpy* и т. д. Для использования этих функций включите в программу заголовочный файл ShlwApi.h. Кроме того, как я уже говорил, каждая строковая функция существует в двух версиях — для ANSI и для Unicode (например, *StrCatA* и *StrCatW*). Поскольку это функции операционной системы, их имена автоматически преобразуются в нужную форму, если в исходном тексте Вашей программы перед ее сборкой определен UNICODE.

Создание программ, способных использовать и ANSI, и Unicode

Неплохая мысль — заранее подготовить свое приложение к Unicode, даже если Вы пока не планируете работать с этой кодировкой. Вот главное, что для этого нужно:

- привыкайте к тому, что текстовые строки — это массивы символов, а не массивы байтов или значений типа *char*;
- используйте универсальные типы данных (вроде TCHAR или PTSTR) для текстовых символов и строк;
- используйте явные типы данных (вроде BYTE или PBYTE) для байтов, указателей на байты и буферов данных;
- применяйте макрос _TEXT для определения символьных и строковых литералов;
- предусмотрите возможность глобальных замен (например, PSTR на PTSTR);
- модифицируйте логику строковой арифметики. Например, функции обычно принимают размер буфера в символах, а не в байтах. Это значит, что вместо *sizeof(szBuffer)* Вы должны передавать (*sizeof(szBuffer)* / *sizeof(TCHAR)*). Но блок памяти для строки известной длины выделяется в байтах, а не символах, т. е. вместо *malloc(nCharacters)* нужно использовать *malloc(nCharacters * sizeof(TCHAR))*. Из всего, что я перечислил, это запомнить труднее всего — если Вы ошибетесь, компилятор не выдаст никаких предупреждений.

Разрабатывая программы-примеры для первого издания книги, я сначала написал их так, что они компилировались только с использованием ANSI. Но, дойдя до этой главы (она была тогда в конце), понял, что Unicode лучше, и решил написать примеры, которые показывали бы, как легко создавать программы, компилируемые с применением и Unicode, и ANSI. В конце концов я преобразовал все программы-примеры так, чтобы их можно было компилировать в расчете на любой из этих стандартов.

Конверсия всех программ заняла примерно 4 часа — неплохо, особенно если учсть, что у меня совсем не было опыта в этом деле.

В Windows есть набор функций для работы с Unicode-строками. Эти функции перечислены ниже.

Функция	Описание
<i>lstrcmp</i>	Выполняет конкатенацию строк
<i>lstrcmpi</i>	Сравнивает две строки с учетом регистра букв
<i>lstrcmpi</i>	Сравнивает две строки без учета регистра букв
<i>lstrcpy</i>	Копирует строку в другой участок памяти
<i>lstrlen</i>	Возвращает длину строки в символах

Они реализованы как макросы, вызывающие либо Unicode-, либо ANSI-версию функции в зависимости от того, определен ли UNICODE при компиляции исходного модуля. Например, если UNICODE не определен, *lstrcmp* раскрывается в *lstrcmpA*, определен — в *lstrcmpW*.

Строковые функции *lstrcmp* и *lstrcmpi* ведут себя не так, как их аналоги из библиотеки C (*strcmp*, *strncpy*, *wcsstrcmp* и *wcsncpy*), которые просто сравнивают кодовые позиции в символах строк. Игнорируя фактические символы, они сравнивают числовое значение каждого символа первой строки с числовым значением символа второй

строки. Но *lstrcmp* и *lstrcmpi* реализованы через вызовы Windows-функции *CompareString*:

```
int CompareString(
    LCID lcid,
    DWORD fdwStyle,
    PCWSTR pString1,
    int cch1,
    PCWSTR pString2,
    int cch2);
```

Она сравнивает две Unicode-строки. Первый параметр задает так называемый идентификатор локализации (locale ID, LCID) — 32-битное значение, определяющее конкретный язык. С помощью этого идентификатора *CompareString* сравнивает строки с учетом значения конкретных символов в данном языке. Так что она действует куда осмысленнее, чем функции библиотеки С.

Когда любая из функций семейства *lstrcmp* вызывает *CompareString*, в первом параметре передается результат вызова Windows-функции *GetThreadLocale*:

```
LCID GetThreadLocale();
```

Она возвращает уже упомянутый идентификатор, который назначается потоку в момент его создания.

Второй параметр функции *CompareString* указывает флаги, модифицирующие метод сравнения строк. Допустимые флаги перечислены в следующей таблице.

Флаг	Действие
NORM_IGNORECASE	Различия в регистре букв игнорируются
NORM_IGNOREKANATYPE	Различия между знаками хираганы и катаканы игнорируются
NORM_IGNORENONSPACE	Знаки, отличные от пробелов, игнорируются
NORM_IGNORESYMBOLS	Символы, отличные от алфавитно-цифровых, игнорируются
NORM_IGNOREWIDTH	Разница между одно- и двухбайтовым представлением одного и того же символа игнорируется
SORT_STRINGSORT	Знаки препинания обрабатываются так же, как и символы, отличные от алфавитно-цифровых

Вызывая *CompareString*, функция *lstrcmp* передает в параметре *fdwStyle* нуль, а *lstrcmpi* — флаг NORM_IGNORECASE. Остальные четыре параметра определяют две строки и их длину. Если *cch1* равен –1, функция считает, что строка *pString1* завершается нулевым символом, и автоматически вычисляет ее длину. То же относится и к параметрам *cch2* и *pString2*.

Многие функции С-библиотеки с Unicode-строками толком не работают. Так, *tolower* и *toupper* неправильно преобразуют регистр букв со знаками ударения. Поэтому для Unicode-строк лучше использовать соответствующие Windows-функции. К тому же они корректно работают и с ANSI-строками.

Первые две функции:

```
PTSTR CharLower(PTSTR pszString);
```

```
PTSTR CharUpper(PTSTR pszString);
```

преобразуют либо отдельный символ, либо целую строку с нулевым символом в конце. Чтобы преобразовать всю строку, просто передайте ее адрес. Но, преобразуя отдельный символ, Вы должны передать его так:

```
TCHAR cLowerCaseChr = CharLower((PTSTR) szString[0]);
```

Приведение типа отдельного символа к PTSTR вызывает обнуление старших 16 битов передаваемого значения, а в его младшие 16 битов помещается сам символ. Обнаружив, что старшие 16 битов этого значения равны 0, функция «поймет», что Вы хотите преобразовать не строку, а отдельный символ. Возвращаемое 32-битное значение содержит результат преобразования в младших 16 битах.

Следующие две функции аналогичны двум предыдущим за исключением того, что они преобразуют символы, содержащиеся в буфере (который не требуется завершать нулевым символом):

```
DWORD CharLowerBuff(  
    PTSTR pszString,  
    DWORD cchString);
```

```
DWORD CharUpperBuff(  
    PTSTR pszString,  
    DWORD cchString);
```

Прочие функции библиотеки С (например, *isalpha*, *islower* и *isupper*) возвращают значение, которое сообщает, является ли данный символ буквой, а также строчная она или прописная. В Windows API тоже есть подобные функции, но они учитывают и язык, выбранный пользователем в Control Panel:

```
BOOL IsCharAlpha(TCHAR ch);  
BOOL IsCharAlphaNumeric(TCHAR ch);  
BOOL IsCharLower(TCHAR ch);  
BOOL IsCharUpper(TCHAR ch);
```

И последняя группа функций из библиотеки С, о которых я хотел рассказать, — *printf*. Если при компиляции _UNICODE определен, они ожидают передачи всех символьных и строковых параметров в Unicode; в ином случае — в ANSI.

Microsoft ввела в семейство функций *printf* своей С-библиотеки дополнительные типы полей, часть из которых не поддерживается в ANSI C. Они позволяют легко сравнивать и смешивать символы и строки с разной кодировкой. Также расширена функция *wsprintf* операционной системы. Вот несколько примеров (обратите внимание на использование буквы *s* в верхнем и нижнем регистре):

```
char szA[100]; // строковый буфер в ANSI  
WCHAR szW[100]; // строковый буфер в Unicode  
  
// обычный вызов sprintf: все строки в ANSI  
sprintf(szA, "%s", "ANSI Str");  
  
// преобразуем строку из Unicode в ANSI  
sprintf(szA, "%S", L"Unicode Str");  
  
// обычный вызов swprintf: все строки в Unicode  
swprintf(szW, L"%s", L"Unicode Str");  
  
// преобразуем строку из ANSI в Unicode  
swprintf(szW, L"%S", "ANSI Str");
```

Ресурсы

Компилятор ресурсов генерирует двоичное представление всех ресурсов, используемых Вашей программой. Строки в ресурсах (таблицы строк, шаблоны диалоговых окон, меню и др.) всегда записываются в Unicode. Если в программе не определяется макрос UNICODE, Windows 98 и Windows 2000 сами проводят нужные преобразования. Например, если при компиляции исходного модуля UNICODE не определен, вызов *LoadString* на самом деле приводит к вызову *LoadStringA*, которая читает строку из ресурсов и преобразует ее в ANSI. Затем Вашей программе возвращается ANSI-представление строки.

Текстовые файлы

Текстовых файлов в кодировке Unicode пока очень мало. Ни в одном текстовом файле, поставляемом с операционными системами или другими программными продуктами Microsoft, не используется Unicode. Думаю, однако, что эта тенденция изменится в будущем — пусть даже в отдаленном. Например, программа Notepad в Windows 2000 позволяет создавать или открывать как Unicode-, так и ANSI-файлы. Посмотрите на ее диалоговое окно Save As (рис. 2-2) и обратите внимание на предлагаемые форматы текстовых файлов.

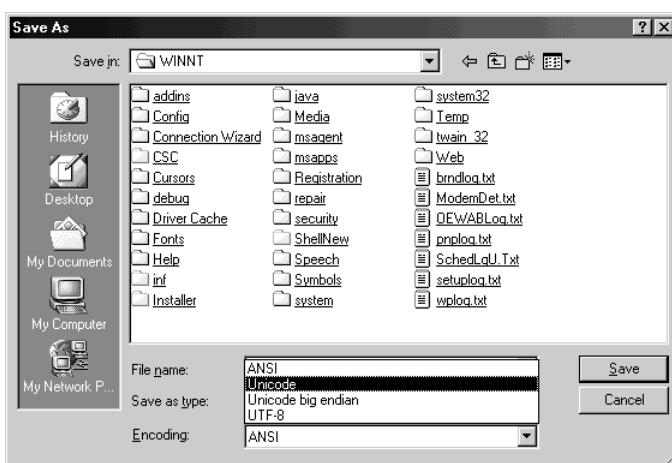


Рис. 2-2. Диалоговое окно Save As программы Notepad в Windows 2000

Многим приложениям, которые открывают и обрабатывают текстовые файлы (например, компиляторам), было бы удобнее, если после открытия файла можно было бы определить, содержит он символы в ANSI или в Unicode. В этом может помочь функция *IsUnicode*:

```
DWORD IsTextUnicode(CONST PVOID pvBuffer, int cb, PINT pResult);
```

Проблема с текстовыми файлами в том, что не существует четких и строгих правил относительно их содержимого. Это крайне затрудняет определение того, содержит файл символы в ANSI или в Unicode. Поэтому *IsUnicode* применяет набор статистических и детерминистских методов для того, чтобы сделать взвешенное предположение о содержимом буфера. Поскольку тут больше алхимии, чем точной науки, нет гарантий, что Вы не получите неверные результаты от *IsUnicode*.

Первый ее параметр, *pvBuffer*, указывает на буфер, подлежащий проверке. При этом используется указатель типа void, поскольку неизвестно, в какой кодировке данный массив символов.

Параметр *cb* определяет число байтов в буфере, на который указывает *pvBuffer*. Так как содержимое буфера не известно, *cb* — счетчик именно байтов, а не символов. Заметьте: вовсе не обязательно задавать всю длину буфера. Но чем больше байтов проанализирует функция, тем больше шансов получить правильный результат.

Параметр *pResult* — это адрес целочисленной переменной, которую надо инициализировать перед вызовом функции. Ее значение сообщает, какие тесты должна пройти *IsUnicode*. Если *pResult* равен NULL, функция *IsUnicode* делает все проверки. (Подробнее об этом см. документацию Platform SDK.)

Функция возвращает TRUE, если считает, что буфер содержит текст в Unicode, и FALSE — в ином случае. Да-да, она возвращает именно булево значение, хотя в прототипе указано DWORD. Если через целочисленную переменную, на которую указывает *pResult*, были запрошены лишь определенные тесты, функция (перед возвратом управления) устанавливает ее биты в соответствии с результатами этих тестов.

WINDOWS 98 В Windows 98 функция *IsUnicode* по сути не реализована и просто возвращает FALSE; последующий вызов *GetLastError* дает код ошибки ERROR_CALL_NOT_IMPLEMENTED.

Применение функции *IsUnicode* иллюстрирует программа-пример FileRev (см. главу 17).

Перекодировка строк из Unicode в ANSI и обратно

Windows-функция *MultiByteToWideChar* преобразует мультибайтовые символы строки в «широкобайтовые»:

```
int MultiByteToWideChar(
    UINT uCodePage,
    DWORD dwFlags,
    PCSTR pMultiByteStr,
    int cchMultiByte,
    PWSTR pWideCharStr,
    int cchWideChar);
```

Параметр *uCodePage* задает номер кодовой страницы, связанной с мультибайтовой строкой. Параметр *dwFlags* влияет на преобразование букв с диакритическими знаками. Обычно эти флаги не используются, и *dwFlags* равен 0. Параметр *pMultiByteStr* указывает на преобразуемую строку, а *cchMultiByte* определяет ее длину в байтах. Функция самостоятельно определяет длину строки, если *cchMultiByte* равен –1.

Unicode-версия строки, полученная в результате преобразования, записывается в буфер по адресу, указанному в *pWideCharStr*. Максимальный размер этого буфера (в символах) задается в параметре *cchWideChar*. Если он равен 0, функция ничего не преобразует, а просто возвращает размер буфера, необходимого для сохранения результата преобразования. Обычно конверсия мультибайтовой строки в ее Unicode-эквивалент проходит так:

1. Вызывают *MultiByteToWideChar*, передавая NULL в параметре *pWideCharStr* и 0 в параметре *cchWideChar*.
2. Выделяют блок памяти, достаточный для сохранения преобразованной строки. Его размер получают из предыдущего вызова *MultiByteToWideChar*.

3. Снова вызывают *MultiByteToWideChar*, на этот раз передавая адрес выделенного буфера в параметре *pWideCharStr*, а размер буфера, полученный при первом обращении к этой функции, — в параметре *cchWideChar*.
4. Работают с полученной строкой.
5. Освобождают блок памяти, занятый Unicode-строкой.

Обратное преобразование выполняет функция *WideCharToMultiByte*:

```
int WideCharToMultiByte(
    UINT uCodePage,
    DWORD dwFlags,
    PCWSTR pWideCharStr,
    int cchWideChar,
    PSTR pMultiByteStr,
    int cchMultiByte,
    PCSTR pDefaultChar,
    PBOOL pfUsedDefaultChar);
```

Она очень похожа на *MultiByteToWideChar*. И опять *uCodePage* определяет кодовую страницу для строки — результата преобразования. Дополнительный контроль над процессом преобразования дает параметр *dwFlags*. Его флаги влияют на символы с диакритическими знаками и на символы, которые система не может преобразовать. Такой уровень контроля обычно не нужен, и *dwFlags* приравнивается 0.

Параметр *pWideCharStr* указывает адрес преобразуемой строки, а *cchWideChar* задает ее длину в символах. Функция сама определяет длину исходной строки, если *cchWideChar* равен –1.

Мультибайтовый вариант строки, полученный в результате преобразования, записывается в буфер, на который указывает *pMultiByteStr*. Параметр *cchMultiByte* определяет максимальный размер этого буфера в байтах. Передав нулевое значение в *cchMultiByte*, Вы заставите функцию сообщить размер буфера, требуемого для записи результата. Обычно конверсия широкобайтовой строки в мультибайтовую проходит в той же последовательности, что и при обратном преобразовании.

Очевидно, Вы заметили, что *WideCharToMultiByte* принимает на два параметра больше, чем *MultiByteToWideChar*; это *pDefaultChar* и *pfUsedDefaultChar*. Функция *WideCharToMultiByte* использует их, только если встречает широкий символ, не представленный в кодовой странице, на которую ссылается *uCodePage*. Если его преобразование невозможно, функция берет символ, на который указывает *pDefaultChar*. Если этот параметр равен NULL (как обычно и бывает), функция использует системный символ по умолчанию. Таким символом обычно служит знак вопроса, что при операциях с именами файлов очень опасно, поскольку он является и символом подстановки.

Параметр *pfUsedDefaultChar* указывает на переменную типа BOOL, которую функция устанавливает как TRUE, если хоть один символ из широкосимвольной строки не преобразован в свой мультибайтовый эквивалент. Если же все символы преобразованы успешно, функция устанавливает переменную как FALSE. Обычно Вы передаете NULL в этом параметре.

Подробнее эти функции и их применение описаны в документации Platform SDK.

Эти две функции позволяют легко создавать ANSI- и Unicode-версии других функций, работающих со строками. Например, у Вас есть DLL, содержащая функцию, которая переставляет все символы строки в обратном порядке. Unicode-версию этой функции можно было бы написать следующим образом.

```
BOOL StringReverseW(PWSTR pWideCharStr) {  
  
    // получаем указатель на последний символ в строке  
    PWSTR pEndOfStr = pWideCharStr + wcslen(pWideCharStr) - 1;  
    wchar_t cCharT;  
    // повторяем, пока не дойдем до середины строки  
    while (pWideCharStr < pEndOfStr) {  
        // записываем символ во временную переменную  
        cCharT = *pWideCharStr;  
  
        // помещаем последний символ на место первого  
        *pWideCharStr = *pEndOfStr;  
  
        // копируем символ из временной переменной на место  
        // последнего символа  
        *pEndOfStr = cCharT;  
  
        // продвигаемся на 1 символ вправо  
        pWideCharStr++;  
  
        // продвигаемся на 1 символ влево  
        pEndOfStr--;  
    }  
  
    // строка обращена; сообщаем об успешном завершении  
    return(TRUE);  
}
```

ANSI-версию этой функции можно написать так, чтобы она вообще ничем не занималась, а просто преобразовывала ANSI-строку в Unicode, передавала ее в функцию *StringReverseW* и конвертировала обращенную строку снова в ANSI. Тогда функция должна выглядеть примерно так:

```
BOOL StringReverseA(PSTR pMultiByteStr) {  
    PWSTR pWideCharStr;  
    int nLenOfWideCharStr;  
    BOOL fOk = FALSE;  
  
    // вычисляем количество символов, необходимых  
    // для хранения широкосимвольной версии строки  
    nLenOfWideCharStr = MultiByteToWideChar(CP_ACP, 0,  
        pMultiByteStr, -1, NULL, 0);  
  
    // Выделяем память из стандартной кучи процесса,  
    // достаточную для хранения широкосимвольной строки.  
    // Не забудьте, что MultiByteToWideChar возвращает  
    // количество символов, а не байтов, поэтому мы должны  
    // умножить это число на размер широкого символа.  
    pWideCharStr = HeapAlloc(GetProcessHeap(), 0,  
        nLenOfWideCharStr * sizeof(WCHAR));  
  
    if (pWideCharStr == NULL)  
        return(fOk);
```

```
// преобразуем мультибайтовую строку в широкосимвольную
MultiByteToWideChar(CP_ACP, 0, pMultiByteStr, -1,
    pWideCharStr, nLenOfWideCharStr);

// вызываем широкосимвольную версию этой функции
// для выполнения настоящей работы
fOk = StringReverseW(pWideCharStr);

if (fOk) {
    // преобразуем широкосимвольную строку обратно в мультибайтовую
    WideCharToMultiByte(CP_ACP, 0, pWideCharStr, -1,
        pMultiByteStr, strlen(pMultiByteStr), NULL, NULL);
}

// освобождаем память, выделенную под широкобайтовую строку
HeapFree(GetProcessHeap(), 0, pWideCharStr);

return(fOk);
}
```

И, наконец, в заголовочном файле, поставляемом вместе с DLL, прототипы этих функций были бы такими:

```
BOOL StringReverseW (PWSTR pWideCharStr);
BOOL StringReverseA (PSTR pMultiByteStr);

#ifndef UNICODE
#define StringReverse StringReverseW
#else
#define StringReverse StringReverseA
#endif // !UNICODE
```

Объекты ядра

Изучение Windows API мы начнем с объектов ядра и их описателей (handles). Эта глава посвящена сравнительно абстрактным концепциям, т. е. мы, не углубляясь в специфику тех или иных объектов ядра, рассмотрим их общие свойства.

Я бы предпочел начать с чего-то более конкретного, но без четкого понимания объектов ядра Вам не стать настоящим профессионалом в области разработки Windows-программ. Эти объекты используются системой и нашими приложениями для управления множеством самых разных ресурсов: процессами, потоками, файлами и т. д. Концепции, представленные здесь, будут встречаться на протяжении всей книги. Однако я прекрасно понимаю, что часть материалов не уляжется у Вас в голове до тех пор, пока Вы не приступите к работе с объектами ядра, используя реальные функции. И при чтении последующих глав книги Вы, наверное, будете время от времени возвращаться к этой главе.

Что такое объект ядра

Создание, открытие и прочие операции с объектами ядра станут для Вас, как разработчика Windows-приложений, повседневной рутиной. Система позволяет создавать и оперировать с несколькими типами таких объектов, в том числе: маркерами доступа (access token objects), файлами (file objects), проекциями файлов (file-mapping objects), портами завершения ввода-вывода (I/O completion port objects), заданиями (job objects), почтовыми ящиками (mailslot objects), мьютексами (mutex objects), каналами (pipe objects), процессами (process objects), семафорами (semaphore objects), потоками (thread objects) и ожидающими таймерами (waitable timer objects). Эти объекты создаются Windows-функциями. Например, *CreateFileMapping* заставляет систему сформировать объект «проекция файла». Каждый объект ядра — на самом деле просто блок памяти, выделенный ядром и доступный только ему. Этот блок представляет собой структуру данных, в элементах которой содержится информация об объекте. Некоторые элементы (дескриптор защиты, счетчик числа пользователей и др.) присутствуют во всех объектах, но большая их часть специфична для объектов конкретного типа. Например, у объекта «процесс» есть идентификатор, базовый приоритет и код завершения, а у объекта «файл» — смещение в байтах, режим разделения и режим открытия.

Поскольку структуры объектов ядра доступны только ядру, приложение не может самостоятельно найти эти структуры в памяти и напрямую модифицировать их содержимое. Такое ограничение Microsoft ввела намеренно, чтобы ни одна программа не нарушила целостность структур объектов ядра. Это же ограничение позволяет Microsoft вводить, убирать или изменять элементы структур, не нарушая работы каких-либо приложений.

Но вот вопрос: если мы не можем напрямую модифицировать эти структуры, то как же наши приложения оперируют с объектами ядра? Ответ в том, что в Windows

предусмотрен набор функций, обрабатывающих структуры объектов ядра по строго определенным правилам. Мы получаем доступ к объектам ядра только через эти функции. Когда Вы вызываете функцию, создающую объект ядра, она возвращает описатель, идентифицирующий созданный объект. Описатель следует рассматривать как «непрозрачное» значение, которое может быть использовано любым потоком Вашего процесса. Этот описатель Вы передаете Windows-функциям, сообщая системе, какой объект ядра Вас интересует. Но об описателях мы поговорим позже (в этой главе).

Для большей надежности операционной системы Microsoft сделала так, чтобы значения описателей зависели от конкретного процесса. Поэтому, если Вы передадите такое значение (с помощью какого-либо механизма межпроцессной связи) потоку другого процесса, любой вызов из того процесса со значением описателя, полученного в Вашем процессе, даст ошибку. Но не волнуйтесь, в конце главы мы рассмотрим три механизма корректного использования несколькими процессами одного объекта ядра.

Учет пользователей объектов ядра

Объекты ядра принадлежат ядру, а не процессу. Иначе говоря, если Ваш процесс вызывает функцию, создающую объект ядра, а затем завершается, объект ядра может быть не разрушен. В большинстве случаев такой объект все же разрушается; но если созданный Вами объект ядра используется другим процессом, ядро запретит разрушение объекта до тех пор, пока от него не откажется и тот процесс.

Ядру известно, сколько процессов использует конкретный объект ядра, поскольку в каждом объекте есть счетчик числа его пользователей. Этот счетчик — один из элементов данных, общих для всех типов объектов ядра. В момент создания объекта счетчику присваивается 1. Когда к существующему объекту ядра обращается другой процесс, счетчик увеличивается на 1. А когда какой-то процесс завершается, счетчики всех используемых им объектов ядра автоматически уменьшаются на 1. Как только счетчик какого-либо объекта обнуляется, ядро уничтожает этот объект.

Защита

Объекты ядра можно защитить дескриптором защиты (security descriptor), который описывает, кто создал объект и кто имеет права на доступ к нему. Дескрипторы защиты обычно используют при написании серверных приложений; создавая клиентское приложение, Вы можете игнорировать это свойство объектов ядра.

WINDOWS 98 В Windows 98 дескрипторы защиты отсутствуют, так как она не предназначена для выполнения серверных приложений. Тем не менее Вы должны знать о тонкостях, связанных с защитой, и реализовать соответствующие механизмы, чтобы Ваше приложение корректно работало и в Windows 2000.

Почти все функции, создающие объекты ядра, принимают указатель на структуру **SECURITY_ATTRIBUTES** как аргумент, например:

```
HANDLE CreateFileMapping(
    HANDLE hFile,
    PSECURITY_ATTRIBUTES psa,
    DWORD f1Protect,
    DWORD dwMaximumSizeHigh,
    DWORD dwMaximumSizeLow,
    PCTSTR pszName);
```

Большинство приложений вместо этого аргумента передает NULL и создает объект с защитой по умолчанию. Такая защита подразумевает, что создатель объекта и любой член группы администраторов получают к нему полный доступ, а все прочие к объекту не допускаются. Однако Вы можете создать и инициализировать структуру SECURITY_ATTRIBUTES, а затем передать ее адрес. Она выглядит так:

```
typedef struct _SECURITY_ATTRIBUTES {  
    DWORD nLength;  
    LPVOID lpSecurityDescriptor;  
    BOOL bInheritHandle;  
} SECURITY_ATTRIBUTES;
```

Хотя структура называется SECURITY_ATTRIBUTES, лишь один ее элемент имеет отношение к защите — *lpSecurityDescriptor*. Если надо ограничить доступ к созданному Вами объекту ядра, создайте дескриптор защиты и инициализируйте структуру SECURITY_ATTRIBUTES следующим образом:

```
SECURITY_ATTRIBUTES sa;  
sa.nLength = sizeof(sa);           // используется для выяснения версий  
sa.lpSecurityDescriptor = pSD;     // адрес инициализированной SD  
sa.bInheritHandle = FALSE;         // об этом позже  
HANDLE hFileMapping = CreateFileMapping(INVALID_HANDLE_VALUE, &sa,  
    PAGE_READWRITE, 0, 1024, "MyFileMapping");  
:
```

Рассмотрение элемента *bInheritHandle* я отложу до раздела о наследовании, так как этот элемент не имеет ничего общего с защитой.

Желая получить доступ к существующему объекту ядра (вместо того чтобы создавать новый), укажите, какие операции Вы намерены проводить над объектом. Например, если бы я захотел считывать данные из существующей проекции файла, то вызвал бы функцию *OpenFileMapping* таким образом:

```
HANDLE hFileMapping = OpenFileMapping(FILE_MAP_READ, FALSE, "MyFileMapping");
```

Передавая FILE_MAP_READ первым параметром в функцию *OpenFileMapping*, я сообщаю, что, как только мне предоставят доступ к проекции файла, я буду считывать из нее данные. Функция *OpenFileMapping*, прежде чем вернуть действительный описатель, проверяет тип защиты объекта. Если меня, как зарегистрировавшегося пользователя, допускают к существующему объекту ядра «проекция файла», *OpenFileMapping* возвращает действительный описатель. Но если мне отказывают в доступе, *OpenFileMapping* возвращает NULL, а вызов *GetLastError* дает код ошибки 5 (или ERROR_ACCESS_DENIED). Но опять же, в основной массе приложений защиту не используют, и поэтому я больше не буду задерживаться на этой теме.

WINDOWS 98 Хотя в большинстве приложений нет нужды беспокоиться о защите, многие функции Windows требуют, чтобы Вы передавали им информацию о нужном уровне защиты. Некоторые приложения, написанные для Windows 98, в Windows 2000 толком не работают из-за того, что при их реализации не было уделено должного внимания защите.

Представьте, что при запуске приложение считывает данные из какого-то раздела реестра. Чтобы делать это корректно, оно должно вызывать функцию *RegOpenKeyEx*, передавая значение KEY_QUERY_VALUE, которое разрешает операцию чтения в указанном разделе.

Однако многие приложения для Windows 98 создавались без учета специфики Windows 2000. Поскольку Windows 98 не защищает свой реестр, разработчики часто вызывали *RegOpenKeyEx* со значением KEY_ALL_ACCESS. Так проще и не надо ломать голову над тем, какой уровень доступа требуется на самом деле. Но проблема в том, что раздел реестра может быть доступен для чтения и блокирован для записи. В Windows 2000 вызов *RegOpenKeyEx* со значением KEY_ALL_ACCESS заканчивается неудачно, и без соответствующего контроля ошибок приложение может повести себя совершенно непредсказуемо.

Если бы разработчик хоть немного подумал о защите и поменял значение KEY_ALL_ACCESS на KEY_QUERY_VALUE (только-то и всего!), его продукт мог бы работать в обеих операционных системах.

Пренебрежение флагами, определяющими уровень доступа, — одна из самых крупных ошибок, совершаемых разработчиками. Правильное их использование позволило бы легко перенести многие приложения Windows 98 в Windows 2000.

Кроме объектов ядра Ваша программа может использовать объекты других типов — меню, окна, курсоры мыши, кисти и шрифты. Они относятся к объектам User или GDI. Новичок в программировании для Windows может запутаться, пытаясь отличить объекты User или GDI от объектов ядра. Как узнать, например, чьим объектом — User или ядра — является данный значок? Выяснить, не принадлежит ли объект ядру, проще всего так: проанализировать функцию, создающую объект. Практически у всех функций, создающих объекты ядра, есть параметр, позволяющий указать атрибуты защиты, — как у *CreateFileMapping*.

В то же время у функций, создающих объекты User или GDI, нет параметра типа PSECURITY_ATTRIBUTES, и пример тому — функция *CreateIcon*:

```
HICON CreateIcon(
    HINSTANCE hinst,
    int nWidth,
    int nHeight,
    BYTE cPlanes,
    BYTE cBitsPixel,
    CONST BYTE *pbANDbits,
    CONST BYTE *pbXORbits);
```

Таблица описателей объектов ядра

При инициализации процесса система создает в нем таблицу описателей, используемую только для объектов ядра. Сведения о структуре этой таблицы и управлении ею незадокументированы. Вообще-то я воздерживаюсь от рассмотрения недокументированных частей операционных систем. Но в данном случае стоит сделать исключение — квалифицированный Windows-программист, на мой взгляд, должен понимать, как устроена таблица описателей в процессе. Поскольку информация о таблице описателей незадокументирована, я не ручаюсь за ее стопроцентную достоверность и к тому же эта таблица по-разному реализуется в Windows 2000, Windows 98 и Windows CE. Таким образом, следующие разделы помогут понять, что представляет собой таблица описателей, но вот что система действительно делает с ней — этот вопрос я оставляю открытым.

В таблице 3-1 показано, как выглядит таблица описателей, принадлежащая процессу. Как видите, это просто массив структур данных. Каждая структура содержит указатель на какой-нибудь объект ядра, маску доступа и некоторые флаги.

Индекс	Указатель на блок памяти объекта ядра	Маска доступа (DWORD с набором битовых флагов)	Флаги (DWORD с набором битовых флагов)
1	0x????????	0x????????	0x????????
2	0x????????	0x????????	0x????????
...

Таблица 3-1. Структура таблицы описателей, принадлежащей процессу

Создание объекта ядра

Когда процесс инициализируется в первый раз, таблица описателей еще пуста. Но стоит одному из его потоков вызвать функцию, создающую объект ядра (например, *CreateFileMapping*), как ядро выделяет для этого объекта блок памяти и инициализирует его; далее ядро просматривает таблицу описателей, принадлежащую данному процессу, и отыскивает свободную запись. Поскольку таблица еще пуста, ядро обнаруживает структуру с индексом 1 и инициализирует ее. Указатель устанавливается на внутренний адрес структуры данных объекта, маска доступа — на доступ без ограничений и, наконец, определяется последний компонент — флаги. (О флагах мы поговорим позже, в разделе о наследовании.)

Вот некоторые функции, создающие объекты ядра (список ни в коей мере на полноту не претендует):

```
HANDLE CreateThread(
    PSECURITY_ATTRIBUTES psa,
    DWORD dwStackSize,
    PTHREAD_START_ROUTINE pfnStartAddr,
    PVOID pvParam,
    DWORD dwCreationFlags,
    PDWORD pdwThreadId);

HANDLE CreateFile(
    PCTSTR pszFileName,
    DWORD dwDesiredAccess,
    DWORD dwShareMode,
    PSECURITY_ATTRIBUTES psa,
    DWORD dwCreationDistribution,
    DWORD dwFlagsAndAttributes,
    HANDLE hTemplateFile);

HANDLE CreateFileMapping(
    HANDLE hFile,
    PSECURITY_ATTRIBUTES psa,
    DWORD flProtect,
    DWORD dwMaximumSizeHigh,
    DWORD dwMaximumSizeLow,
    PCTSTR pszName);

HANDLE CreateSemaphore(
    PSECURITY_ATTRIBUTES psa,
```

```
LONG lInitialCount,
LONG lMaximumCount,
PCTSTR pszName);
```

Все функции, создающие объекты ядра, возвращают описатели, которые привязаны к конкретному процессу и могут быть использованы в любом потоке данного процесса. Значение описателя представляет собой индекс в таблице описателей, принадлежащей процессу, и таким образом идентифицирует место, где хранится информация, связанная с объектом ядра. Вот поэтому при отладке своего приложения и просмотре фактического значения описателя объекта ядра Вы и видите такие малые величины: 1, 2 и т. д. Но помните, что физическое содержимое описателей не задокументировано и может быть изменено. Кстати, в Windows 2000 это значение определяет, по сути, не индекс, а скорее байтовое смещение нужной записи от начала таблицы описателей.

Всякий раз, когда Вы вызываете функцию, принимающую описатель объекта ядра как аргумент, Вы передаете ей значение, возвращенное одной из *Create*-функций. При этом функция смотрит в таблицу описателей, принадлежащую Вашему процессу, и считывает адрес нужного объекта ядра.

Если Вы передаете неверный индекс (описатель), функция завершается с ошибкой и *GetLastError* возвращает 6 (ERROR_INVALID_HANDLE). Это связано с тем, что на самом деле описатели представляют собой индексы в таблице, их значения привязаны к конкретному процессу и недействительны в других процессах.

Если вызов функции, создающей объект ядра, оказывается неудачен, то обычно возвращается 0 (NULL). Такая ситуация возможна только при острой нехватке памяти или при наличии проблем с защитой. К сожалению, отдельные функции возвращают в таких случаях не 0, а -1 (INVALID_HANDLE_VALUE). Например, если *CreateFile* не сможет открыть указанный файл, она вернет именно INVALID_HANDLE_VALUE. Будьте очень осторожны при проверке значения, возвращаемого функцией, которая создает объект ядра. Так, для *CreateMutex* проверка на INVALID_HANDLE_VALUE бессмысленна:

```
HANDLE hMutex = CreateMutex(...);
if (hMutex == INVALID_HANDLE_VALUE) {
    // этот код никогда не будет выполнен, так как
    // при ошибке CreateMutex возвращает NULL
}
```

Точно так же бессмыслен и следующий код:

```
HANDLE hFile = CreateFile(...);
if (hFile == NULL) {
    // и этот код никогда не будет выполнен, так как
    // при ошибке CreateFile возвращает INVALID_HANDLE_VALUE (-1)
}
```

Закрытие объекта ядра

Независимо от того, как именно Вы создали объект ядра, по окончании работы с ним его нужно закрыть вызовом *CloseHandle*:

```
BOOL CloseHandle(HANDLE hobj);
```

Эта функция сначала проверяет таблицу описателей, принадлежащую вызывающему процессу, чтобы убедиться, идентифицирует ли переданный ей индекс (описа-

тель) объект, к которому этот процесс действительно имеет доступ. Если переданный индекс правилен, система получает адрес структуры данных объекта и уменьшает в этой структуре счетчик числа пользователей; как только счетчик обнулится, ядро удалит объект из памяти.

Если же описатель неверен, происходит одно из двух. В нормальном режиме выполнения процесса *CloseHandle* возвращает FALSE, а *GetLastError* — код ERROR_INVALID_HANDLE. Но при выполнении процесса в режиме отладки система просто уведомляет отладчик об ошибке.

Перед самым возвратом управления *CloseHandle* удаляет соответствующую запись из таблицы описателей: данный описатель теперь недействителен в Вашем процессе и использовать его нельзя. При этом запись удаляется независимо от того, разрушен объект ядра или нет! После вызова *CloseHandle* Вы больше не получите доступ к этому объекту ядра; но, если его счетчик не обнулен, объект остается в памяти. Тут все нормально, это означает лишь то, что объект используется другим процессом (или процессами). Когда и остальные процессы завершат свою работу с этим объектом (тоже вызвав *CloseHandle*), он будет разрушен.

А вдруг Вы забыли вызвать *CloseHandle* — будет ли утечка памяти? И да, и нет. Утечка ресурсов (тех же объектов ядра) вполне вероятна, пока процесс еще исполняется. Однако по завершении процесса операционная система гарантированно освобождает все ресурсы, принадлежавшие этому процессу, и в случае объектов ядра действует так: в момент завершения процесса просматривает его таблицу описателей и закрывает любые открытые описатели.

Совместное использование объектов ядра несколькими процессами

Время от времени возникает необходимость в разделении объектов ядра между потоками, исполняемыми в разных процессах. Причин тому может быть несколько:

- объекты «проекции файлов» позволяют двум процессам, исполняемым на одной машине, совместно использовать одни и те же блоки данных;
- почтовые ящики и именованные каналы дают возможность программам обмениваться данными с процессами, исполняемыми на других машинах в сети;
- мьютексы, семафоры и события позволяют синхронизировать потоки, исполняемые в разных процессах, чтобы одно приложение могло уведомить другое об окончании той или иной операции.

Но поскольку описатели объектов ядра имеют смысл только в конкретном процессе, разделение объектов ядра между несколькими процессами — задача весьма непростая. У Microsoft было несколько веских причин сделать описатели «процессно-независимыми», и самая главная — устойчивость операционной системы к сбоям. Если бы описатели объектов ядра были общесистемными, то один процесс мог бы запросто получить описатель объекта, используемого другим процессом, и устроить в нем (этом процессе) настоящий хаос. Другая причина — защита. Объекты ядра защищены, и процесс, прежде чем оперировать с ними, должен запрашивать разрешение на доступ к ним.

Три механизма, позволяющие процессам совместно использовать одни и те же объекты ядра, мы рассмотрим в следующем разделе.

Наследование описателя объекта

Наследование применимо, только когда процессы связаны «родственными» отношениями (родительский-дочерний). Например, родительскому процессу доступен один или несколько описателей объектов ядра, и он решает, породив дочерний процесс, передать ему по наследству доступ к своим объектам ядра. Чтобы такой сценарий наследования сработал, родительский процесс должен выполнить несколько операций.

Во-первых, еще при создании объекта ядра этот процесс должен сообщить системе, что ему нужен наследуемый описатель данного объекта. (Имейте в виду: описатели объектов ядра наследуются, но сами объекты ядра — нет.)

Чтобы создать наследуемый описатель, родительский процесс выделяет и инициализирует структуру SECURITY_ATTRIBUTES, а затем передает ее адрес требуемой *Create*-функции. Следующий код создает объект-мьютекс и возвращает его описатель:

```
SECURITY_ATTRIBUTES sa;
sa.nLength = sizeof(sa);
sa.lpSecurityDescriptor = NULL;
sa.bInheritHandle = TRUE; // делаем возвращаемый описатель наследуемым

HANDLE hMutex = CreateMutex(&sa, FALSE, NULL);

:
```

Этот код инициализирует структуру SECURITY_ATTRIBUTES, указывая, что объект следует создать с защитой по умолчанию (в Windows 98 это игнорируется) и что возвращаемый описатель должен быть наследуемым.

WINDOWS 98 Хотя Windows 98 не полностью поддерживает защиту, она все же поддерживает наследование и поэтому корректно обрабатывает элемент *bInheritHandle*.

А теперь перейдем к флагам, которые хранятся в таблице описателей, принадлежащей процессу. В каждой ее записи присутствует битовый флаг, сообщающий, является данный описатель наследуемым или нет. Если Вы, создавая объект ядра, передадите в параметре типа PSECURITY_ATTRIBUTES значение NULL, то получите ненаследуемый описатель, и этот флаг будет нулевым. А если элемент *bInheritHandle* равен TRUE, флагу присваивается 1.

Допустим, какому-то процессу принадлежит таблица описателей, как в таблице 3-2.

Индекс	Указатель на блок памяти объекта ядра	Маска доступа (DWORD с набором битовых флагов)	Флаги (DWORD с набором битовых флагов)
1	0xF0000000	0x????????	0x00000000
2	0x00000000	(неприменим)	(неприменим)
3	0xF0000010	0x????????	0x00000001

Таблица 3-2. Таблица описателей с двумя действительными записями

Эта таблица свидетельствует, что данный процесс имеет доступ к двум объектам ядра: описатель 1 (ненаследуемый) и 3 (наследуемый).

Следующий этап — родительский процесс порождает дочерний. Это делается с помощью функции *CreateProcess*.

```
BOOL CreateProcess(
    PCTSTR pszApplicationName,
    PTSTR pszCommandLine,
    PSECURITY_ATTRIBUTES psaProcess,
    PSECURITY_ATTRIBUTES psaThread,
    BOOL bInheritHandles,
    DWORD fdwCreate,
    PVOID pvEnvironment,
    PCTSTR pszCurDir,
    PSTARTUPINFO psiStartInfo,
    PPROCESS_INFORMATION ppiProcInfo);
```

Подробно мы рассмотрим эту функцию в следующей главе, а сейчас я хочу лишь обратить Ваше внимание на параметр *bInheritHandles*. Создавая процесс, Вы обычно передаете в этом параметре FALSE, тем самым сообщая системе, что дочерний процесс не должен наследовать наследуемые описатели, зафиксированные в таблице родительского процесса. Если же Вы передаете TRUE, дочерний процесс наследует описатели родительского. Тогда операционная система создает дочерний процесс, но не дает ему немедленно начать свою работу. Сформировав в нем, как обычно, новую (пустую) таблицу описателей, она считывает таблицу родительского процесса и копирует все ее действительные записи в таблицу дочернего — причем в те же позиции. Последний факт чрезвычайно важен, так как означает, что описатели будут идентичны в обоих процессах (родительском и дочернем).

Помимо копирования записей из таблицы описателей, система увеличивает значения счетчиков соответствующих объектов ядра, поскольку эти объекты теперь используются обоими процессами. Чтобы уничтожить какой-то объект ядра, его описатель должны закрыть (вызовом *CloseHandle*) оба процесса. Кстати, сразу после возврата управления функцией *CreateProcess* родительский процесс может закрыть свой описатель объекта, и это никак не отразится на способности дочернего процесса манипулировать с этим объектом.

В таблице 3-3 показано состояние таблицы описателей в дочернем процессе — перед самым началом его исполнения. Как видите, записи 1 и 2 не инициализированы, и поэтому данные описатели неприменимы в дочернем процессе. Однако индекс 3 действительно идентифицирует объект ядра по тому же (что и в родительском) адресу 0xF0000010. При этом маска доступа и флаги в родительском и дочернем процессах тоже идентичны. Так что, если дочерний процесс в свою очередь породит новый («внука» по отношению к исходному родительскому), «внук» унаследует данный описатель объекта ядра с теми же значениями, правами доступа и флагами, а счетчик числа пользователей этого объекта ядра вновь увеличится на 1.

Индекс	Указатель на блок памяти объекта ядра	Маска доступа (DWORD с набором битовых флагов)	Флаги (DWORD с набором битовых флагов)
1	0x00000000	(неприменим)	(неприменим)
2	0x00000000	(неприменим)	(неприменим)
3	0xF0000010	0x????????	0x00000001

Таблица 3-3. Таблица описателей в дочернем процессе (после того как он унаследовал от родительского один наследуемый описатель)

Наследуются только описатели объектов, существующие на момент создания дочернего процесса. Если родительский процесс создаст после этого новые объекты

ядра с наследуемыми описателями, то эти описатели будут уже недоступны дочернему процессу.

Для наследования описателей объектов характерно одно очень странное свойство: дочерний процесс не имеет ни малейшего понятия, что он унаследовал какие-то описатели. Поэтому наследование описателей объектов ядра полезно, только когда дочерний процесс сообщает, что при его создании родительским процессом он ожидает доступа к какому-нибудь объекту ядра. Тут надо заметить, что обычно родительское и дочернее приложения пишутся одной фирмой, но в принципе дочернее приложение может написать и стороннюю фирму, если в этой программе задокументировано, чего именно она ждет от родительского процесса.

Для этого в дочерний процесс обычно передают значение ожидаемого им описателя объекта ядра как аргумент в командной строке. Инициализирующий код дочернего процесса анализирует командную строку (чаще всего вызовом *sscanf*), извлекает из нее значение описателя, и дочерний процесс получает неограниченный доступ к объекту. При этом механизм наследования срабатывает только потому, что значение описателя общего объекта ядра в родительском и дочернем процессах одинаково, — и именно по этой причине родительский процесс может передать значение описателя как аргумент в командной строке.

Для наследственной передачи описателя объекта ядра от родительского процесса дочернему, конечно же, годятся и другие формы межпроцессной связи. Один из приемов заключается в том, что родительский процесс дожидается окончания инициализации дочернего (через функцию *WaitForInputIdle*, рассматриваемую в главе 9), а затем посыпает (синхронно или асинхронно) сообщение окну, созданному потоком дочернего процесса.

Еще один прием: родительский процесс добавляет в свой блок переменных окружения новую переменную. Она должна быть «узнаваема» дочерним процессом и содержать значение наследуемого описателя объекта ядра. Далее родительский процесс создает дочерний, тот наследует переменные окружения родительского процесса и, вызвав *GetEnvironmentVariable*, получает нужный описатель. Такой прием особенно хорош, когда дочерний процесс тоже порождает процессы, — ведь все переменные окружения вновь наследуются.

Изменение флагов описателя

Иногда встречаются ситуации, в которых родительский процесс создает объект ядра с наследуемым описателем, а затем порождает два дочерних процесса. Но наследуемый описатель нужен только одному из них. Иначе говоря, время от времени возникает необходимость контролировать, какой из дочерних процессов наследует описатели объектов ядра. Для этого модифицируйте флаг наследования, связанный с описателем, вызовом *SetHandleInformation*:

```
BOOL SetHandleInformation(
    HANDLE hObject,
    DWORD dwMask,
    DWORD dwFlags);
```

Как видите, эта функция принимает три параметра. Первый (*hObject*) идентифицирует допустимый описатель. Второй (*dwMask*) сообщает функции, какой флаг (или флаги) Вы хотите изменить. На сегодняшний день с каждым описателем связано два флага:

```
#define HANDLE_FLAG_INHERIT      0x00000001
#define HANDLE_FLAG_PROTECT_FROM_CLOSE 0x00000002
```

Чтобы изменить сразу все флаги объекта, нужно объединить их побитовой операцией OR.

И, наконец, третий параметр функции *SetHandleInformation* — *dwFlags* — указывает, в какое именно состояние следует перевести флаги. Например, чтобы установить флаг наследования для описателя объекта ядра:

```
SetHandleInformation(hobj, HANDLE_FLAG_INHERIT, HANDLE_FLAG_INHERIT);
```

а чтобы сбросить этот флаг:

```
SetHandleInformation(hobj, HANDLE_FLAG_INHERIT, 0);
```

Флаг HANDLE_FLAG_PROTECT_FROM_CLOSE сообщает системе, что данный описатель закрывать нельзя:

```
SetHandleInformation(hobj, HANDLE_FLAG_PROTECT_FROM_CLOSE,
    HANDLE_FLAG_PROTECT_FROM_CLOSE);
CloseHandle(hobj); // генерируется исключение
```

Если какой-нибудь поток попытается закрыть защищенный описатель, *CloseHandle* приведет к исключению. Необходимость в такой защите возникает очень редко. Однако этот флаг весьма полезен, когда процесс порождает дочерний, а тот в свою очередь — еще один процесс. При этом родительский процесс может ожидать, что его «внук» унаследует определенный описатель объекта, переданный дочернему. Но тут вполне возможно, что дочерний процесс, прежде чем породить новый процесс, закрывает нужный описатель. Тогда родительский процесс теряет связь с «внуком», поскольку тот не унаследовал требуемый объект ядра. Защитив описатель от закрытия, Вы исправите ситуацию, и «внук» унаследует предназначенный ему объект.

У этого подхода, впрочем, есть один недостаток. Дочерний процесс, вызвав:

```
SetHandleInformation(hobj, HANDLE_FLAG_PROTECT_FROM_CLOSE, 0);
CloseHandle(hobj);
```

может сбросить флаг HANDLE_FLAG_PROTECT_FROM_CLOSE и закрыть затем соответствующий описатель. Родительский процесс ставит на то, что дочерний не исполнит этот код. Но одновременно он ставит и на то, что дочерний процесс породит ему «внука», поэтому в целом ставки не слишком рискованны.

Для полноты картины стоит, пожалуй, упомянуть и функцию *GetHandleInformation*:

```
BOOL GetHandleInformation(
    HANDLE hObj,
    PDWORD pdwFlags);
```

Эта функция возвращает текущие флаги для заданного описателя в переменной типа *DWORD*, на которую указывает *pdwFlags*. Чтобы проверить, является ли описатель наследуемым, сделайте так:

```
DWORD dwFlags;
GetHandleInformation(hObj, &dwFlags);
BOOL fHandleIsInheritable = (0 != (dwFlags & HANDLE_FLAG_INHERIT));
```

Именованные объекты

Второй способ, позволяющий нескольким процессам совместно использовать одни и те же объекты ядра, связан с именованием этих объектов. Именование допускают многие (но не все) объекты ядра. Например, следующие функции создают именованные объекты ядра:

```

HANDLE CreateMutex(
    PSECURITY_ATTRIBUTES psa,
    BOOL bInitialOwner,
    PCTSTR pszName);

HANDLE CreateEvent(
    PSECURITY_ATTRIBUTES psa,
    BOOL bManualReset,
    BOOL bInitialState,
    PCTSTR pszName);

HANDLE CreateSemaphore(
    PSECURITY_ATTRIBUTES psa,
    LONG lInitialCount,
    LONG lMaximumCount,
    PCTSTR pszName);

HANDLE CreateWaitableTimer(
    PSECURITY_ATTRIBUTES psa,
    BOOL bManualReset,
    PCTSTR pszName);

HANDLE CreateFileMapping(
    HANDLE hFile,
    PSECURITY_ATTRIBUTES psa,
    DWORD f1Protect,
    DWORD dwMaximumSizeHigh,
    DWORD dwMaximumSizeLow,
    PCTSTR pszName);

HANDLE CreateJobObject(
    PSECURITY_ATTRIBUTES psa,
    PCTSTR pszName);

```

Последний параметр, *pszName*, у всех этих функций одинаков. Передавая в нем NULL, Вы создаете безымянный (анонимный) объект ядра. В этом случае Вы можете разделять объект между процессами либо через наследование (см. предыдущий раздел), либо с помощью *DuplicateHandle* (см. следующий раздел). А чтобы разделять объект по имени, Вы должны присвоить ему какое-нибудь имя. Тогда вместо NULL в параметре *pszName* нужно передать адрес строки с именем, завершающей нулевым символом. Имя может быть длиной до MAX_PATH знаков (это значение определено как 260). К сожалению, Microsoft ничего не сообщает о правилах именования объектов ядра. Например, создавая объект с именем JeffObj, Вы никак не застрахованы от того, что в системе еще нет объекта ядра с таким именем. И что хуже, все эти объекты делят единное пространство имен. Из-за этого следующий вызов *CreateSemaphore* будет всегда возвращать NULL:

```

HANDLE hMutex = CreateMutex(NULL, FALSE, "JeffObj");
HANDLE hSem = CreateSemaphore(NULL, 1, 1, "JeffObj");
DWORD dwErrorCode = GetLastError();

```

После выполнения этого фрагмента значение *dwErrorCode* будет равно 6 (ERROR_INVALID_HANDLE). Полученный код ошибки не слишком вразумителен, но другого не дано.

Теперь, когда Вы научились именовать объекты, рассмотрим, как разделять их между процессами по именам. Допустим, после запуска процесса A вызывается функция:

```
HANDLE hMutexProcessA = CreateMutex(NULL, FALSE, "JeffMutex");
```

Этот вызов заставляет систему создать новенький, как с иголочки, объект ядра «мьютекс» и присвоить ему имя JeffMutex. Заметьте, что описатель *hMutexProcessA* в процессе A не является наследуемым, — он и не должен быть таковым при простом именовании объектов.

Спустя какое-то время некий процесс порождает процесс B. Необязательно, чтобы последний был дочерним от процесса A; он может быть порожден Explorer или любым другим приложением. (В этом, кстати, и состоит преимущество механизма именования объектов перед наследованием.) Когда процесс B приступает к работе, исполняется код:

```
HANDLE hMutexProcessB = CreateMutex(NULL, FALSE, "JeffMutex");
```

При этом вызове система сначала проверяет, не существует ли уже объект ядра с таким именем. Если да, то ядро проверяет тип этого объекта. Поскольку мы пытаемся создать мьютекс и его имя тоже JeffMutex, система проверяет права доступа вызывающего процесса к этому объекту. Если у него есть все права, в таблице описателей, принадлежащей процессу B, создается новая запись, указывающая на существующий объект ядра. Если же вызывающий процесс не имеет полных прав на доступ к объекту или если типы двух объектов с одинаковыми именами не совпадают, вызов *CreateMutex* заканчивается неудачно и возвращается NULL.

Однако, хотя процесс B успешно вызвал *CreateMutex*, новый объект-мьютекс он не создал. Вместо этого он получил свой описатель существующего объекта-мьютекса. Счетчик объекта, конечно же, увеличился на 1, и теперь этот объект не разрушится, пока его описатели не закроют оба процесса — A и B. Заметьте, что значения описателей объекта в обоих процессах скорее всего разные, но так и должно быть: каждый процесс будет оперировать с данным объектом ядра, используя свой описатель.



Разделяя объекты ядра по именам, помните об одной крайне важной вещи. Вызывая *CreateMutex*, процесс B передает ей атрибуты защиты и второй параметр. Так вот, эти параметры игнорируются, если объект с указанным именем уже существует! Приложение может определить, что оно делает: создает новый объект ядра или просто открывает уже существующий, — вызвав *GetLastError* сразу же после вызова одной из *Create*-функций:

```
HANDLE hMutex = CreateMutex(&sa, FALSE, "JeffObj");
if (GetLastError() == ERROR_ALREADY_EXISTS) {
    // открыт описатель существующего объекта;
    // sa.lpSecurityDescriptor и второй параметр
    // (FALSE) игнорируются
} else {
    // создан совершенно новый объект;
    // sa.lpSecurityDescriptor и второй параметр
    // (FALSE) используются при создании объекта
}
```

Есть и другой способ разделения объектов по именам. Вместо вызова *Create*-функции процесс может обратиться к одной из следующих *Open*-функций:

```

HANDLE OpenMutex(
    DWORD dwDesiredAccess,
    BOOL bInheritHandle,
    PCTSTR pszName);

HANDLE OpenEvent(
    DWORD dwDesiredAccess,
    BOOL bInheritHandle,
    PCTSTR pszName);

HANDLE OpenSemaphore(
    DWORD dwDesiredAccess,
    BOOL bInheritHandle,
    PCTSTR pszName);

HANDLE OpenWaitableTimer(
    DWORD dwDesiredAccess,
    BOOL bInheritHandle,
    PCTSTR pszName);

HANDLE OpenFileMapping(
    DWORD dwDesiredAccess,
    BOOL bInheritHandle,
    PCTSTR pszName);

HANDLE OpenJobObject(
    DWORD dwDesiredAccess,
    BOOL bInheritHandle,
    PCTSTR pszName);

```

Заметьте: все эти функции имеют один прототип. Последний параметр, *pszName*, определяет имя объекта ядра. В нем нельзя передать NULL — только адрес строки с нулевым символом в конце. Эти функции просматривают единое пространство имен объектов ядра, пытаясь найти совпадение. Если объекта ядра с указанным именем нет, функции возвращают NULL, а *GetLastError* — код 2 (ERROR_FILE_NOT_FOUND). Но если объект ядра с заданным именем существует и если его тип идентичен тому, что Вы указали, система проверяет, разрешен ли к данному объекту доступ запрошенного вида (через параметр *dwDesiredAccess*). Если такой вид доступа разрешен, таблица описателей в вызывающем процессе обновляется, и счетчик числа пользователей объекта возрастает на 1. Если Вы присвоили параметру *bInheritHandle* значение TRUE, то получите наследуемый описатель.

Главное отличие между вызовом *Create*- и *Open*-функций в том, что при отсутствии указанного объекта *Create*-функция создает его, а *Open*-функция просто уведомляет об ошибке.

Как я уже говорил, Microsoft ничего не сообщает о правилах именования объектов ядра. Но представьте себе, что пользователь запускает две программы от разных компаний и каждая программа пытается создать объект с именем «MyObject». Ничего хорошего из этого не выйдет. Чтобы избежать такой ситуации, я бы посоветовал со-здавать GUID и использовать его строковое представление как имя объекта.

Именованные объекты часто применяются для того, чтобы не допустить запуска нескольких экземпляров одного приложения. Для этого Вы просто вызываете одну из *Create*-функций в своей функции *main* или *WinMain* и создаете некий именованный

объект. Какой именно — не имеет ни малейшего значения. Сразу после *Create*-функции Вы должны вызвать *GetLastError*. Если она вернет ERROR_ALREADY_EXISTS, значит, один экземпляр Вашего приложения уже выполняется и новый его экземпляр можно закрыть. Вот фрагмент кода, иллюстрирующий этот прием:

```
int WINAPI WinMain(HINSTANCE hinstExe, HINSTANCE, PSTR pszCmdLine, int nCmdShow) {  
    HANDLE h = CreateMutex(NULL, FALSE,  
        "{FA531CC1-0497-11d3-A180-00105A276C3E}");  
    if (GetLastError() == ERROR_ALREADY_EXISTS) {  
        // экземпляр этого приложения уже выполняется  
        return(0);  
    }  
    // запущен первый экземпляр данного приложения  
  
    :  
  
    // перед выходом закрываем объект  
    CloseHandle(h);  
    return(0);  
}
```

Пространства имен Terminal Server

Terminal Server несколько меняет описанный выше сценарий. На машине с Terminal Server существует множество пространств имен для объектов ядра. Объекты, которые должны быть доступны всем клиентам, используют одно глобальное пространство имен. (Такие объекты, как правило, связаны с сервисами, предоставляемыми клиентским программам.) В каждом клиентском сеансе формируется свое пространство имен, чтобы исключить конфликты между несколькими сессиями, в которых запускается одно и то же приложение. Ни из какого сеанса нельзя получить доступ к объектам другого сеанса, даже если у их объектов идентичные имена.

Именованные объекты ядра, относящиеся к какому-либо сервису, всегда находятся в глобальном пространстве имен, а аналогичный объект, связанный с приложением, Terminal Server по умолчанию помещает в пространство имен клиентского сеанса. Однако и его можно перевести в глобальное пространство имен, поставив перед именем объекта префикс «Global\\», как в примере ниже.

```
HANDLE h = CreateEvent(NULL, FALSE, FALSE, "Global\\MyName");
```

Если Вы хотите явно указать, что объект ядра должен находиться в пространстве имен клиентского сеанса, используйте префикс «Local\\»:

```
HANDLE h = CreateEvent(NULL, FALSE, FALSE, "Local\\MyName");
```

Microsoft рассматривает префиксы Global и Local как зарезервированные ключевые слова, которые не должны встречаться в самих именах объектов. К числу таких слов Microsoft относит и Session, хотя на сегодняшний день оно не связано ни с какой функциональностью. Также обратите внимание на две вещи: все эти ключевые слова чувствительны к регистру букв и игнорируются, если компьютер работает без Terminal Server.

Дублирование описателей объектов

Последний механизм совместного использования объектов ядра несколькими процессами требует функции *DuplicateHandle*:

```
BOOL DuplicateHandle(
    HANDLE hSourceProcessHandle,
    HANDLE hSourceHandle,
    HANDLE hTargetProcessHandle,
    PHANDLE phTargetHandle,
    DWORD dwDesiredAccess,
    BOOL bInheritHandle,
    DWORD dwOptions);
```

Говоря по-простому, эта функция берет запись в таблице описателей одного процесса и создает ее копию в таблице другого. *DuplicateHandle* принимает несколько параметров, но на самом деле весьма незамысловата. Обычно ее применение требует наличия в системе трех разных процессов.

Первый и третий параметры функции *DuplicateHandle* представляют собой описатели объектов ядра, специфичные для вызывающего процесса. Кроме того, эти параметры должны идентифицировать именно процессы — функция завершится с ошибкой, если Вы передадите описатели на объекты ядра любого другого типа. Подробнее объекты ядра «процессы» мы обсудим в главе 4, а сейчас Вам достаточно знать только одно: объект ядра «процесс» создается при каждой инициации в системе нового процесса.

Второй параметр, *hSourceHandle*, — описатель объекта ядра любого типа. Однако его значение специфично не для процесса, вызывающего *DuplicateHandle*, а для того, на который указывает описатель *hSourceProcessHandle*. Параметр *phTargetHandle* — это адрес переменной типа HANDLE, в которой возвращается индекс записи с копией описателя из процесса-источника. Значение возвращаемого описателя специфично для процесса, определяемого параметром *hTargetProcessHandle*.

Предпоследние два параметра *DuplicateHandle* позволяют задать маску доступа и флаг наследования, устанавливаемые для данного описателя в процессе-приемнике. И, наконец, параметр *dwOptions* может быть 0 или любой комбинацией двух флагов: DUPLICATE_SAME_ACCESS и DUPLICATE_CLOSE_SOURCE.

Первый флаг подсказывает *DuplicateHandle*: у описателя, получаемого процессом-приемником, должна быть та же маска доступа, что и у описателя в процессе-источнике. Этот флаг заставляет *DuplicateHandle* игнорировать параметр *dwDesiredAccess*.

Второй флаг приводит к закрытию описателя в процессе-источнике. Он позволяет процессам обмениваться объектом ядра как эстафетной палочкой. При этом счетчик объекта не меняется.

Попробуем проиллюстрировать работу функции *DuplicateHandle* на примере. Здесь S — это процесс-источник, имеющий доступ к какому-то объекту ядра, T — это процесс-приемник, который получит доступ к тому же объекту ядра, а C — процесс-катализатор, вызывающий функцию *DuplicateHandle*.

Таблица описателей в процессе С (см. таблицу 3-4) содержит два индекса — 1 и 2. Описатель с первым значением идентифицирует объект ядра «процесс S», описатель со вторым значением — объект ядра «процесс T».

Индекс	Указатель на блок памяти объекта ядра	Маска доступа (DWORD с набором битовых флагов)	Флаги (DWORD с набором битовых флагов)
1	0xF0000000 (объект ядра процесса S)	0x????????	0x00000000
2	0xF0000010 (объект ядра процесса T)	0x????????	0x00000000

Таблица 3-4. Таблица описателей в процессе С

Таблица 3-5 иллюстрирует таблицу описателей в процессе S, содержащую единственную запись со значением описателя, равным 2. Этот описатель может идентифицировать объект ядра любого типа, а не только «процесс».

Индекс	Указатель на блок памяти объекта ядра	Маска доступа (DWORD с набором битовых флагов)	Флаги (DWORD с набором битовых флагов)
1	0x00000000	(неприменим)	(неприменим)
2	0xF0000020 (объект ядра любого типа)	0x????????	0x00000000

Таблица 3-5. Таблица описателей в процессе S

В таблице 3-6 показано, что именно содержит таблица описателей в процессе T перед вызовом процессом С функции *DuplicateHandle*. Как видите, в ней всего одна запись со значением описателя, равным 2, а запись с индексом 1 пока пуста.

Индекс	Указатель на блок памяти объекта ядра	Маска доступа (DWORD с набором битовых флагов)	Флаги (DWORD с набором битовых флагов)
1	0x00000000	(неприменим)	(неприменим)
2	0xF0000030 (объект ядра любого типа)	0x????????	0x00000000

Таблица 3-6. Таблица описателей в процессе T перед вызовом *DuplicateHandle*

Если процесс С теперь вызовет *DuplicateHandle* так:

```
DuplicateHandle(1, 2, 2, &hObj, 0, TRUE, DUPLICATE_SAME_ACCESS);
```

то после вызова изменится только таблица описателей в процессе T (см. таблицу 3-7).

Индекс	Указатель на блок памяти объекта ядра	Маска доступа (DWORD с набором битовых флагов)	Флаги (DWORD с набором битовых флагов)
1	0xF0000020	0x????????	0x00000001
2	0xF0000030 (объект ядра любого типа)	0x????????	0x00000000

Таблица 3-7. Таблица описателей в процессе T после вызова *DuplicateHandle*

Вторая строка таблицы описателей в процессе S скопирована в первую строку таблицы описателей в процессе T. Функция *DuplicateHandle* присвоила также переменной *hObj* процесса С значение 1 — индекс той строки таблицы в процессе T, в которую занесен новый описатель.

Поскольку функции *DuplicateHandle* передан флаг DUPLICATE_SAME_ACCESS, маска доступа для этого описателя в процессе T идентична маске доступа в процессе S. Кроме того, данный флаг заставляет *DuplicateHandle* проигнорировать параметр *dwDesiredAccess*. Заметьте также, что система установила битовый флаг наследования, так как в параметре *bInheritHandle* функции *DuplicateHandle* мы передали TRUE.

Очевидно, Вы никогда не станете передавать в *DuplicateHandle* жестко зашитые значения, как это сделал я, просто демонстрируя работу функции. В реальных программах значения описателей хранятся в переменных и, конечно же, именно эти переменные передаются функциям.

Как и механизм наследования, функция *DuplicateHandle* тоже обладает одной странностью: процесс-приемник никак не уведомляется о том, что он получил доступ к новому объекту ядра. Поэтому процесс С должен каким-то образом сообщить

процессу Т, что тот имеет теперь доступ к новому объекту; для этого нужно воспользоваться одной из форм межпроцессной связи и передать в процесс Т значение описателя в переменной *hObj*. Ясное дело, в данном случае не годится ни командная строка, ни изменение переменных окружения процесса Т, поскольку этот процесс уже выполняется. Здесь придется послать сообщение окну или задействовать какой-нибудь другой механизм межпроцессной связи.

Я рассказал Вам о функции *DuplicateHandle* в самом общем виде. Надеюсь, Вы увидели, насколько она гибка. Но эта функция редко используется в ситуациях, требующих участия трех разных процессов. Обычно ее вызывают применительно к двум процессам. Представьте, что один процесс имеет доступ к объекту, к которому хочет обратиться другой процесс, или что один процесс хочет предоставить другому доступ к «своему» объекту ядра. Например, если процесс S имеет доступ к объекту ядра и Вам нужно, чтобы к этому объекту мог обращаться процесс Т, используйте *DuplicateHandle* так:

```
// весь приведенный ниже код исполняется процессом S

// создаем объект-мьютекс, доступный процессу S
HANDLE hObjProcessS = CreateMutex(NULL, FALSE, NULL);

// открываем описатель объекта ядра "процесс Т"
HANDLE hProcessT = OpenProcess(PROCESS_ALL_ACCESS, FALSE, dwProcessIdT);

HANDLE hObjProcessT; // неинициализированный описатель,
// связанный с процессом Т

// предоставляем процессу Т доступ к объекту-мьютексу
DuplicateHandle(GetCurrentProcess(), hObjProcessS, hProcessT,
&hObjProcessT, 0, FALSE, DUPLICATE_SAME_ACCESS);

// используем какую-нибудь форму межпроцессной связи, чтобы передать
// значение описателя из hObjProcessS в процесс Т

:

// связь с процессом Т больше не нужна
CloseHandle(hProcessT);

:

// если процессу S не нужен объект-мьютекс, он должен закрыть его
CloseHandle(hObjProcessS);
```

Вызов *GetCurrentProcess* возвращает псевдоописатель, который всегда идентифицирует вызывающий процесс, в данном случае — процесс S. Как только функция *DuplicateHandle* возвращает управление, *hObjProcessT* становится описателем, связанным с процессом Т и идентифицирующим тот же объект, что и описатель *hObjProcessS* (когда на него ссылается код процесса S). При этом процесс S ни в коем случае не должен исполнять следующий код:

```
// процесс S никогда не должен пытаться исполнять код,
// закрывающий продублированный описатель
CloseHandle(hObjProcessT);
```

Если процесс S выполнит этот код, вызов может дать (а может и не дать) ошибку. Он будет успешен, если у процесса S случайно окажется описатель с тем же значением, что и в *hObjProcessT*. При этом процесс S закроет неизвестно какой объект, и что будет потом — остается только гадать.

Теперь о другом способе применения *DuplicateHandle*. Допустим, некий процесс имеет полный доступ (для чтения и записи) к объекту «проекция файла» и из этого процесса вызывается функция, которая должна обратиться к проекции файла и считать из нее какие-то данные. Так вот, если мы хотим повысить отказоустойчивость приложения, то могли бы с помощью *DuplicateHandle* создать новый описатель существующего объекта и разрешить доступ только для чтения. Потом мы передали бы этот описатель функции, и та уже не смогла бы случайно что-то записать в проекцию файла. Взгляните на код, который иллюстрирует этот пример:

```
int WINAPI WinMain(HINSTANCE hinstExe, HINSTANCE,
    PSTR pszCmdLine, int nCmdShow) {

    // создаем объект "проекция файла";
    // его описатель разрешает доступ как для чтения, так и для записи
    HANDLE hFileMapRW = CreateFileMapping(INVALID_HANDLE_VALUE,
        NULL, PAGE_READWRITE, 0, 10240, NULL);

    // создаем другой описатель на тот же объект;
    // этот описатель разрешает доступ только для чтения
    HANDLE hFileMapRO;
    DuplicateHandle(GetCurrentProcess(), hFileMapRW, GetCurrentProcess(),
        &hFileMapRO, FILE_MAP_READ, FALSE, 0);

    // вызываем функцию, которая не должна ничего записывать в проекцию файла
    ReadFromTheFileMapping(hFileMapRO);

    // закрываем объект "проекция файла", доступный только для чтения
    CloseHandle(hFileMapRO);

    // проекция файла нам по-прежнему полностью доступна через hFileMapRW
    :

    // если проекция файла больше не нужна основному коду, закрываем ее
    CloseHandle(hFileMapRW);
}
```

ЧАСТЬ II

НАЧИНАЕМ РАБОТАТЬ



Процессы

Эта глава о том, как система управляет выполняемыми приложениями. Сначала я определию понятие «процесс» и объясню, как система создает объект ядра «процесс». Затем я покажу, как управлять процессом, используя сопоставленный с ним объект ядра. Далее мы обсудим атрибуты (или свойства) процесса и поговорим о нескольких функциях, позволяющих обращаться к этим свойствам и изменять их. Я расскажу также о функциях, которые создают (порождают) в системе дополнительные процессы. Ну и, конечно, описание процессов было бы неполным, если бы я не рассмотрел механизм их завершения. О'кэй, приступим.

Процесс обычно определяют как экземпляр выполняемой программы, и он состоит из двух компонентов:

- объекта ядра, через который операционная система управляет процессом. Там же хранится статистическая информация о процессе;
- адресного пространства, в котором содержится код и данные всех EXE- и DLL-модулей. Именно в нем находятся области памяти, динамически распределяемой для стеков потоков и других нужд.

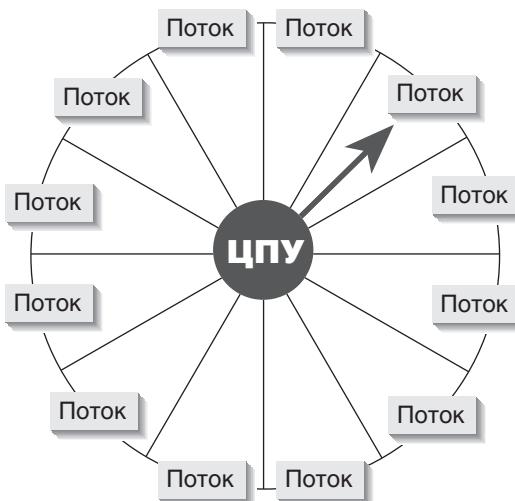


Рис. 4-1. Операционная система выделяет потокам кванты времени по принципу карусели

Процессы инертны. Чтобы процесс что-нибудь выполнил, в нем нужно создать поток. Именно потоки отвечают за исполнение кода, содержащегося в адресном пространстве процесса. В принципе, один процесс может владеть несколькими потоками, и тогда они «одновременно» исполняют код в адресном пространстве процесса.

Для этого каждый поток должен располагать собственным набором регистров процессора и собственным стеком. В каждом процессе есть минимум один поток. Если бы у процесса не было ни одного потока, ему нечего было бы делать на этом свете, и система автоматически уничтожила бы его вместе с выделенным ему адресным пространством.

Чтобы все эти потоки работали, операционная система отводит каждому из них определенное процессорное время. Выделяя потокам отрезки времени (называемые *квантами*) по принципу карусели, она создает тем самым иллюзию одновременного выполнения потоков. Рис. 4-1 иллюстрирует распределение процессорного времени между потоками на машине с одним процессором. Если в машине установлено более одного процессора, алгоритм работы операционной системы значительно усложняется (в этом случае система стремится сбалансировать нагрузку между процессорами).

При создании процесса первый (точнее, первичный) поток создается системой автоматически. Далее этот поток может породить другие потоки, те в свою очередь — новые и т. д.

WINDOWS 2000 Windows 2000 в полной мере использует возможности машин с несколькими процессорами. Например, эту книгу я писал, сидя за машиной с двумя процессорами. Windows 2000 способна закрепить каждый поток за отдельным процессором, и тогда два потока исполняются действительно одновременно. Ядро Windows 2000 полностью поддерживает распределение процессорного времени между потоками и управление ими на таких системах. Вам не придется делать ничего особенного в своем коде, чтобы задействовать преимущества многопроцессорной машины.

WINDOWS 98 Windows 98 работает только с одним процессором. Даже если у компьютера несколько процессоров, под управлением Windows 98 действует лишь один из них — остальные пристаивают.

Ваше первое Windows-приложение

Windows поддерживает два типа приложений: основанные на графическом интерфейсе (graphical user interface, GUI) и консольные (console user interface, CUI). У приложений первого типа внешний интерфейс чисто графический. GUI-приложения создают окна, имеют меню, взаимодействуют с пользователем через диалоговые окна и вообще пользуются всей стандартной «Windows'овской» начинкой. Почти все стандартные программы Windows — Notepad, Calculator, Wordpad и др. — являются GUI-приложениями. Приложения консольного типа работают в текстовом режиме: они не формируют окна, не обрабатывают сообщения и не требуют GUI. И хотя консольные приложения на экране тоже размещаются в окне, в нем содержится только текст. Командные процессоры вроде Cmd.exe (в Windows 2000) или Command.com (в Windows 98) — типичные образцы подобных приложений.

Вместе с тем граница между двумя типами приложений весьма условна. Можно, например, создать консольное приложение, способное отображать диалоговые окна. Скажем, в командном процессоре вполне может быть специальная команда, открывающая графическое диалоговое окно со списком команд; вроде мелочь — а избавляет от запоминания лишней информации. В то же время можно создать и GUI-приложение, выводящее текстовые строки в консольное окно. Я сам часто писал такие про-

грамммы: создав консольное окно, я пересыпал в него отладочную информацию, связанную с исполняемым приложением. Но, конечно, графический интерфейс предпочтительнее, чем старомодный текстовый. Как показывает опыт, приложения на основе GUI «дружественнее» к пользователю, а значит и более популярны.

Когда Вы создаете проект приложения, Microsoft Visual C++ устанавливает такие ключи для компоновщика, чтобы в исполняемом файле был указан соответствующий тип подсистемы. Для CUI-программ используется ключ /SUBSYSTEM:CONSOLE, а для GUI-приложений — /SUBSYSTEM:WINDOWS. Когда пользователь запускает приложение, загрузчик операционной системы проверяет номер подсистемы, хранящийся в заголовке образа исполняемого файла, и определяет, что это за программа — GUI или CUI. Если номер указывает на приложение последнего типа, загрузчик автоматически создает текстовое консольное окно, а если номер свидетельствует о противоположном — просто загружает программу в память. После того как приложение начинает работать, операционная система больше не интересуется, к какому типу оно относится.

Во всех Windows-приложениях должна быть входная функция, за реализацию которой отвечаете Вы. Существует четыре такие функции:

```
int WINAPI WinMain(
    HINSTANCE hinstExe,
    HINSTANCE,
    PSTR pszCmdLine,
    int nCmdShow);

int WINAPI wWinMain(
    HINSTANCE hinstExe,
    HINSTANCE,
    PWSTR pszCmdLine,
    int nCmdShow);

int __cdecl main(
    int argc,
    char *argv[],
    char *envp[]);

int __cdecl wmain(
    int argc,
    wchar_t *argv[],
    wchar_t *envp[]);
```

На самом деле входная функция операционной системой не вызывается. Вместо этого происходит обращение к стартовой функции из библиотеки C/C++. Она инициализирует библиотеку C/C++, чтобы можно было вызывать такие функции, как *malloc* и *free*, а также обеспечивает корректное создание любых объявленных Вами глобальных и статических C++-объектов до того, как начнется выполнение Вашего кода. В следующей таблице показано, в каких случаях реализуются те или иные входные функции.

Тип приложения	Входная функция	Стартовая функция, встраиваемая в Ваш исполняемый файл
GUI-приложение, работающее с ANSI-символами и строками	<i>WinMain</i>	<i>WinMainCRTStartup</i>
GUI-приложение, работающее с Unicode-символами и строками	<i>wWinMain</i>	<i>wWinMainCRTStartup</i>
CUI-приложение, работающее с ANSI-символами и строками	<i>main</i>	<i>mainCRTStartup</i>
CUI-приложение, работающее с Unicode-символами и строками	<i>wmain</i>	<i>wmainCRTStartup</i>

Нужную стартовую функцию в библиотеке C/C++ выбирает компоновщик при сборке исполняемого файла. Если указан ключ /SUBSYSTEM:WINDOWS, компоновщик ищет в Вашем коде функцию *WinMain* или *wWinMain*. Если ни одной из них нет, он сообщает об ошибке «unresolved external symbol» («неразрешенный внешний символ»); в ином случае — выбирает *WinMainCRTStartup* или *wWinMainCRTStartup* соответственно.

Аналогичным образом, если задан ключ /SUBSYSTEM:CONSOLE, компоновщик ищет в коде функцию *main* или *wmain* и выбирает соответственно *mainCRTStartup* или *wmainCRTStartup*; если в коде нет ни *main*, ни *wmain*, сообщается о той же ошибке — «unresolved external symbol».

Но не многие знают, что в проекте можно вообще не указывать ключ /SUBSYSTEM компоновщика. Если Вы так и сделаете, компоновщик будет сам определять подсистему для Вашего приложения. При компоновке он проверит, какая из четырех функций (*WinMain*, *wWinMain*, *main* или *wmain*) присутствует в Вашем коде, и на основании этого выберет подсистему и стартовую функцию из библиотеки C/C++.

Одна из частых ошибок, допускаемых теми, кто лишь начинает работать с Visual C++, — выбор неверного типа проекта. Например, разработчик хочет создать проект Win32 Application, а сам включает в код функцию *main*. При его сборке он получает сообщение об ошибке, так как для проекта Win32 Application в командной строке компоновщика автоматически указывается ключ /SUBSYSTEM:WINDOWS, который требует присутствия в коде функции *WinMain* или *wWinMain*. В этот момент разработчик может выбрать один из четырех вариантов дальнейших действий:

- заменить *main* на *WinMain*. Как правило, это не лучший вариант, поскольку разработчик скорее всего и хотел создать консольное приложение;
- открыть новый проект, на этот раз — Win32 Console Application, и перенести в него все модули кода. Этот вариант весьма утомителен, и возникает ощущение, будто начинаешь все заново;
- открыть вкладку Link в диалоговом окне Project Settings и заменить ключ /SUBSYSTEM:WINDOWS на /SUBSYSTEM:CONSOLE. Некоторые думают, что это единственный вариант;
- открыть вкладку Link в диалоговом окне Project Settings и вообще убрать ключ /SUBSYSTEM:WINDOWS. Я предпочитаю именно этот способ, потому что он самый гибкий. Компоновщик сам сделает все, что надо, в зависимости от входной функции, которую Вы реализуете в своем коде. Никак не пойму, почему это не предлагается по умолчанию при создании нового проекта Win32 Application или Win32 Console Application.

Все стартовые функции из библиотеки С/С++ делают практически одно и то же. Разница лишь в том, какие строки они обрабатывают (в ANSI или Unicode) и какую входную функцию вызывают после инициализации библиотеки. Кстати, с Visual C++ поставляется исходный код этой библиотеки, и стартовые функции находятся в файле CRt0.c. А теперь рассмотрим, какие операции они выполняют:

- считывают указатель на полную командную строку нового процесса;
- считывают указатель на переменные окружения нового процесса;
- инициализируют глобальные переменные из библиотеки С/С++, доступ к которым из Вашего кода обеспечивается включением файла StdLib.h. Список этих переменных приведен в таблице 4-1;
- инициализируют кучу (динамически распределяемую область памяти), используемую С-функциями выделения памяти (т. е. *malloc* и *calloc*) и другими процедурами низкоуровневого ввода-вывода;
- вызывают конструкторы всех глобальных и статических объектов С++-классов.

Закончив эти операции, стартовая функция обращается к входной функции в Вашей программе. Если Вы написали ее в виде *wWinMain*, то она вызывается так:

```
GetStartupInfo(&StartupInfo);
int nMainRetVal = wWinMain(GetModuleHandle(NULL), NULL, pszCommandLineUnicode,
(StartupInfo.dwFlags & STARTF_USESHOWWINDOW)
? StartupInfo.wShowWindow : SW_SHOWDEFAULT);
```

А если Вы предпочли *WinMain*, то:

```
GetStartupInfo(&StartupInfo);
int nMainRetVal = WinMain(GetModuleHandle(NULL), NULL, pszCommandLineANSI,
(StartupInfo.dwFlags & STARTF_USESHOWWINDOW)
? StartupInfo.wShowWindow : SW_SHOWDEFAULT);
```

И, наконец, то же самое для функций *wmain* и *main*:

```
int nMainRetVal = wmain(__argc, __wargv, __wenviron);
int nMainRetVal = main(__argc, __argv, __environ);
```

Когда Ваша входная функция возвращает управление, стартовая обращается к функции *exit* библиотеки С/С++ и передает ей значение *nMainRetVal*. Функция *exit* выполняет следующие операции:

- вызывает все функции, зарегистрированные вызовами функции *_onexit*;
- вызывает деструкторы всех глобальных и статических объектов С++-классов;
- вызывает Windows-функцию *ExitProcess*, передавая ей значение *nMainRetVal*. Это заставляет операционную систему уничтожить Ваш процесс и установить код его завершения.

Имя переменной	Тип	Описание
<i>_osver</i>	<i>unsigned int</i>	Версия сборки операционной системы. Например, у Windows 2000 Beta 3 этот номер был 2031, соответственно <i>_osver</i> равна 2031.
<i>_winmajor</i>	<i>unsigned int</i>	Основной номер версии Windows в шестнадцатеричной форме. Для Windows 2000 это значение равно 5.

Таблица 4-1. Глобальные переменные из библиотеки С/С++, доступные Вашим программам

продолжение

Имя переменной	Тип	Описание
<code>_winminor</code>	<code>unsigned int</code>	Дополнительный номер версии Windows в шестнадцатеричной форме. Для Windows 2000 это значение равно 0.
<code>_winver</code>	<code>unsigned int</code>	Вычисляется как (<code>_winmajor << 8</code>) + <code>_winminor</code> .
<code>_argc</code>	<code>unsigned int</code>	Количество аргументов, переданных в командной строке.
<code>_argv</code> <code>_wargv</code>	<code>char **</code> <code>wchar_t **</code>	Массив размером <code>_argc</code> с указателями на ANSI- или Unicode-строки. Каждый элемент массива указывает на один из аргументов командной строки.
<code>_environ</code> <code>_wenviron</code>	<code>char **</code> <code>wchar_t **</code>	Массив указателей на ANSI- или Unicode-строки. Каждый элемент массива указывает на строку — переменную окружения.
<code>_pgmptr</code> <code>_wpgmptr</code>	<code>char **</code> <code>wchar_t **</code>	Полный путь и имя (в ANSI или Unicode) запускаемой программы.

Описатель экземпляра процесса

Любому EXE- или DLL-модулю, загружаемому в адресное пространство процесса, присваивается уникальный описатель экземпляра. Описатель экземпляра Вашего EXE-файла передается как первый параметр функции (*w*)*WinMain* — *hinstExe*. Это значение обычно требуется при вызовах функций, загружающих те или иные ресурсы. Например, чтобы загрузить из образа EXE-файла такой ресурс, как значок, надо вызвать:

```
HICON LoadIcon(
    HINSTANCE hinst,
    PCTSTR pszIcon);
```

Первый параметр в *LoadIcon* указывает, в каком файле (EXE или DLL) содержится интересующий Вас ресурс. Многие приложения сохраняют параметр *hinstExe* функции (*w*)*WinMain* в глобальной переменной, благодаря чему он доступен из любой части кода EXE-файла.

В документации Platform SDK утверждается, что некоторые Windows-функции требуют параметр типа `HMODULE`. Пример — функция *GetModuleFileName*:

```
DWORD GetModuleFileName(
    HMODULE hinstModule,
    PTSTR pszPath,
    DWORD cchPath);
```



Как оказалось, `HMODULE` и `HINSTANCE` — это одно и то же. Встретив в документации указание передать какой-то функции `HMODULE`, смело передавайте `HINSTANCE`, и наоборот. Они существуют в таком виде лишь потому, что в 16-разрядной Windows идентифицировали совершенно разные вещи.

Истинное значение параметра *hinstExe* функции (*w*)*WinMain* — базовый адрес в памяти, определяющий ту область в адресном пространстве процесса, куда был загружен образ данного EXE-файла. Например, если система открывает исполняемый файл и загружает его содержимое по адресу 0x00400000, то *hinstExe* функции (*w*)*WinMain* получает значение 0x00400000.

Базовый адрес, по которому загружается приложение, определяется компоновщиком. Разные компоновщики выбирают и разные (по умолчанию) базовые адреса. Компоновщик Visual C++ использует по умолчанию базовый адрес 0x00400000 — самый нижний в Windows 98, начиная с которого в ней допускается загрузка образа исполняемого файла. Указав параметр /BASE: *адрес* (в случае компоновщика от Microsoft), можно изменить базовый адрес, по которому будет загружаться приложение.

При попытке загрузить исполняемый файл в Windows 98 по базовому адресу ниже 0x00400000 загрузчик переместит его на другой адрес. Это увеличит время загрузки приложения, но оно по крайней мере будет выполнено. Если Вы разрабатываете программы и для Windows 98, и для Windows 2000, сделайте так, чтобы приложение загружалось по базовому адресу не ниже 0x00400000.

Функция *GetModuleHandle*:

```
MODULE GetModuleHandle(  
    PCTSTR pszModule);
```

возвращает описатель/базовый адрес, указывающий, куда именно (в адресном пространстве процесса) загружается EXE- или DLL-файл. При вызове этой функции имя нужного EXE- или DLL-файла передается как строка с нулевым символом в конце. Если система находит указанный файл, *GetModuleHandle* возвращает базовый адрес, по которому располагается образ данного файла. Если же файл системой не найден, функция возвращает NULL. Кроме того, можно вызвать эту функцию, передав ей NULL вместо параметра *pszModule*, — тогда Вы узнаете базовый адрес EXE-файла. Именно это и делает стартовый код из библиотеки C/C++ при вызове (*w*)*WinMain* из Вашей программы.

Есть еще две важные вещи, касающиеся *GetModuleHandle*. Во-первых, она проверяет адресное пространство только того процесса, который ее вызвал. Если этот процесс не использует никаких функций, связанных со стандартными диалоговыми окнами, то, вызвав *GetModuleHandle* и передав ей аргумент «ComDlg32», Вы получите NULL — пусть даже модуль ComDlg32.dll и загружен в адресное пространство какого-нибудь другого процесса. Во-вторых, вызов этой функции и передача ей NULL дает в результате базовый адрес EXE-файла в адресном пространстве процесса. Так что, вызывая функцию в виде *GetModuleHandle(NULL)* — даже из кода в DLL, — Вы получаете базовый адрес EXE-, а не DLL-файла.

Описатель предыдущего экземпляра процесса

Я уже говорил, что стартовый код из библиотеки C/C++ всегда передает в функцию (*w*)*WinMain* параметр *hinstExePrev* как NULL. Этот параметр предусмотрен исключительно для совместимости с 16-разрядными версиями Windows и не имеет никакого смысла для Windows-приложений. Поэтому я всегда пишу заголовок (*w*)*WinMain* так:

```
int WINAPI WinMain(  
    HINSTANCE hinstExe,  
    HINSTANCE,  
    PSTR pszCmdLine,  
    int nCmdShow);
```

Поскольку у второго параметра нет имени, компилятор не выдает предупреждение «parameter not referenced» («нет ссылки на параметр»).

Командная строка процесса

При создании новому процессу передается командная строка, которая почти никогда не бывает пустой — как минимум, она содержит имя исполняемого файла, использованного при создании этого процесса. Однако, как Вы увидите ниже (при обсуждении функции *CreateProcess*), возможны случаи, когда процесс получает командную строку, состоящую из единственного символа — нуля, завершающего строку. В момент запуска приложения стартовый код из библиотеки C/C++ считывает командную строку процесса, пропускает имя исполняемого файла и заносит в параметр *pszCmdLine* функции (*w*)*WinMain* указатель на оставшуюся часть командной строки.

Параметр *pszCmdLine* всегда указывает на ANSI-строку. Но, заменив *WinMain* на *wWinMain*, Вы получите доступ к Unicode-версии командной строки для своего процесса.

Программа может анализировать и интерпретировать командную строку как угодно. Поскольку *pszCmdLine* относится к типу PSTR, а не PCSTR, не стесняйтесь и записывайте строку прямо в буфер, на который указывает этот параметр, но ни при каких условиях не переступайте границу буфера. Лично я всегда рассматриваю этот буфер как «только для чтения». Если в командную строку нужно внести изменения, я сначала копирую буфер, содержащий командную строку, в локальный буфер (в своей программе), который затем и модифицирую.

Указатель на полную командную строку процесса можно получить и вызовом функции *GetCommandLine*:

```
PTSTR GetCommandLine();
```

Она возвращает указатель на буфер, содержащий полную командную строку, включая полное имя (вместе с путем) исполняемого файла.

Во многих приложениях безусловно удобнее использовать командную строку, предварительно разбитую на отдельные компоненты, доступ к которым приложение может получить через глобальные переменные *_argc* и *_argv* (или *_wargv*). Функция *CommandLineToArgvW* расщепляет Unicode-строку на отдельные компоненты:

```
PWSTR CommandLineToArgvW(
    PWSTR pszCmdLine,
    int pNumArgs);
```

Буква *W* в конце имени этой функции намекает на «широкие» (wide) символы и подсказывает, что функция существует только в Unicode-версии. Параметр *pszCmdLine* указывает на командную строку. Его обычно получают предварительным вызовом *GetCommandLineW*. Параметр *pNumArgs* — это адрес целочисленной переменной, в которой задается количество аргументов в командной строке. Функция *CommandLineToArgvW* возвращает адрес массива указателей на Unicode-строки.

CommandLineToArgvW выделяет нужную память автоматически. Большинство приложений не освобождает эту память, полагаясь на операционную систему, которая проводит очистку ресурсов по завершении процесса. И такой подход вполне приемлем. Но если Вы хотите сами освободить эту память, сделайте так:

```
int pNumArgs;
PWSTR *ppArgv = CommandLineToArgvW(GetCommandLineW(), &pNumArgs);
```

см. след. стр.

```
// используйте эти аргументы...
if (*ppArgv[1] == L'x') {
    :
// освободите блок памяти
HeapFree(GetProcessHeap(), 0, ppArgv);
```

Переменные окружения

С любым процессом связан блок переменных окружения — область памяти, выделенная в адресном пространстве процесса. Каждый блок содержит группу строк такого вида:

```
VarName1=VarValue1\0
VarName2=VarValue2\0
VarName3=VarValue3\0
:
VarNameX=VarValueX\0
\0
```

Первая часть каждой строки — имя переменной окружения. За ним следует знак равенства и значение, присваиваемое переменной. Строки в блоке переменных окружения должны быть отсортированы в алфавитном порядке по именам переменных.

Знак равенства разделяет имя переменной и ее значение, так что его нельзя использовать как символ в имени переменной. Важную роль играют и пробелы. Например, объявив две переменные:

```
XYZ= Windows      ( обратите внимание на пробел за знаком равенства)
ABC=Windows
```

и сравнив значения переменных *XYZ* и *ABC*, Вы увидите, что система их различает, — она учитывает любой пробел, поставленный перед знаком равенства или после него. Вот что будет, если записать, скажем, так:

```
XYZ =Home        ( обратите внимание на пробел перед знаком равенства)
XYZ=Work
```

Вы получите первую переменную с именем «*XYZ*», содержащую строку «*Home*», и вторую переменную «*XYZ*», содержащую строку «*Work*».

Конец блока переменных окружения помечается дополнительным нулевым символом.

WINDOWS 98 Чтобы создать исходный набор переменных окружения для Windows 98, надо модифицировать файл Autoexec.bat, поместив в него группу строк SET в виде:
SET VarName=VarValue

При перезагрузке система учтет новое содержимое файла Autoexec.bat, и тогда любые заданные Вами переменные окружения станут доступны всем процессам, инициируемым в сеансе работы с Windows 98.

WINDOWS 2000 При регистрации пользователя на входе в Windows 2000 система создает процесс-оболочку, связывая с ним группу строк — переменных окружения. Система получает начальные значения этих строк, анализируя два раздела в реестре. В первом:

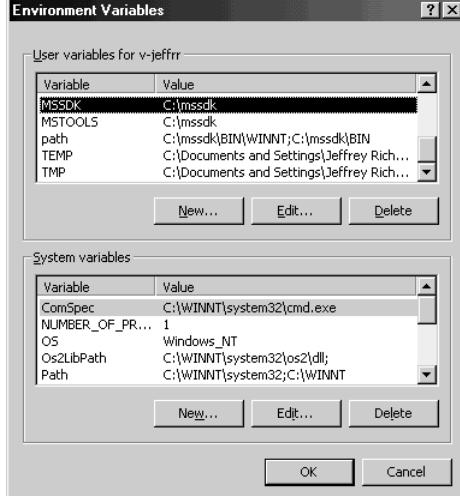
HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\SessionManager\Environment

содержится список переменных окружения, относящихся к системе, а во втором:

HKEY_CURRENT_USER\Environment

находится список переменных окружения, относящихся к пользователю, который в настоящее время зарегистрирован в системе.

Пользователь может добавлять, удалять или изменять любые переменные через аплет System из Control Panel. В этом аплете надо открыть вкладку Advanced и щелкнуть кнопку Environment Variables — тогда на экране появится следующее диалоговое окно.



Модифицировать переменные из списка System Variables разрешается только пользователю с правами администратора.

Кроме того, для модификации записей в реестре Ваша программа может обращаться к Windows-функциям, позволяющим манипулировать с реестром. Однако, чтобы изменения вступили в силу, пользователь должен выйти из системы и вновь войти в нее. Некоторые приложения типа Explorer, Task Manager или Control Panel могут обновлять свои блоки переменных окружения на базе новых значений в реестре, когда их главные окна получают сообщение WM_SETTINGCHANGE. Например, если Вы, изменив реестр, хотите, чтобы какие-то приложения соответственно обновили свои блоки переменных окружения, вызовите:

```
SendMessage(HWND_BROADCAST, WM_SETTINGCHANGE,
0, (LPARAM) TEXT("Environment"));
```

Обычно дочерний процесс наследует набор переменных окружения от родительского. Однако последний способен управлять тем, какие переменные окружения наследуются дочерним процессом, а какие — нет. Но об этом я расскажу, когда мы най-

мемся функцией *CreateProcess*. Под наследованием я имею в виду, что дочерний процесс получает свою копию блока переменных окружения от родительского, а не то, что дочерний и родительский процессы совместно используют один и тот же блок. Так что дочерний процесс может добавлять, удалять или модифицировать переменные в своем блоке, и эти изменения не затронут блок, принадлежащий родительскому процессу.

Переменные окружения обычно применяются для тонкой настройки приложения. Пользователь создает и инициализирует переменную окружения, затем запускает приложение, и оно, обнаружив эту переменную, проверяет ее значение и соответствующим образом настраивается.

Увы, многим пользователям не под силу разобраться в переменных окружения, а значит, трудно указать правильные значения. Ведь для этого надо не только хорошо знать синтаксис переменных, но и, конечно, понимать, что стоит за теми или иными их значениями. С другой стороны, почти все (а может, и все) приложения, основанные на GUI, дают возможность тонкой настройки через диалоговые окна. Такой подход, естественно, нагляднее и проще.

А теперь, если у Вас еще не пропало желание манипулировать переменными окружения, поговорим о предназначенных для этой цели функциях. *GetEnvironmentVariable* позволяет выявлять присутствие той или иной переменной окружения и определять ее значение:

```
DWORD GetEnvironmentVariable(
    PCTSTR pszName,
    PTSTR pszValue,
    DWORD cchValue);
```

При вызове *GetEnvironmentVariable* параметр *pszName* должен указывать на имя интересующей Вас переменной, *pszValue* — на буфер, в который будет помещено значение переменной, а в *cchValue* следует сообщить размер буфера в символах. Функция возвращает либо количество символов, скопированных в буфер, либо 0, если ей удалось обнаружить переменную окружения с таким именем.

Кстати, в реестре многие строки содержат подставляемые части, например:

```
%USERPROFILE%\My Documents
```

Часть, заключенная в знаки процента, является подставляемой. В данном случае в строку должно быть подставлено значение переменной окружения *USERPROFILE*. На моей машине эта переменная выглядит так:

```
C:\Documents and Settings\Administrator
```

После подстановки переменной в строку реестра получим:

```
C:\Documents and Settings\Administrator\My Documents
```

Поскольку такие подстановки делаются очень часто, в Windows есть функция *ExpandEnvironmentStrings*:

```
DWORD ExpandEnvironmentStrings(
    PCTSTR pszSrc,
    PTSTR pszDst,
    DWORD nSize);
```

Параметр *pszSrc* принимает адрес строки, содержащей подставляемые части, а параметр *pszDst* — адрес буфера, в который записывается развернутая строка. Параметр *nSize* определяет максимальный размер буфера в символах.

Наконец, функция *SetEnvironmentVariable* позволяет добавлять, удалять и модифицировать значение переменной:

```
DWORD SetEnvironmentVariable(
    PCTSTR pszName,
    PCTSTR pszValue);
```

Она устанавливает ту переменную, на чье имя указывает параметр *pszName*, и присваивает ей значение, заданное параметром *pszValue*. Если такая переменная уже существует, функция модифицирует ее значение. Если же в *pszValue* содержится NULL, переменная удаляется из блока.

Для манипуляций с блоком переменных окружения всегда используйте именно эти функции. Как я уже говорил, строки в блоке переменных нужно отсортировать в алфавитном порядке по именам переменных (тогда *GetEnvironmentVariable* быстрее находит нужные переменные), а *SetEnvironmentVariable* как раз и следит за порядком расположения переменных.

Привязка к процессорам

Обычно потоки внутри процесса могут выполняться на любом процессоре компьютера. Однако их можно закрепить за определенным подмножеством процессоров из числа имеющихся на компьютере. Это свойство называется *привязкой к процессорам* (processor affinity) и подробно обсуждается в главе 7. Дочерние процессы наследуют привязку к процессорам от родительских.

Режим обработки ошибок

С каждым процессом связан набор флагов, сообщающих системе, каким образом процесс должен реагировать на серьезные ошибки: повреждения дисковых носителей, необрабатываемые исключения, ошибки операций поиска файлов и неверное выравнивание данных. Процесс может указать системе, как обрабатывать каждую из этих ошибок, через функцию *SetErrorMode*:

```
UINT SetErrorMode(UINT fuErrorMode);
```

Параметр *fuErrorMode* — это набор флагов, комбинируемых побитовой операцией OR:

Флаг	Описание
SEM_FAILCRITICALERRORS	Система не выводит окно с сообщением от обработчика критических ошибок и возвращает ошибку в вызывающий процесс
SEM_NOGPFAULTERRORBOX	Система не выводит окно с сообщением о нарушении общей защиты; этим флагом манипулируют только отладчики, самостоятельно обрабатывающие нарушения общей защиты с помощью обработчика исключений
SEM_NOOPENFILEERRORBOX	Система не выводит окно с сообщением об отсутствии искомого файла
SEM_NOALIGNMENTFAULTEXCEPT	Система автоматически исправляет нарушения в выравнивании данных, и они становятся невидимы приложению; этот флаг не действует на процессорах x86

По умолчанию дочерний процесс наследует от родительского флаги, указывающие на режим обработки ошибок. Иначе говоря, если у процесса в данный момент установлен флаг SEM_NOGPFAULTERRORBOX и он порождает другой процесс, этот

флаг будет установлен и у дочернего процесса. Однако «наследник» об этом не уведомляется, и он вообще может быть не рассчитан на обработку ошибок такого типа (в данном случае — нарушений общей защиты). В результате, если в одном из потоков дочернего процесса все-таки произойдет подобная ошибка, этот процесс может завершиться, ничего не сообщив пользователю. Но родительский процесс способен предотвратить наследование дочерним процессом своего режима обработки ошибок, указав при вызове функции *CreateProcess* флаг *CREATE_DEFAULT_ERROR_MODE* (о *CreateProcess* чуть позже).

Текущие диск и каталог для процесса

Текущий каталог текущего диска — то место, где Windows-функции ищут файлы и подкаталоги, если полные пути в соответствующих параметрах не указаны. Например, если поток в процессе вызывает функцию *CreateFile*, чтобы открыть какой-нибудь файл, а полный путь не задан, система просматривает список файлов в текущем каталоге текущего диска. Этот каталог отслеживается самой системой, и, поскольку такая информация относится ко всему процессу, смена текущего диска или каталога одним из потоков распространяется и на остальные потоки в данном процессе.

Поток может получать и устанавливать текущие каталог и диск для процесса с помощью двух функций:

```
DWORD GetCurrentDirectory()  
    DWORD cchCurDir,  
    PTSTR pszCurDir);  
  
BOOL SetCurrentDirectory(PCTSTR pszCurDir);
```

Текущие каталоги для процесса

Система отслеживает текущие диск и каталог для процесса, но не текущие каталоги на каждом диске. Однако в операционной системе предусмотрен кое-какой сервис для манипуляций с текущими каталогами на разных дисках. Он реализуется через переменные окружения конкретного процесса. Например:

```
=C:=C:\Utility\Bin  
=D:=D:\Program Files
```

Эти переменные указывают, что текущим каталогом на диске С является \Utility\Bin, а на диске D — Program Files.

Если Вы вызываете функцию, передавая ей путь с именем диска, отличного от текущего, система сначала просматривает блок переменных окружения и пытается найти переменную, связанную с именем указанного диска. Если таковая есть, система выбирает текущий каталог на заданном диске в соответствии с ее значением, нет — текущим каталогом считается корневой.

Скажем, если текущий каталог для процесса — C:\Utility\Bin и Вы вызываете функцию *CreateFile*, чтобы открыть файл D:\ReadMe.txt, система ищет переменную **=D:**. Поскольку переменная **=D:** существует, система пытается открыть файл ReadMe.txt в каталоге D:\Program Files. А если бы таковой переменной не было, система искала бы файл ReadMe.txt в корневом каталоге диска D. Кстати, файловые Windows-функции никогда не добавляют и не изменяют переменные окружения, связанные с именами дисков, а лишь считывают их значения.



Для смены текущего каталога вместо Windows-функции *SetCurrentDirectory* можно использовать функцию *_chdir* из библиотеки С. Внутренне она тоже обращается к *SetCurrentDirectory*, но, кроме того, способна добавлять или модифицировать переменные окружения, что позволяет запоминать в программе текущие каталоги на различных дисках.

Если родительский процесс создает блок переменных окружения и хочет передать его дочернему процессу, тот не наследует текущие каталоги родительского процесса автоматически. Вместо этого у дочернего процесса текущими на всех дисках становятся корневые каталоги. Чтобы дочерний процесс унаследовал текущие каталоги родительского, последний должен создать соответствующие переменные окружения (и сделать это до порождения другого процесса). Родительский процесс может узнать, какие каталоги являются текущими, вызвав *GetFullPathName*:

```
DWORD GetFullPathName(
    PCTSTR pszFile,
    DWORD cchPath,
    PTSTR pszPath,
    PTSTR *ppszFilePart);
```

Например, чтобы получить текущий каталог на диске С, функцию вызывают так:

```
TCHAR szCurDir[MAX_PATH];
DWORD GetFullPathName(TEXT("C:\\"), MAX_PATH, szCurDir, NULL);
```

Не забывайте, что переменные окружения процесса должны всегда храниться в алфавитном порядке. Поэтому переменные, связанные с дисками, обычно приходится размещать в самом начале блока.

Определение версии системы

Весьма часто приложению требуется определять, в какой версии Windows оно выполняется. Причин тому несколько. Например, программа может использовать функции защиты, заложенные в Windows API, но в полной мере эти функции реализованы лишь в Windows 2000.

Насколько я помню, функция *GetVersion* есть в API всех версий Windows:

```
DWORD GetVersion();
```

С этой простой функцией связана целая история. Сначала ее разработали для 16-разрядной Windows, и она должна была в старшем слове возвращать номер версии MS-DOS, а в младшем — номер версии Windows. Соответственно в каждом слове старший байт сообщал основной номер версии, младший — дополнительный номер версии.

Увы, программист, писавший ее код, слегка ошибся, и получилось так, что номера версии Windows поменялись местами: в старший байт попадал дополнительный номер, а в младший — основной. Поскольку многие программисты уже начали пользоваться этой функцией, Microsoft пришлось оставить все, как есть, и изменить документацию с учетом ошибки.

Из-за всей этой неразберихи вокруг *GetVersion* в Windows API включили новую функцию — *GetVersionEx*:

```
BOOL GetVersionEx(POSVERSIONINFO pVersionInformation);
```

Перед обращением к *GetVersionEx* программа должна создать структуру OSVERSIONINFOEX, показанную ниже, и передать ее адрес этой функции.

```
typedef struct {
    DWORD dwOSVersionInfoSize;
    DWORD dwMajorVersion;
    DWORD dwMinorVersion;
    DWORD dwBuildNumber;
    DWORD dwPlatformId;
    TCHAR szCSDVersion[128];
    WORD wServicePackMajor;
    WORD wServicePackMinor;
    WORD wSuiteMask;
    BYTE wProductType;
    BYTE wReserved;
} OSVERSIONINFOEX, *POSVERSIONINFOEX;
```

Эта структура — новинка Windows 2000. В остальных версиях Windows используется структура OSVERSIONINFO, в которой нет последних пяти элементов, присутствующих в структуре OSVERSIONINFOEX.

Обратите внимание, что каждому компоненту номера версии операционной системы соответствует свой элемент структуры. Это сделано специально — чтобы программисты не возились с выборкой данных из всяких там старших-младших байтовых слов (и не путались в них!); теперь программе гораздо проще сравнивать ожидаемый номер версии операционной системы с действительным. Назначение каждого элемента структуры OSVERSIONINFOEX описано в таблице 4-2.

Элемент	Описание
<i>dwOSVersionInfoSize</i>	Размер структуры; перед обращением к функции <i>GetVersionEx</i> должен быть заполнен вызовом <i>sizeof(OSVERSIONINFO)</i> или <i>sizeof(OSVERSIONINFOEX)</i>
<i>dwMajorVersion</i>	Основной номер версии операционной системы
<i>dwMinorVersion</i>	Дополнительный номер версии операционной системы
<i>dwBuildNumber</i>	Версия сборки данной системы
<i>dwPlatformId</i>	Идентификатор платформы, поддерживаемой данной системой; его возможные значения: VER_PLATFORM_WIN32s (Win32s), VER_PLATFORM_WIN32_WINDOWS (Windows 95/98), VER_PLATFORM_WIN32_NT (Windows NT или Windows 2000), VER_PLATFORM_WIN32_CEHH (Windows CE)
<i>szCSDVersion</i>	Этот элемент содержит текст — дополнительную информацию об установленной операционной системе
<i>wServicePackMajor</i>	Основной номер версии последнего установленного пакета исправлений (service pack)
<i>wServicePackMinor</i>	Дополнительный номер версии последнего установленного пакета исправлений

Таблица 4-2. Элементы структуры OSVERSIONINFOEX

продолжение

Элемент	Описание
<i>wSuiteMask</i>	Сообщает, какие программные пакеты (suites) доступны в системе; его возможные значения: VER_SUITE_SMALLBUSINESS, VER_SUITE_ENTERPRISE, VER_SUITE_BACKOFFICE, VER_SUITE_COMMUNICATIONS, VER_SUITE_TERMINAL, VER_SUITE_SMALLBUSINESS_RESTRICTED, VER_SUITE_EMBEDDEDNT, VER_SUITE_DATACENTER
<i>wProductType</i>	Сообщает, какой именно вариант операционной системы установлен; его возможные значения: VER_NT_WORKSTATION, VER_NT_SERVER, VER_NT_DOMAIN_CONTROLLER
<i>wReserved</i>	Зарезервирован на будущее

В Windows 2000 появилась новая функция, *VerifyVersionInfo*, которая сравнивает версию установленной операционной системы с тем, что требует Ваше приложение:

```
BOOL VerifyVersionInfo(
    POSVERSIONINFOEX pVersionInformation,
    DWORD dwTypeMask,
    DWORDLONG dwlConditionMask);
```

Чтобы использовать эту функцию, создайте структуру OSVERSIONINFOEX, запишите в ее элемент *dwOSVersionInfoSize* размер структуры, а потом инициализируйте любые другие элементы, важные для Вашей программы. При вызове *VerifyVersionInfo* параметр *dwTypeMask* указывает, какие элементы структуры Вы инициализировали. Этот параметр принимает любые комбинации следующих флагов: VER_MINORVERSION, VER_MAJORVERSION, VER_BUILDNUMBER, VER_PLATFORMID, VER_SERVICEPACKMINOR, VER_SERVICEPACKMAJOR, VER_SUITENAME и VER_PRODUCT_TYPE. Последний параметр, *dwlConditionMask*, является 64-разрядным значением, которое управляет тем, как именно функция сравнивает информацию о версии системы с нужными Вам данными.

Параметр *dwlConditionMask* устанавливает правила сравнения через сложный набор битовых комбинаций. Для создания требуемой комбинации используйте макрос *VER_SET_CONDITION*:

```
VER_SET_CONDITION(
    DWORDLONG dwlConditionMask,
    ULONG dwTypeBitMask,
    ULONG dwConditionMask)
```

Первый параметр, *dwlConditionMask*, идентифицирует переменную, битами которой Вы манипулируете. Вы не передаете адрес этой переменной, потому что *VER_SET_CONDITION* — макрос, а не функция. Параметр *dwTypeBitMask* указывает один элемент в структуре OSVERSIONINFOEX, который Вы хотите сравнить со своими данными. (Для сравнения нескольких элементов придется обращаться к *VER_SET_CONDITION* несколько раз подряд.) Флаги, передаваемые в этом параметре, идентичны передаваемым в параметре *dwTypeMask* функции *VerifyVersionInfo*.

Последний параметр макроса `VER_SET_CONDITION`, *dwConditionMask*, сообщает, как Вы хотите проводить сравнение. Он принимает одно из следующих значений: `VER_EQUAL`, `VER_GREATER`, `VER_GREATER_EQUAL`, `VER_LESS` или `VER_LESS_EQUAL`. Вы можете использовать эти значения в сравнениях по `VER_PRODUCT_TYPE`. Например, значение `VER_NT_WORKSTATION` меньше, чем `VER_NT_SERVER`. Но в сравнениях по `VER_SUITENAME` вместо этих значений применяется `VER_AND` (должны быть установлены все программные пакеты) или `VER_OR` (должен быть установлен хотя бы один из программных пакетов).

Подготовив набор условий, Вы вызываете `VerifyVersionInfo` и получаете ненулевое значение, если система отвечает требованиям Вашего приложения, или 0, если она не удовлетворяет этим требованиям или если Вы неправильно вызвали функцию. Чтобы определить, почему `VerifyVersionInfo` вернула 0, вызовите `GetLastError`. Если та вернет `ERROR_OLD_WIN_VERSION`, значит, Вы правильно вызвали функцию `VerifyVersionInfo`, но система не соответствует предъявленным требованиям.

Вот как проверить, установлена ли Windows 2000:

```
// готовим структуру OSVERSIONINFOEX, сообщая, что нам нужна Windows 2000
OSVERSIONINFOEX osver = { 0 };
osver.dwOSVersionInfoSize = sizeof(osver);
osver.dwMajorVersion = 5;
osver.dwMinorVersion = 0;
osver.dwPlatformId = VER_PLATFORM_WIN32_NT;

// формируем маску условий
DWORDLONG dwlConditionMask = 0;      // всегда инициализируйте этот элемент так
VER_SET_CONDITION(dwlConditionMask, VER_MAJORVERSION, VER_EQUAL);
VER_SET_CONDITION(dwlConditionMask, VER_MINORVERSION, VER_EQUAL);
VER_SET_CONDITION(dwlConditionMask, VER_PLATFORMID, VER_EQUAL);

// проверяем версию
if (VerifyVersionInfo(&osver, VER_MAJORVERSION | VER_MINORVERSION | VER_PLATFORMID,
    dwlConditionMask)) {
    // хост-система точно соответствует Windows 2000
} else {
    // хост-система не является Windows 2000
}
```

Функция *CreateProcess*

Процесс создается при вызове Вашим приложением функции *CreateProcess*:

```
BOOL CreateProcess(
    PCTSTR pszApplicationName,
    PTSTR pszCommandLine,
    PSECURITY_ATTRIBUTES psaProcess,
    PSECURITY_ATTRIBUTES psaThread,
    BOOL bInheritHandles,
    DWORD fdwCreate,
    PVOID pvEnvironment,
    PCTSTR pszCurDir,
    PSTARTUPINFO psiStartInfo,
    PPROCESS_INFORMATION ppiProcInfo);
```

Когда поток в приложении вызывает *CreateProcess*, система создает объект ядра «процесс» с начальным значением счетчика числа его пользователей, равным 1. Этот объект — не сам процесс, а компактная структура данных, через которую операционная система управляет процессом. (Объект ядра «процесс» следует рассматривать как структуру данных со статистической информацией о процессе.) Затем система создает для нового процесса виртуальное адресное пространство и загружает в него код и данные как для исполняемого файла, так и для любых DLL (если таковые требуются).

Далее система формирует объект ядра «поток» (со счетчиком, равным 1) для первичного потока нового процесса. Как и в первом случае, объект ядра «поток» — это компактная структура данных, через которую система управляет потоком. Первичный поток начинает с исполнения стартового кода из библиотеки С/С++, который в конечном счете вызывает функцию *WinMain*, *wWinMain*, *main* или *wmain* в Вашей программе. Если системе удастся создать новый процесс и его первичный поток, *CreateProcess* вернет TRUE.



CreateProcess возвращает TRUE до окончательной инициализации процесса. Это означает, что на данном этапе загрузчик операционной системы еще не искал все необходимые DLL. Если он не сможет найти хотя бы одну из DLL или корректно провести инициализацию, процесс завершится. Но, поскольку *CreateProcess* уже вернула TRUE, родительский процесс ничего не узнает об этих проблемах.

На этом мы закончим общее описание и перейдем к подробному рассмотрению параметров функции *CreateProcess*.

Параметры *pszApplicationName* и *pszCommandLine*

Эти параметры определяют имя исполняемого файла, которым будет пользоваться новый процесс, и командную строку, передаваемую этому процессу. Начнем с *pszCommandLine*.



Обратите внимание на тип параметра *pszCommandLine*: PTSTR. Он означает, что *CreateProcess* ожидает передачи адреса строки, которая не является константой. Дело в том, что *CreateProcess* в процессе своего выполнения модифицирует переданную командную строку, но перед возвратом управления восстанавливает ее.

Это очень важно: если командная строка содержится в той части образа Вашего файла, которая предназначена только для чтения, возникнет ошибка доступа. Например, следующий код приведет к такой ошибке, потому что Visual C++ 6.0 поместит строку «NOTEPAD» в память только для чтения:

```
STARTUPINFO si = { sizeof(si) };
PROCESS_INFORMATION pi;
CreateProcess(NULL, TEXT("NOTEPAD"), NULL, NULL,
    FALSE, 0, NULL, NULL, &si, &pi);
```

Когда *CreateProcess* попытается модифицировать строку, произойдет ошибка доступа. (В прежних версиях Visual C++ эта строка была бы размещена в памяти для чтения и записи, и вызовы *CreateProcess* не приводили бы к ошибкам доступа.)

см. след. стр.

Лучший способ решения этой проблемы — перед вызовом *CreateProcess* копировать константную строку во временный буфер:

```
STARTUPINFO si = { sizeof(si) };
PROCESS_INFORMATION pi;
TCHAR szCommandLine[] = TEXT("NOTEPA");
CreateProcess(NULL, szCommandLine, NULL, NULL,
    FALSE, 0, NULL, &si, &pi);
```

Возможно, Вас заинтересуют ключи /Gf и /GF компилятора Visual C++, которые исключают дублирование строк и запрещают их размещение в области только для чтения. (Также обратите внимание на ключ /ZI, который позволяет задействовать отладочную функцию Edit & Continue, поддерживаемую Visual Studio, и подразумевает активизацию ключа /GF.) В общем, лучшее, что можете сделать Вы, — использовать ключ /GF или создать временный буфер. А еще лучше, если Microsoft исправит функцию *CreateProcess*, чтобы та не морочила нам голову. Надеюсь, в следующей версии Windows так и будет.

Кстати, при вызове ANSI-версии *CreateProcess* в Windows 2000 таких проблем нет, поскольку в этой версии функции командная строка копируется во временный буфер (см. главу 2).

Параметр *pszCommandLine* позволяет указать полную командную строку, используемую функцией *CreateProcess* при создании нового процесса. Разбирая эту строку, функция полагает, что первый компонент в ней представляет собой имя исполняемого файла, который Вы хотите запустить. Если в имени этого файла не указано расширение, она считает его EXE. Далее функция приступает к поиску заданного файла и делает это в следующем порядке:

1. Каталог, содержащий EXE-файл вызывающего процесса.
2. Текущий каталог вызывающего процесса.
3. Системный каталог Windows.
4. Основной каталог Windows.
5. Каталоги, перечисленные в переменной окружения PATH.

Конечно, если в имени файла указан полный путь доступа, система сразу обращается туда и не просматривает эти каталоги. Найдя нужный исполняемый файл, она создает новый процесс и проецирует код и данные исполняемого файла на адресное пространство этого процесса. Затем обращается к процедурам стартового кода из библиотеки C/C++. Тот в свою очередь, как уже говорилось, анализирует командную строку процесса и передает (*w*)*WinMain* адрес первого (за именем исполняемого файла) аргумента как *pszCmdLine*.

Все, о чем я сказал, произойдет, только если параметр *pszApplicationName* равен NULL (что и бывает в 99% случаев). Вместо NULL можно передать адрес строки с именем исполняемого файла, который надо запустить. Однако тогда придется указать не только его имя, но и расширение, поскольку в этом случае имя не дополняется расширением EXE автоматически. *CreateProcess* предполагает, что файл находится в текущем каталоге (если полный путь не задан). Если в текущем каталоге файла нет, функция не станет искать его в других каталогах, и на этом все закончится.

Но даже при указанном в *pszApplicationName* имени файла *CreateProcess* все равно передает новому процессу содержимое параметра *pszCommandLine* как командную строку. Допустим, Вы вызвали *CreateProcess* так:

```
// размещаем строку пути в области памяти для чтения и записи
TCHAR szPath[ ] = TEXT("WORDPAD README.TXT");

// порождаем новый процесс
CreateProcess(TEXT("C:\\WINNT\\SYSTEM32\\NOTEPAD.EXE"), szPath, ...);
```

Система запускает Notepad, а в его командной строке мы видим «WORDPAD README.TXT». Странно, да? Но так уж она работает, эта функция *CreateProcess*. Упомянутая возможность, которую обеспечивает параметр *pszApplicationName*, на самом деле введена в *CreateProcess* для поддержки подсистемы POSIX в Windows 2000.

Параметры *psaProcess*, *psaThread* и *bInheritHandles*

Чтобы создать новый процесс, система должна сначала создать объекты ядра «процесс» и «поток» (для первичного потока процесса). Поскольку это объекты ядра, родительский процесс получает возможность связать с ними атрибуты защиты. Параметры *psaProcess* и *psaThread* позволяют определить нужные атрибуты защиты для объектов «процесс» и «поток» соответственно. В эти параметры можно занести NULL, и система закрепит за данными объектами дескрипторы защиты по умолчанию. В качестве альтернативы можно объявить и инициализировать две структуры SECURITY_ATTRIBUTES; тем самым Вы создадите и присвоите объектам «процесс» и «поток» свои атрибуты защиты.

Структуры SECURITY_ATTRIBUTES для параметров *psaProcess* и *psaThread* используются и для того, чтобы какой-либо из этих двух объектов получил статус наследуемого любым дочерним процессом. (О теории, на которой построено наследование описателей объектов ядра, я рассказывал в главе 3.)

Короткая программа на рис. 4-2 демонстрирует, как наследуются описатели объектов ядра. Будем считать, что процесс А порождает процесс В и заносит в параметр *psaProcess* адрес структуры SECURITY_ATTRIBUTES, в которой элемент *bInheritHandle* установлен как TRUE. Одновременно параметр *psaThread* указывает на другую структуру SECURITY_ATTRIBUTES, в которой значение элемента *bInheritHandle* — FALSE.

Создавая процесс В, система формирует объекты ядра «процесс» и «поток», а затем — в структуре, на которую указывает параметр *ppriProcInfo* (о нем поговорим позже), — возвращает их описатели процессу А, и с этого момента тот может манипулировать только что созданными объектами «процесс» и «поток».

Теперь предположим, что процесс А собирается вторично вызвать функцию *CreateProcess*, чтобы породить процесс С. Сначала ему нужно определить, стоит ли предоставить процессу С доступ к своим объектам ядра. Для этого используется параметр *bInheritHandles*. Если он приравнен TRUE, система передает процессу С все наследуемые описатели. В этом случае наследуется и описатель объекта ядра «процесс» процесса В. А вот описатель объекта «первичный поток» процесса В не наследуется ни при каком значении *bInheritHandles*. Кроме того, если процесс А вызывает *CreateProcess*, передавая через параметр *bInheritHandles* значение FALSE, процесс С не наследует никаких описателей, используемых в данный момент процессом А.

Inherit.c

```
*****  
Модуль: Inherit.c  
Автор: Copyright (c) 2000, Джейфри Рихтер (Jeffrey Richter)  
*****  
  
#include <Windows.h>  
  
int WINAPI WinMain(HINSTANCE hinstExe, HINSTANCE,  
PSTR pszCmdLine, int nCmdShow) {  
  
    // готовим структуру STARTUPINFO для создания процессов  
    STARTUPINFO si = { sizeof(si) };  
    SECURITY_ATTRIBUTES saProcess, saThread;  
    PROCESS_INFORMATION piProcessB, piProcessC;  
    TCHAR szPath[MAX_PATH];  
  
    // готовимся к созданию процесса В из процесса А;  
    // описатель, идентифицирующий новый объект "процесс",  
    // должен быть наследуемым  
    saProcess.nLength = sizeof(saProcess);  
    saProcess.lpSecurityDescriptor = NULL;  
    saProcess.bInheritHandle = TRUE;  
  
    // описатель, идентифицирующий новый объект "поток",  
    // НЕ должен быть наследуемым  
    saThread.nLength = sizeof(saThread);  
    saThread.lpSecurityDescriptor = NULL;  
    saThread.bInheritHandle = FALSE;  
  
    // порождаем процесс В  
    lstrcpy(szPath, TEXT("ProcessB"));  
    CreateProcess(NULL, szPath, &saProcess, &saThread,  
        FALSE, 0, NULL, NULL, &si, &piProcessB);  
  
    // структура pi содержит два описателя, относящиеся к процессу А:  
    // hProcess, который идентифицирует объект "процесс" процесса В  
    // и является наследуемым, и hThread, который идентифицирует объект  
    // "первичный поток" процесса В и НЕ является наследуемым  
  
    // готовимся создать процесс С из процесса А;  
    // так как в psaProcess и psaThread передаются NULL, описатели  
    // объектов "процесс" и "первичный поток" процесса С считаются  
    // ненаследуемыми по умолчанию  
  
    // если процесс А создаст еще один процесс, тот НЕ унаследует  
    // описатели объектов "процесс" и "первичный поток" процесса С  
  
    // поскольку в параметре bInheritHandles передается TRUE,  
    // процесс С унаследует описатель, идентифицирующий объект  
    // "процесс" процесса В, но НЕ описатель, идентифицирующий объект
```

Рис. 4-2. Пример, иллюстрирующий наследование описателей объектов ядра

Рис. 4-2. продолжение

```

// "первичный поток" того же процесса
lstrcpy(szPath, TEXT("ProcessC"));
CreateProcess(NULL, szPath, NULL, NULL,
    TRUE, 0, NULL, NULL, &si, &piProcessC);

return(0);
}

```

Параметр *fdwCreate*

Параметр *fdwCreate* определяет флаги, влияющие на то, как именно создается новый процесс. Флаги комбинируются булевым оператором OR.

- Флаг DEBUG_PROCESS дает возможность родительскому процессу проводить отладку дочернего, а также всех процессов, которые последним могут быть порождены. Если этот флаг установлен, система уведомляет родительский процесс (он теперь получает статус отладчика) о возникновении определенных событий в любом из дочерних процессов (а они получают статус отлаживаемых).
- Флаг DEBUG_ONLY_THIS_PROCESS аналогичен флагу DEBUG_PROCESS с тем исключением, что заставляет систему уведомлять родительский процесс о возникновении специфических событий только в одном дочернем процессе — его прямом потомке. Тогда, если дочерний процесс создаст ряд дополнительных, отладчик уже не уведомляется о событиях, «происходящих» в них.
- Флаг CREATE_SUSPENDED позволяет создать процесс и в то же время приостановить его первичный поток. Это позволяет родительскому процессу модифицировать содержимое памяти в адресном пространстве дочернего, изменять приоритет его первичного потока или включать этот процесс в задание (job) до того, как он получит шанс на выполнение. Внеся нужные изменения в дочерний процесс, родительский разрешает выполнение его кода вызовом функции *ResumeThread* (см. главу 7).
- Флаг DETACHED_PROCESS блокирует доступ процессу, иницииированному консольной программой, к созданному родительским процессом консольному окну и сообщает системе, что вывод следует перенаправить в новое окно. CUI-процесс, создаваемый другим CUI-процессом, по умолчанию использует консольное окно родительского процесса. (Вы, очевидно, заметили, что при запуске компилятора С из командного процессора новое консольное окно не создается; весь его вывод «подписывается» в нижнюю часть существующего консольного окна.) Таким образом, этот флаг заставляет новый процесс перенаправлять свой вывод в новое консольное окно.
- Флаг CREATE_NEW_CONSOLE приводит к созданию нового консольного окна для нового процесса. Имейте в виду, что одновременная установка флагов CREATE_NEW_CONSOLE и DETACHED_PROCESS недопустима.
- Флаг CREATE_NO_WINDOW не дает создавать никаких консольных окон для данного приложения и тем самым позволяет исполнять его без пользовательского интерфейса.

- Флаг CREATE_NEW_PROCESS_GROUP служит для модификации списка процессов, уведомляемых о нажатии клавиш Ctrl+C и Ctrl+Break. Если в системе одновременно исполняется несколько CUI-процессов, то при нажатии одной из упомянутых комбинаций клавиш система уведомляет об этом только процессы, включенные в группу. Указав этот флаг при создании нового CUI-процесса, Вы создаете и новую группу.
- Флаг CREATE_DEFAULT_ERROR_MODE сообщает системе, что новый процесс не должен наследовать режимы обработки ошибок, установленные в родительском (см. раздел, где я рассказывал о функции *SetErrorMode*).
- Флаг CREATE_SEPARATE_WOW_VDM полезен только при запуске 16-разрядного Windows-приложения в Windows 2000. Если он установлен, система создает отдельную виртуальную DOS-машину (Virtual DOS-machine, VDM) и запускает 16-разрядное Windows-приложение именно в ней. (По умолчанию все 16-разрядные Windows-приложения выполняются в одной, общей VDM.) Выполнение приложения в отдельной VDM дает большое преимущество: «рухнув», приложение уничтожит лишь эту VDM, а программы, выполняемые в других VDM, продолжат нормальную работу. Кроме того, 16-разрядные Windows-приложения, выполняемые в раздельных VDM, имеют и раздельные очереди ввода. Это значит, что, если одно приложение вдруг «зависнет», приложения в других VDM продолжат прием ввода. Единственный недостаток работы с несколькими VDM в том, что каждая из них требует значительных объемов физической памяти. Windows 98 выполняет все 16-разрядные Windows-приложения только в одной VDM, и изменить тут ничего нельзя.
- Флаг CREATE_SHARED_WOW_VDM полезен только при запуске 16-разрядного Windows-приложения в Windows 2000. По умолчанию все 16-разрядные Windows-приложения выполняются в одной VDM, если только не указан флаг CREATE_SEPARATE_WOW_VDM. Однако стандартное поведение Windows 2000 можно изменить, присвоив значение «yes» параметру DefaultSeparateVDM в разделе HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\WOW. (После модификации этого параметра систему надо перезагрузить.) Установив значение «yes», но указав флаг CREATE_SHARED_WOW_VDM, Вы вновь заставите Windows 2000 выполнять все 16-разрядные Windows-приложения в одной VDM.
- Флаг CREATE_UNICODE_ENVIRONMENT сообщает системе, что блок переменных окружения дочернего процесса должен содержать Unicode-символы. По умолчанию блок формируется на основе ANSI-символов.
- Флаг CREATE_FORCEDOS заставляет систему выполнять программу MS-DOS,строенную в 16-разрядное приложение OS/2.
- Флаг CREATE_BREAKAWAY_FROM_JOB позволяет процессу, включенному в задание, создать новый процесс, отделенный от этого задания (см. главу 5).

Параметр *fdwCreate* разрешает задать и класс приоритета процесса. Однако это необязательно и даже, как правило, не рекомендуется; система присваивает новому процессу класс приоритета по умолчанию. Возможные классы приоритета перечислены в следующей таблице.

Класс приоритета	Флаговый идентификатор
Idle (простаивающий)	IDLE_PRIORITY_CLASS
Below normal (ниже обычного)	BELOW_NORMAL_PRIORITY_CLASS
Normal (обычный)	NORMAL_PRIORITY_CLASS
Above normal (выше обычного)	ABOVE_NORMAL_PRIORITY_CLASS
High (высокий)	HIGH_PRIORITY_CLASS
Realtime (реального времени)	REALTIME_PRIORITY_CLASS

Классы приоритета влияют на распределение процессорного времени между процессами и их потоками. (Подробнее на эту тему см. главу 7.)



Классы приоритета BELOW_NORMAL_PRIORITY_CLASS и ABOVE_NORMAL_PRIORITY_CLASS введены лишь в Windows 2000; они не поддерживаются в Windows NT 4.0, Windows 95 или Windows 98.

Параметр *pvEnvironment*

Параметр *pvEnvironment* указывает на блок памяти, хранящий строки переменных окружения, которыми будет пользоваться новый процесс. Обычно вместо этого параметра передается NULL, в результате чего дочерний процесс наследует строки переменных окружения от родительского процесса. В качестве альтернативы можно вызвать функцию *GetEnvironmentStrings*:

```
PVOID GetEnvironmentStrings();
```

Она позволяет узнать адрес блока памяти со строками переменных окружения, используемых вызывающим процессом. Полученный адрес можно занести в параметр *pvEnvironment* функции *CreateProcess*. (Именно это и делает *CreateProcess*, если Вы передаете ей NULL вместо *pvEnvironment*.) Если этот блок памяти Вам больше не нужен, освободите его, вызвав функцию *FreeEnvironmentStrings*:

```
BOOL FreeEnvironmentStrings(PTSTR pszEnvironmentBlock);
```

Параметр *pszCurDir*

Он позволяет родительскому процессу установить текущие диск и каталог для дочернего процесса. Если его значение — NULL, рабочий каталог нового процесса будет тем же, что и у приложения, его породившего. А если он отличен от NULL, то должен указывать на строку (с нулевым символом в конце), содержащую нужный диск и каталог. Заметьте, что в путь надо включать и букву диска.

Параметр *psiStartInfo*

Этот параметр указывает на структуру STARTUPINFO:

```
typedef struct _STARTUPINFO {
    DWORD cb;
    PSTR lpReserved;
    PSTR lpDesktop;
    PSTR lpTitle;
    DWORD dwX;
    DWORD dwY;
```

см. след. стр.

```
DWORD dwXSize;
DWORD dwYSize;
DWORD dwXCountChars;
DWORD dwYCountChars;
DWORD dwFillAttribute;
DWORD dwFlags;
WORD wShowWindow;
WORD cbReserved2;
PBYTE lpReserved2;
HANDLE hStdInput;
HANDLE hStdOutput;
HANDLE hStdError;
} STARTUPINFO, *LPSTARTUPINFO;
```

Элементы структуры `STARTUPINFO` используются Windows-функциями при создании нового процесса. Надо сказать, что большинство приложений порождает процессы с атрибутами по умолчанию. Но и в этом случае Вы должны инициализировать все элементы структуры `STARTUPINFO` хотя бы нулевыми значениями, а в элемент `cb` — заносить размер этой структуры:

```
STARTUPINFO si = { sizeof(si) };
CreateProcess(..., &si, ...);
```

К сожалению, разработчики приложений часто забывают о необходимости инициализации этой структуры. Если Вы не обнулите ее элементы, в них будет содержаться мусор, оставшийся в стеке вызывающего потока. Функция `CreateProcess`, получив такую структуру данных, либо создаст новый процесс, либо нет — все зависит от того, что именно окажется в этом мусоре.

Когда Вам понадобится изменить какие-то элементы структуры, делайте это перед вызовом `CreateProcess`. Все элементы этой структуры подробно рассматриваются в таблице 4-3. Но заметьте, что некоторые элементы имеют смысл, только если до-чернее приложение создает перекрываемое (overlapped) окно, а другие — если это приложение осуществляет ввод-вывод на консоль.

Элемент	Окно или консоль	Описание
<i>cb</i>	То и другое	Содержит количество байтов, занимаемых структурой <code>STARTUPINFO</code> . Служит для контроля версий — на тот случай, если Microsoft расширит эту структуру в будущем. Программа должна инициализировать <i>cb</i> как <code>sizeof(STARTUPINFO)</code> .
<i>lpReserved</i>	То и другое	Зарезервирован. Инициализируйте как NULL.
<i>lpDesktop</i>	То и другое	Идентифицирует имя рабочего стола, на котором запускается приложение. Если указанный рабочий стол существует, новый процесс сразу же связывается с ним. В ином случае система сначала создает рабочий стол с атрибутами по умолчанию, присваивает ему имя, указанное в данном элементе структуры, и связывает его с новым процессом. Если <i>lpDesktop</i> равен NULL (что чаще всего и бывает), процесс связывается с текущим рабочим столом.
<i>lpTitle</i>	Консоль	Определяет заголовок консольного окна. Если <i>lpTitle</i> — NULL, в заголовок выводится имя исполняемого файла.

Таблица 4-3. Элементы структуры `STARTUPINFO`

продолжение

Элемент	Окно или консоль	Описание
<i>dwX</i> <i>dwY</i>	То и другое	Указывают <i>x</i> - и <i>y</i> -координаты (в пикселях) окна приложения. Эти координаты используются, только если дочерний процесс создает свое первое перекрываемое окно с идентификатором CW_USEDEFAULT в параметре <i>x</i> функции <i>CreateWindow</i> . В приложениях, создающих консольные окна, данные элементы определяют верхний левый угол консольного окна.
<i>dwXSize</i> <i>dwYSize</i>	То и другое	Определяют ширину и высоту (в пикселях) окна приложения. Эти значения используются, только если дочерний процесс создает свое первое перекрываемое окно с идентификатором CW_USEDEFAULT в параметре <i>nWidth</i> функции <i>CreateWindow</i> . В приложениях, создающих консольные окна, данные элементы определяют ширину и высоту консольного окна.
<i>dwXCountChars</i> <i>dwYCountChars</i>	Консоль	Определяют ширину и высоту (в символах) консольных окон дочернего процесса.
<i>dwFillAttribute</i>	Консоль	Задает цвет текста и фона в консольных окнах дочернего процесса.
<i>dwFlags</i>	То и другое	См. ниже и следующую таблицу.
<i>wShowWindow</i>	Окно	Определяет, как именно должно выглядеть первое перекрываемое окно дочернего процесса, если приложение при первом вызове функции <i>ShowWindow</i> передает в параметре <i>nCmdShow</i> идентификатор SW_SHOWDEFAULT. В этот элемент можно записать любой из идентификаторов типа SW_*, обычно используемых при вызове <i>ShowWindow</i> .
<i>cbReserved2</i>	То и другое	Зарезервирован. Инициализируйте как 0.
<i>lpReserved2</i>	То и другое	Зарезервирован. Инициализируйте как NULL.
<i>bStdInput</i> <i>bStdOutput</i> <i>bStdError</i>	Консоль	Определяют описатели буферов для консольного ввода-вывода. По умолчанию <i>bStdInput</i> идентифицирует буфер клавиатуры, а <i>bStdOutput</i> и <i>bStdError</i> — буфер консольного окна.

Теперь, как я и обещал, обсудим элемент *dwFlags*. Он содержит набор флагов, позволяющих управлять созданием дочернего процесса. Большая часть флагов просто сообщает функции *CreateProcess*, содержит ли прочие элементы структуры STARTUPINFO полезную информацию или некоторые из них можно игнорировать. Список допустимых флагов приведен в следующей таблице.

Флаг	Описание
STARTF_USESIZE	Заставляет использовать элементы <i>dwXSize</i> и <i>dwYSize</i>
STARTF_USESHOWWINDOW	Заставляет использовать элемент <i>wShowWindow</i>
STARTF_USEPOSITION	Заставляет использовать элементы <i>dwX</i> и <i>dwY</i>
STARTF_USECOUNTCHARS	Заставляет использовать элементы <i>dwXCountChars</i> и <i>dwYCountChars</i>
STARTF_USEFILLATTRIBUTE	Заставляет использовать элемент <i>dwFillAttribute</i>
STARTF_USESTDHANDLES	Заставляет использовать элементы <i>bStdInput</i> , <i>bStdOutput</i> и <i>bStdError</i>

см. след. стр.

продолжение

Флаг	Описание
STARTF_RUN_FULLSCREEN	Приводит к тому, что консольное приложение на компьютере с процессором типа <i>x86</i> запускается в полноэкранном режиме

Два дополнительных флага — STARTF_FORCEONFEEDBACK и STARTF_FORCEOFFFEEDBACK — позволяют контролировать форму курсора мыши в момент запуска нового процесса. Поскольку Windows поддерживает истинную вытесняющую многозадачность, можно запустить одно приложение и, пока оно инициализируется, поработать с другой программой. Для визуальной обратной связи с пользователем функция *CreateProcess* временно изменяет форму системного курсора мыши:



Курсор такой формы подсказывает: можно либо подождать чего-нибудь, что вот-вот случится, либо продолжить работу в системе. Если же Вы укажете флаг STARTF_FORCEOFFFEEDBACK, *CreateProcess* не станет добавлять «песочные часы» к стандартной стрелке.

Флаг STARTF_FORCEONFEEDBACK заставляет *CreateProcess* отслеживать инициализацию нового процесса и в зависимости от результата проверки изменять форму курсора. Когда функция *CreateProcess* вызывается с этим флагом, курсор преобразуется в «песочные часы». Если спустя две секунды от нового процесса не поступает GUI-вызов, она восстанавливает исходную форму курсора.

Если же в течение двух секунд процесс все же делает GUI-вызов, *CreateProcess* ждет, когда приложение откроет свое окно. Это должно произойти в течение пяти секунд после GUI-вызова. Если окно не появилось, *CreateProcess* восстанавливает курсор, а появилось — сохраняет его в виде «песочных часов» еще на пять секунд. Как только приложение вызовет функцию *GetMessage*, сообщая тем самым, что оно закончило инициализацию, *CreateProcess* немедленно сменит курсор на стандартный и прекратит мониторинг нового процесса.

В заключение раздела — несколько слов об элементе *wShowWindow* структуры STARTUPINFO. Этот элемент инициализируется значением, которое Вы передаете в *(w)WinMain* через ее последний параметр, *nCmdShow*. Он позволяет указать, в каком виде должно появиться главное окно Вашего приложения. В качестве значения используется один из идентификаторов, обычно передаваемых в *ShowWindow* (чаще всего SW_SHOWNORMAL или SW_SHOWMINNOACTIVE, но иногда и SW_SHOWDEFAULT).

После запуска программы из Explorer ее функция *(w)WinMain* вызывается с SW_SHOWNORMAL в параметре *nCmdShow*. Если же Вы создаете для нее ярлык, то можете указать в его свойствах, в каком виде должно появляться ее главное окно. На рис. 4-3 показано окно свойств для ярлыка Notepad. Обратите внимание на список Run, в котором выбирается начальное состояние окна Notepad.

Когда Вы активизируете этот ярлык из Explorer, последний создает и инициализирует структуру STARTUPINFO, а затем вызывает *CreateProcess*. Это приводит к запуску Notepad, а его функция *(w)WinMain* получает SW_SHOWMINNOACTIVE в параметре *nCmdShow*.

Таким образом, пользователь может легко выбирать, в каком окне запускать программу — нормальном, свернутом или развернутом.

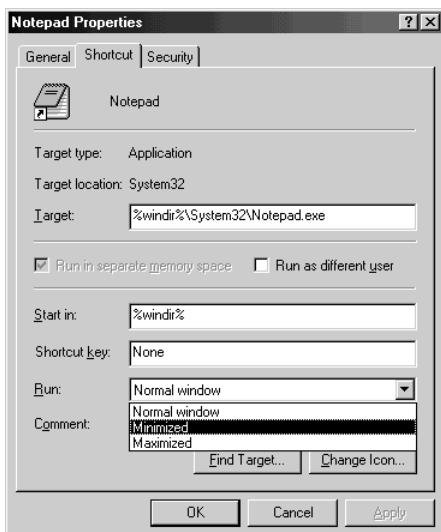


Рис. 4-3. Окно свойств для ярлыка Notepad

Наконец, чтобы получить копию структуры STARTUPINFO, инициализированной родительским процессом, приложение может вызвать:

```
VOID GetStartupInfo(PSTARTUPINFO pStartupInfo);
```

Анализируя эту структуру, дочерний процесс может изменять свое поведение в зависимости от значений ее элементов.



Хотя в документации на Windows об этом четко не сказано, перед вызовом *GetStartupInfo* нужно инициализировать элемент *cb* структуры STARTUPINFO:

```
STARTUPINFO si = { sizeof(si) };
GetStartupInfo(&si);
:
```

Параметр *ppiProcInfo*

Параметр *ppiProcInfo* указывает на структуру PROCESS_INFORMATION, которую Вы должны предварительно создать; ее элементы инициализируются самой функцией *CreateProcess*. Структура представляет собой следующее:

```
typedef struct _PROCESS_INFORMATION {
    HANDLE hProcess;
    HANDLE hThread;
    DWORD dwProcessId;
    DWORD dwThreadId;
} PROCESS_INFORMATION;
```

Как я уже говорил, создание нового процесса влечет за собой создание объектов ядра «процесс» и «поток». В момент создания система присваивает счетчику каждого объекта начальное значение — единицу. Далее функция *CreateProcess* (перед самым возвратом управления) открывает объекты «процесс» и «поток» и заносит их описатели, специфичные для данного процесса, в элементы *hProcess* и *hThread* структуры PROCESS_INFORMATION. Когда *CreateProcess* открывает эти объекты, счетчики каждого из них увеличиваются до 2.

Это означает, что, перед тем как система сможет высвободить из памяти объект «процесс», процесс должен быть завершен (счетчик уменьшен до 1), а родительский процесс обязан вызвать функцию *CloseHandle* (и тем самым обнулить счетчик). То же самое относится и к объекту «поток»: поток должен быть завершен, а родительский процесс должен закрыть описатель объекта «поток». Подробнее об освобождении объектов «поток» см. раздел «Дочерние процессы» в этой главе.



Не забывайте закрывать описатели дочернего процесса и его первичного потока, иначе, пока Вы не закроете свое приложение, будет происходить утечка ресурсов. Конечно, система высвободит все эти ресурсы после завершения Вашего процесса, но хорошо написанная программа должна сама закрывать описатели дочернего процесса и его первичного потока, как только необходимость в них отпадает. Пропуск этой операции — одна из самых частых ошибок.

Почему-то многие разработчики считают, будто закрытие описателя процесса или потока заставляет систему уничтожить этот процесс или поток. Это абсолютно неправильно. Закрывая описатель, Вы просто сообщаете системе, что статистические данные для этого процесса или потока Вас больше не интересуют, но процесс или поток продолжает исполняться системой до тех пор, пока он сам не завершит себя.

Созданному объекту ядра «процесс» присваивается уникальный идентификатор; ни у каких других объектов этого типа в системе не может быть одинаковых идентификаторов. Это же касается и объектов ядра «поток». Причем идентификаторы процесса и потока тоже разные, и их значения никогда не бывают нулевыми. Завершая свою работу, *CreateProcess* заносит значения идентификаторов в элементы *dwProcessId* и *dwThreadId* структуры *PROCESS_INFORMATION*. Эти идентификаторы просто облегчают определение процессов и потоков в системе; их используют, как правило, лишь специализированные утилиты вроде Task Manager.

Подчеркну еще один чрезвычайно важный момент: система способна повторно использовать идентификаторы процессов и потоков. Например, при создании процесса система формирует объект «процесс», присваивая ему идентификатор со значением, допустим, 122. Создавая новый объект «процесс», система уже не присвоит ему данный идентификатор. Но после выгрузки из памяти первого объекта следующему создаваемому объекту «процесс» может быть присвоен тот же идентификатор — 122.

Эту особенность нужно учитывать при написании кода, избегая ссылок на неверный объект «процесс» (или «поток»). Действительно, затребовать и сохранить идентификатор процесса несложно, но задумайтесь, что получится, если в следующий момент этот процесс будет завершен, а новый получит тот же идентификатор: сохраненный ранее идентификатор уже связан совсем с другим процессом.

Иногда программе приходится определять свой родительский процесс. Однако родственные связи между процессами существуют лишь на стадии создания дочернего процесса. Непосредственно перед началом исполнения кода в дочернем процессе Windows перестает учитывать его родственные связи. В предыдущих версиях Windows не было функций, которые позволяли бы программе обращаться с запросом к ее родительскому процессу. Но ToolHelp-функции, появившиеся в современных версиях Windows, сделали это возможным. С этой целью Вы должны использовать структуру *PROCESSENTRY32*: ее элемент *th32ParentProcessID* возвращает идентификатор «родителя» данного процесса. Тем не менее, если Вашей программе нужно взаимодей-

ствовать с родительским процессом, от идентификаторов лучше отказаться. Почему — я уже говорил. Для определения родительского процесса существуют более надежные механизмы: объекты ядра, описатели окон и т. д.

Единственный способ добиться того, чтобы идентификатор процесса или потока не использовался повторно, — не допускать разрушения объекта ядра «процесс» или «поток». Если Вы только что создали новый процесс или поток, то можете просто не закрывать описатели на эти объекты — вот и все. А по окончании операций с идентификатором, вызовите функцию *CloseHandle* и освободите соответствующие объекты ядра. Однако для дочернего процесса этот способ не годится, если только он не унаследовал описатели объектов ядра от родительского процесса.

Завершение процесса

Процесс можно завершить четырьмя способами:

- входная функция первичного потока возвращает управление (рекомендуемый способ);
- один из потоков процесса вызывает функцию *ExitProcess* (нежелательный способ);
- поток другого процесса вызывает функцию *TerminateProcess* (тоже нежелательно);
- все потоки процесса умирают по своей воле (большая редкость).

В этом разделе мы обсудим только что перечисленные способы завершения процесса, а также рассмотрим, что на самом деле происходит в момент его окончания.

Возврат управления входной функцией первичного потока

Приложение следует проектировать так, чтобы его процесс завершался только после возврата управления входной функцией первичного потока. Это единственный способ, гарантирующий корректную очистку всех ресурсов, принадлежавших первично-му потоку. При этом:

- любые C++-объекты, созданные данным потоком, уничтожаются соответствующими деструкторами;
- система освобождает память, которую занимал стек потока;
- система устанавливает код завершения процесса (поддерживаемый объектом ядра «процесс») — его и возвращает Ваша входная функция;
- счетчик пользователей данного объекта ядра «процесс» уменьшается на 1.

Функция *ExitProcess*

Процесс завершается, когда один из его потоков вызывает *ExitProcess*:

```
VOID ExitProcess(UINT fuExitCode);
```

Эта функция завершает процесс и заносит в параметр *fuExitCode* код завершения процесса. Возвращаемого значения у *ExitProcess* нет, так как результат ее действия — завершение процесса. Если за вызовом этой функции в программе присутствует какой-нибудь код, он никогда не исполняется.

Когда входная функция (*WinMain*, *wWinMain*, *main* или *wmain*) в Вашей программе возвращает управление, оно передается стартовому коду из библиотеки C/C++, и тот проводит очистку всех ресурсов, выделенных им процессу, а затем обращается к

ExitProcess, передавая ей значение, возвращенное входной функцией. Вот почему возврат управления входной функцией первичного потока приводит к завершению всего процесса. Обратите внимание, что при завершении процесса прекращается выполнение и всех других его потоков.

Кстати, в документации из Platform SDK утверждается, что процесс не завершается до тех пор, пока не завершится выполнение всех его потоков. Это, конечно, верно, но тут есть одна тонкость. Стартовый код из библиотеки C/C++ обеспечивает завершение процесса, вызывая *ExitProcess* после того, как первичный поток Вашего приложения возвращается из входной функции. Однако, вызвав из нее функцию *ExitThread* (вместо того чтобы вызвать *ExitProcess* или просто вернуть управление), Вы завершили первый поток, но не сам процесс — если в нем еще выполняется какой-то другой поток (или потоки).

Заметьте, что такой вызов *ExitProcess* или *ExitThread* приводит к уничтожению процесса или потока, когда выполнение функции еще не завершилось. Что касается операционной системы, то здесь все в порядке: она корректно очистит все ресурсы, выделенные процессу или потоку. Но в приложении, написанном на C/C++, следует избегать вызова этих функций, так как библиотеке C/C++ скорее всего не удастся провести должную очистку. Взгляните на этот код:

```
#include <windows.h>
#include <stdio.h>

class CSomeObj {
public:
    CSomeObj() { printf("Constructor\r\n"); }
    ~CSomeObj() { printf("Destructor\r\n"); }
};

CSomeObj g_GlobalObj;

void main () {
    CSomeObj LocalObj;
    ExitProcess(0);           // этого здесь не должно быть

    // в конце этой функции компилятор автоматически вставил код
    // для вызова деструктора LocalObj, но ExitProcess не дает его выполнить
}
```

При его выполнении Вы увидите:

```
Constructor
Constructor
```

Код конструирует два объекта: глобальный и локальный. Но Вы никогда не увидите строку *Destructor*. C++-объекты не разрушаются должным образом из-за того, что *ExitProcess* форсирует уничтожение процесса и библиотека C/C++ не получает шанса на очистку.

Как я уже говорил, никогда не вызывайте *ExitProcess* в явном виде. Если я уберу из предыдущего примера вызов *ExitProcess*, программа выведет такие строки:

```
Constructor
Constructor
Destructor
Destructor
```

Простой возврат управления от входной функции первичного потока позволил библиотеке C/C++ провести нужную очистку и корректно разрушить C++-объекты. Кстати, все, о чем я рассказал, относится не только к объектам, но и ко многим другим вещам, которые библиотека C/C++ делает для Вашего процесса.



Явные вызовы ExitProcess и ExitThread — распространенная ошибка, которая мешает правильной очистке ресурсов. В случае ExitThread процесс продолжает работать, но при этом весьма вероятна утечка памяти или других ресурсов.

Функция *TerminateProcess*

Вызов функции *TerminateProcess* тоже завершает процесс:

```
BOOL TerminateProcess(
    HANDLE hProcess,
    UINT fuExitCode);
```

Главное отличие этой функции от *ExitProcess* в том, что ее может вызвать любой поток и завершить любой процесс. Параметр *hProcess* идентифицирует описатель завершаемого процесса, а в параметре *fuExitCode* возвращается код завершения процесса.

Пользуйтесь *TerminateProcess* лишь в том случае, когда иным способом завершить процесс не удается. Процесс не получает абсолютно никаких уведомлений о том, что он завершается, и приложение не может ни выполнить очистку, ни предотвратить свое неожиданное завершение (если оно, конечно, не использует механизмы защиты). При этом теряются все данные, которые процесс не успел переписать из памяти на диск.

Процесс действительно не имеет ни малейшего шанса самому провести очистку, но операционная система высвобождает все принадлежавшие ему ресурсы: возвращает себе выделенную им память, закрывает любые открытые файлы, уменьшает счетчики соответствующих объектов ядра и разрушает все его User- и GDI-объекты.

По завершении процесса (не важно каким способом) система гарантирует: после него ничего не останется — даже намеков на то, что он когда-то выполнялся. *Завершенный процесс не оставляет за собой никаких следов*. Надеюсь, я сказал ясно.



TerminateProcess — функция асинхронная, т. е. она сообщает системе, что Вы хотите завершить процесс, но к тому времени, когда она вернет управление, процесс может быть еще не уничтожен. Так что, если Вам нужно точно знать момент завершения процесса, используйте *WaitForSingleObject* (см. главу 9) или аналогичную функцию, передав ей описатель этого процесса.

Когда все потоки процесса уходят

В такой ситуации (а она может возникнуть, если все потоки вызвали *ExitThread* или их закрыли вызовом *TerminateThread*) операционная система больше не считает нужным «содержать» адресное пространство данного процесса. Обнаружив, что в процессе не исполняется ни один поток, она немедленно завершает его. При этом код завершения процесса приравнивается коду завершения последнего потока.

Что происходит при завершении процесса

А происходит вот что:

1. Выполнение всех потоков в процессе прекращается.
2. Все User- и GDI-объекты, созданные процессом, уничтожаются, а объекты ядра закрываются (если их не использует другой процесс).
3. Код завершения процесса меняется со значения `STILL_ACTIVE` на код, переданный в `ExitProcess` или `TerminateProcess`.
4. Объект ядра «процесс» переходит в свободное, или незанятое (`signaled`), состояние. (Подробнее на эту тему см. главу 9.) Прочие потоки в системе могут приступить к выполнению вплоть до завершения данного процесса.
5. Счетчик объекта ядра «процесс» уменьшается на 1.

Связанный с завершающим процессом объект ядра не высвобождается, пока не будут закрыты ссылки на него и из других процессов. В момент завершения процесса система автоматически уменьшает счетчик пользователей этого объекта на 1, и объект разрушается, как только его счетчик обнуляется. Кроме того, закрытие процесса не приводит к автоматическому завершению порожденных им процессов.

По завершении процесса его код и выделенные ему ресурсы удаляются из памяти. Однако область памяти, выделенная системой для объекта ядра «процесс», не освобождается, пока счетчик числа его пользователей не достигнет нуля. А это произойдет, когда все прочие процессы, создавшие или открывшие описатели для ныне-покойного процесса, уведомят систему (вызовом `CloseHandle`) о том, что ссылки на этот процесс им больше не нужны.

Описатели завершенного процесса уже мало на что пригодны. Разве что родительский процесс, вызвав функцию `GetExitCodeProcess`, может проверить, завершен ли процесс, идентифицируемый параметром `hProcess`, и, если да, определить код завершения:

```
BOOL GetExitCodeProcess(  
    HANDLE hProcess,  
    PDWORD pdwExitCode);
```

Код завершения возвращается как значение типа `DWORD`, на которое указывает `pdwExitCode`. Если на момент вызова `GetExitCodeProcess` процесс еще не завершился, в `DWORD` заносится идентификатор `STILL_ACTIVE` (определенный как `0x103`). А если он уничтожен, функция возвращает реальный код его завершения.

Вероятно, Вы подумали, что можно написать код, который, периодически вызывая функцию `GetExitCodeProcess` и проверяя возвращаемое ею значение, определял бы момент завершения процесса. В принципе такой код мог бы сработать во многих ситуациях, но он был бы неэффективен. Как правильно решить эту задачу, я расскажу в следующем разделе.

Дочерние процессы

При разработке приложения часто бывает нужно, чтобы какую-то операцию выполнил другой блок кода. Поэтому — хочешь, не хочешь — приходится постоянно вызывать функции или подпрограммы. Но вызов функции приводит к приостановке выполнения основного кода Вашей программы до возврата из вызванной функции. Альтернативный способ — передать выполнение какой-то операции другому потоку в пределах данного процесса (поток, разумеется, нужно сначала создать). Это позво-

лит основному коду программы продолжить работу в то время, как дополнительный поток будет выполнять другую операцию. Прием весьма удобный, но, когда основному потоку потребуется узнать результаты работы другого потока, Вам не избежать проблем, связанных с синхронизацией.

Есть еще один прием: Ваш процесс порождает дочерний и возлагает на него выполнение части операций. Будем считать, что эти операции очень сложны. Допустим, для их реализации Вы просто создаете новый поток внутри того же процесса. Вы пишете тот или иной код, тестируете его и получаете некорректный результат — может, ошиблись в алгоритме или запутались в ссылках и случайно перезаписали какие-нибудь важные данные в адресном пространстве своего процесса. Так вот, один из способов защитить адресное пространство основного процесса от подобных ошибок как раз и состоит в том, чтобы передать часть работы отдельному процессу. Далее можно или подождать, пока он завершится, или продолжить работу параллельно с ним.

К сожалению, дочернему процессу, по-видимому, придется оперировать с данными, содержащимися в адресном пространстве родительского процесса. Было бы не плохо, чтобы он работал исключительно в своем адресном пространстве, а в «Вашем» — просто считывал нужные ему данные; тогда он не сможет что-то испортить в адресном пространстве родительского процесса. В Windows предусмотрено несколько способов обмена данными между процессами: DDE (Dynamic Data Exchange), OLE, каналы (pipes), почтовые ящики (mailslots) и т. д. А один из самых удобных способов, обеспечивающих совместный доступ к данным, — использование файлов, проецируемых в память (*memory-mapped files*). (Подробнее на эту тему см. главу 17.)

Если Вы хотите создать новый процесс, заставить его выполнить какие-либо операции и дождаться их результатов, напишите примерно такой код:

```
PROCESS_INFORMATION pi;
DWORD dwExitCode;

// порождаем дочерний процесс
BOOL fSuccess = CreateProcess(..., &pi);
if (fSuccess) {

    // закрываем описатель потока, как только необходимость в нем отпадает!
    CloseHandle(pi.hThread);

    // приостанавливаем выполнение родительского процесса,
    // пока не завершится дочерний процесс
    WaitForSingleObject(pi.hProcess, INFINITE);

    // дочерний процесс завершился; получаем код его завершения
    GetExitCodeProcess(pi.hProcess, &dwExitCode);

    // закрываем описатель процесса, как только необходимость в нем отпадает!
    CloseHandle(pi.hProcess);
}
```

В этом фрагменте кода мы создали новый процесс и, если это прошло успешно, вызвали функцию *WaitForSingleObject*:

```
DWORD WaitForSingleObject(HANDLE hObject, DWORD dwTimeOut);
```

Подробное рассмотрение данной функции мы отложим до главы 9, а сейчас ограничимся одним соображением. Функция задерживает выполнение кода до тех пор,

пока объект, определяемый параметром *bObject*, не перейдет в свободное (незанятое) состояние. Объект «процесс» переходит в такое состояние при его завершении. Поэтому вызов *WaitForSingleObject* приостанавливает выполнение потока родительского процесса до завершения порожденного им процесса. Когда *WaitForSingleObject* вернет управление, Вы узнаете код завершения дочернего процесса через функцию *GetExitCodeProcess*.

Обращение к *CloseHandle* в приведенном выше фрагменте кода заставляет систему уменьшить значения счетчиков объектов «поток» и «процесс» до нуля и тем самым освободить память, занимаемую этими объектами.

Вы, наверное, заметили, что в этом фрагменте я закрыл описатель объекта ядра «первичный поток» (принадлежащий дочернему процессу) сразу после возврата из *CreateProcess*. Это не приводит к завершению первичного потока дочернего процесса — просто уменьшает счетчик, связанный с упомянутым объектом. А вот почему это делается — и, кстати, даже рекомендуется делать — именно так, станет ясно из простого примера. Допустим, первичный поток дочернего процесса порождает еще один поток, а сам после этого завершается. В этот момент система может высвободить объект «первичный поток» дочернего процесса из памяти, если у родительского процесса нет описателя данного объекта. Но если родительский процесс располагает таким описателем, система не сможет удалить этот объект из памяти до тех пор, пока и родительский процесс не закроет его описатель.

Запуск обособленных дочерних процессов

Что ни говори, но чаще приложение все-таки создает другие процессы как *обособленные* (*detached processes*). Это значит, что после создания и запуска нового процесса родительскому процессу нет нужды с ним взаимодействовать или ждать, пока тот закончит работу. Именно так и действует Explorer: запускает для пользователя новые процессы, а дальше его уже не волнует, что там с ними происходит.

Чтобы обрубить все пуповины, связывающие Explorer с дочерним процессом, ему нужно (вызовом *CloseHandle*) закрыть свои описатели, связанные с новым процессом и его первичным потоком. Приведенный ниже фрагмент кода демонстрирует, как, создав процесс, сделать его обособленным:

```
PROCESS_INFORMATION pi;  
  
BOOL fSuccess = CreateProcess(..., &pi);  
if (fSuccess) {  
    // разрешаем системе уничтожить объекты ядра "процесс" и "поток"  
    // сразу после завершения дочернего процесса  
    CloseHandle(pi.hThread);  
    CloseHandle(pi.hProcess);  
}
```

Перечисление процессов, выполняемых в системе

Многие разработчики программного обеспечения пытаются создавать инструментальные средства или утилиты для Windows, требующие перечисления процессов, выполняемых в системе. Изначально в Windows API не было функций, которые позволяли бы это делать. Однако в Windows NT ведется постоянно обновляемая база данных Performance Data. В ней содержится чуть ли не тонна информации, доступной через функции реестра вроде *RegQueryValueEx*, для которой надо указать корне-

вой раздел HKEY_PERFORMANCE_DATA. Мало кто из программистов знает об этой базе данных, и причины тому кроются, на мой взгляд, в следующем.

- Для нее не предусмотрено никаких специфических функций; нужно использовать обычные функции реестра.
- Ее нет в Windows 95 и Windows 98.
- Структура информации в этой базе данных очень сложна; многие просто избегают ею пользоваться (и другим не советуют).

Чтобы упростить работу с этой базой данных, Microsoft создала набор функций под общим названием Performance Data Helper (содержащийся в PDH.dll). Если Вас интересует более подробная информация о библиотеке PDH.dll, ищите раздел по функциям Performance Data Helper в документации Platform SDK.

Как я уже упоминал, в Windows 95 и Windows 98 такой базы данных нет. Вместо них предусмотрен набор функций, позволяющих перечислять процессы. Они включены в ToolHelp API. За информацией о них я вновь отсылаю Вас к документации Platform SDK — ищите разделы по функциям *Process32First* и *Process32Next*.

Но самое смешное, что разработчики Windows NT, которым ToolHelp-функции явно не нравятся, не включили их в Windows NT. Для перечисления процессов они создали свой набор функций под общим названием Process Status (содержащийся в PSAPI.dll). Так что ищите в документации Platform SDK раздел по функции *Emit-Processes*.

Microsoft, которая до сих пор, похоже, старалась усложнить жизнь разработчикам инструментальных средств и утилит, все же включила ToolHelp-функции в Windows 2000. Наконец-то эти разработчики смогут унифицировать свой код хотя бы для Windows 95, Windows 98 и Windows 2000!

Программа-пример ProcessInfo

Эта программа, «04 ProcessInfo.exe» (см. листинг на рис. 4-6), демонстрирует, как создать очень полезную утилиту на основе ToolHelp-функций. Файлы исходного кода и ресурсов программы находятся в каталоге 04-ProcessInfo на компакт-диске, прилагаемом к книге. После запуска ProcessInfo открывается окно, показанное на рис. 4-4.

ProcessInfo сначала перечисляет все процессы, выполняемые в системе, а затем выводит в верхний раскрывающийся список имена и идентификаторы каждого процесса. Далее выбирается первый процесс и информация о нем показывается в большом текстовом поле, доступном только для чтения. Как видите, для текущего процесса сообщается его идентификатор (вместе с идентификатором родительского процесса), класс приоритета и количество потоков, выполняемых в настоящий момент в контексте процесса. Объяснение большей части этой информации выходит за рамки данной главы, но будет рассматриваться в последующих главах.

При просмотре списка процессов становится доступен элемент меню VMMap. (Он отключается, когда Вы переключаетесь на просмотр информации о модулях.) Выбрав элемент меню VMMap, Вы запускаете программу-пример VMMap (см. главу 14). Эта программа «проходит» по адресному пространству выбранного процесса.

В информацию о модулях входит список всех модулей (EXE- и DLL-файлов), спроецированных на адресное пространство текущего процесса. Фиксированным модулем (fixed module) считается тот, который был явно загружен при инициализации процесса. Для явно загруженных DLL показываются счетчики числа пользователей этих DLL. Во втором столбце выводится базовый адрес памяти, на который спроеци-

рован модуль. Если модуль размещён не по заданному для него базовому адресу, в скобках появляется и этот адрес. В третьем столбце сообщается размер модуля в байтах, а в последнем — полное (вместе с путём) имя файла этого модуля. И, наконец, внизу показывается информация о потоках, выполняемых в данный момент в контексте текущего процесса. При этом отображается идентификатор потока (thread ID, TID) и его приоритет.

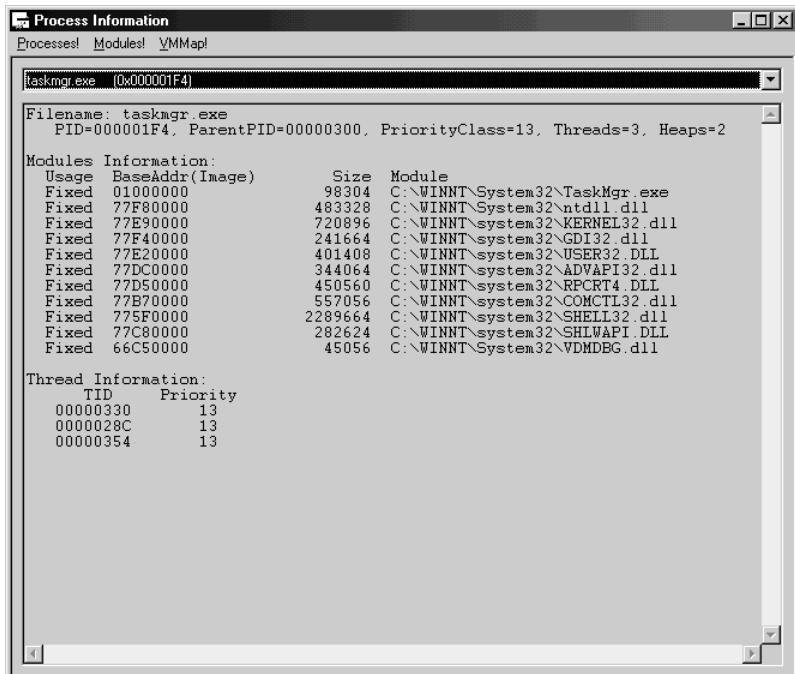


Рис. 4-4. *ProcessInfo* в действии

В дополнение к информации о процессах Вы можете выбрать элемент меню *Modules!*. Это заставит *ProcessInfo* перечислить все модули, загруженные в системе, и поместить их имена в верхний раскрывающийся список. Далее *ProcessInfo* выбирает первый модуль и выводит информацию о нем (рис. 4-5).

В этом режиме утилита *ProcessInfo* позволяет легко определить, в каких процессах задействован данный модуль. Как видите, полное имя модуля появляется в верхней части текстового поля, а в разделе *Process Information* перечисляются все процессы, содержащие этот модуль. Там же показываются идентификаторы и имена процессов, в которые загружен модуль, и его базовые адреса в этих процессах.

Всю эту информацию утилита *ProcessInfo* получает в основном от различных *ToolHelp*-функций. Чтобы чуточку упростить работу с *ToolHelp*-функциями, я создал C++-класс *CToolhelp* (содержащийся в файле *Toolhelp.h*). Он инкапсулирует все, что связано с получением «моментального снимка» состояния системы, и немного облегчает вызов других *ToolHelp*-функций.

Особый интерес представляет функция *GetModulePreferredBaseAddr* в файле *ProcessInfo.cpp*:

```
PVOID GetModulePreferredBaseAddr(
    DWORD dwProcessId,
    PVOID pvModuleRemote);
```

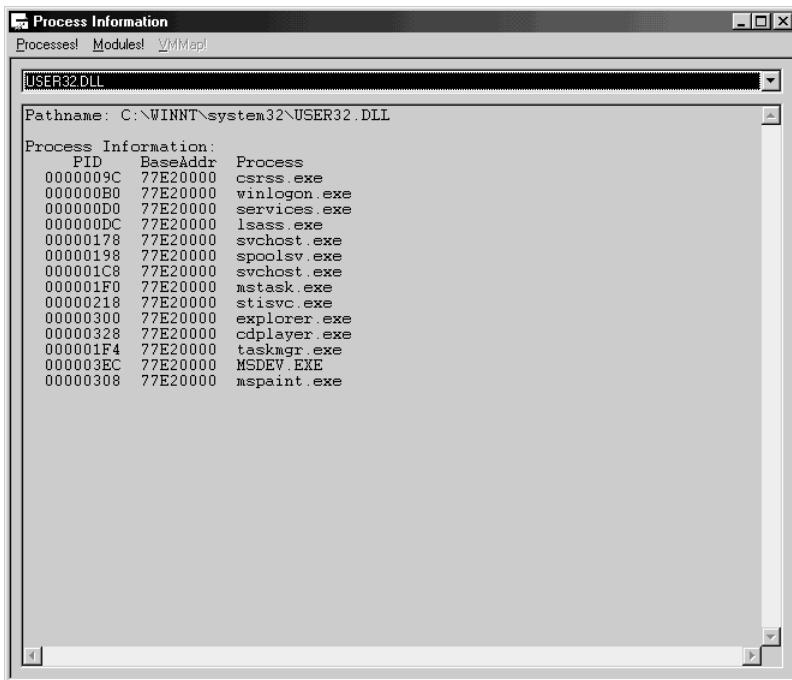


Рис. 4-5. *ProcessInfo* перечисляет все процессы, в адресные пространства которых загружен модуль *User32.dll*

Принимая идентификатор процесса и адрес модуля в этом процессе, она просматривает его адресное пространство, находит модуль и считывает информацию из заголовка модуля, чтобы определить, какой базовый адрес для него предпочтителен. Модуль должен всегда загружаться именно по этому адресу, а иначе приложения, использующие данный модуль, потребуют больше памяти и будут инициализироваться медленнее. Поскольку такая ситуация крайне нежелательна, моя утилита сообщает о случаях, когда модуль загружен не по предпочтительному базовому адресу. Впрочем, на эти темы мы поговорим в главе 20 (в разделе «Модификация базовых адресов модулей»).

ProcessInfo.cpp

```

/*
Модуль: ProcessInfo.cpp
Автор: Copyright (c) 2000, Джейфри Рихтер (Jeffrey Richter)
*/
#include "..\CmnHdr.h"      /* см. приложение A */
#include <windowsx.h>
#include <tlib32.h>
#include <tchar.h>
#include <stdarg.h>
#include <stdio.h>
#include "Toolhelp.h"
#include "Resource.h"

```

Рис. 4-6. Программа-пример *ProcessInfo*

см. след. стр.

Рис. 4-6. продолжение

```
//////////  
  
// добавляем строку в текстовое поле  
void AddText(HWND hwnd, PCTSTR pszFormat, ...) {  
  
    va_list argList;  
    va_start(argList, pszFormat);  
  
    TCHAR sz[20 * 1024];  
    Edit_GetText(hwnd, sz, chDIMOF(sz));  
    _vstprintf(_tcschr(sz, 0), pszFormat, argList);  
    Edit_SetText(hwnd, sz);  
    va_end(argList);  
}  
  
//////////  
  
VOID Dlg_PopulateProcessList(HWND hwnd) {  
  
    HWND hwndList = GetDlgItem(hwnd, IDC_PROCESSMODULELIST);  
    SetWindowRedraw(hwndList, FALSE);  
    ComboBox_ResetContent(hwndList);  
  
    CToolhelp thProcesses(TH32CS_SNAPPROCESS);  
    PROCESSENTRY32 pe = { sizeof(pe) };  
    BOOL fOk = thProcesses.ProcessFirst(&pe);  
    for (; fOk; fOk = thProcesses.ProcessNext(&pe)) {  
        TCHAR sz[1024];  
  
        // помещаем в список имя процесса (без пути) и идентификатор  
        PCTSTR pszExeFile = _tcsrchr(pe.szExeFile, TEXT('\\'));  
        if (pszExeFile == NULL) pszExeFile = pe.szExeFile;  
        else pszExeFile++; // пропускаем наклонную черту ("слэш")  
        wsprintf(sz, TEXT("%s (0x%08X)", pszExeFile, pe.th32ProcessID));  
        int n = ComboBox_AddString(hwndList, sz);  
  
        // сопоставляем идентификатор процесса с добавленным элементом  
        ComboBox_SetItemData(hwndList, n, pe.th32ProcessID);  
    }  
    ComboBox_SetCurSel(hwndList, 0); // выбираем первый элемент  
  
    // имитируем выбор пользователем первого элемента,  
    // чтобы в текстовом поле появилось что-нибудь интересное  
    FORWARD_WM_COMMAND(hwnd, IDC_PROCESSMODULELIST,  
        hwndList, CBN_SELCHANGE, SendMessage);  
  
    SetWindowRedraw(hwndList, TRUE);  
    InvalidateRect(hwndList, NULL, FALSE);  
}  
  
//////////
```

Рис. 4-6. продолжение

```

VOID Dlg_PopulateModuleList(HWND hwnd) {

    HWND hwndModuleHelp = GetDlgItem(hwnd, IDC_MODULEHELP);
    ListBox_ResetContent(hwndModuleHelp);
    CToolhelp thProcesses(TH32CS_SNAPPROCESS);
    PROCESSENTRY32 pe = { sizeof(pe) };
    BOOL fOk = thProcesses.ProcessFirst(&pe);
    for (; fOk; fOk = thProcesses.ProcessNext(&pe)) {

        CToolhelp thModules(TH32CS_SNAPMODULE, pe.th32ProcessID);
        MODULEENTRY32 me = { sizeof(me) };
        BOOL fOk = thModules.ModuleFirst(&me);
        for (; fOk; fOk = thModules.ModuleNext(&me)) {
            int n = ListBox_FindStringExact(hwndModuleHelp, -1, me.szExePath);
            if (n == LB_ERR) {
                // этот модуль еще не был добавлен
                ListBox_AddString(hwndModuleHelp, me.szExePath);
            }
        }
    }

    HWND hwndList = GetDlgItem(hwnd, IDC_PROCESSMODULELIST);
    SetWindowRedraw(hwndList, FALSE);
    ComboBox_ResetContent(hwndList);
    int nNumModules = ListBox_GetCount(hwndModuleHelp);
    for (int i = 0; i < nNumModules; i++) {
        TCHAR sz[1024];
        ListBox_GetText(hwndModuleHelp, i, sz);
        // помещаем в список имя модуля (без пути)
        int nIndex = ComboBox_AddString(hwndList, _tcsrchr(sz, TEXT('\\')) + 1);
        // сопоставляем индекс полного пути с добавленным элементом
        ComboBox_SetItemData(hwndList, nIndex, i);
    }

    ComboBox_SetCurSel(hwndList, 0); // выбираем первый элемент

    // имитируем выбор пользователем первого элемента,
    // чтобы в текстовом поле появилось что-нибудь интересное
    FORWARD_WM_COMMAND(hwnd, IDC_PROCESSMODULELIST,
        hwndList, CBN_SELCHANGE, SendMessage);

    SetWindowRedraw(hwndList, TRUE);
    InvalidateRect(hwndList, NULL, FALSE);
}

///////////////////////////////
PVVOID GetModulePreferredBaseAddr(DWORD dwProcessId, PVVOID pvModuleRemote) {

    PVVOID pvModulePreferredBaseAddr = NULL;
}

```

см. след. стр.

Рис. 4-6. продолжение

```
IMAGE_DOS_HEADER idh;
IMAGE_NT_HEADERS inth;

// считываем DOS-заголовок удаленного модуля
Toolhelp32ReadProcessMemory(dwProcessId,
    pvModuleRemote, &idh, sizeof(idh), NULL);

// проверяем DOS-заголовок его образа
if (idh.e_magic == IMAGE_DOS_SIGNATURE) {
    // считываем NT-заголовок удаленного модуля
    Toolhelp32ReadProcessMemory(dwProcessId,
        (PBYTE) pvModuleRemote + idh.e_lfanew, &inth, sizeof(inth), NULL);

    // проверяем NT-заголовок его образа
    if (inth.Signature == IMAGE_NT_SIGNATURE) {
        // NT-заголовок корректен,
        // получаем предпочтительный базовый адрес для данного образа
        pvModulePreferredBaseAddr = (PVOID) inth.OptionalHeader.ImageBase;
    }
}
return(pvModulePreferredBaseAddr);
}

///////////
VOID ShowProcessInfo(HWND hwnd, DWORD dwProcessID) {

    SetWindowText(hwnd, TEXT(""));
    // очищаем поле вывода

    CToolhelp th(TH32CS_SNAPALL, dwProcessID);

    // показываем подробную информацию о процессе
    PROCESSENTRY32 pe = { sizeof(pe) };
    BOOL fOk = th.ProcessFirst(&pe);
    for (; fOk; fOk = th.ProcessNext(&pe)) {
        if (pe.th32ProcessID == dwProcessID) {
            AddText(hwnd, TEXT("Filename: %s\r\n"), pe.szExeFile);
            AddText(hwnd, TEXT(" PID=%08X, ParentPID=%08X, ")
                TEXT("PriorityClass=%d, Threads=%d, Heaps=%d\r\n"),
                pe.th32ProcessID, pe.th32ParentProcessID,
                pe.pcPriClassBase, pe.cntThreads,
                th.HowManyHeaps());
            break; // продолжать цикл больше не нужно
        }
    }

    // показываем модули в процессе;
    // подсчитываем количество символов для вывода адреса
    const int cchAddress = sizeof(PVOID) * 2;
    AddText(hwnd, TEXT("\r\nModules Information:\r\n"))
        TEXT(" Usage %-s(%-*s) %8s Module\r\n"),
        cchAddress, TEXT("BaseAddr"));
}
```

Рис. 4-6. продолжение

```

cchAddress, TEXT("ImagAddr"), TEXT("Size"));

MODULEENTRY32 me = { sizeof(me) };
f0k = th.ModuleFirst(&me);

for (; f0k; f0k = th.ModuleNext(&me)) {
    if (me.ProccntUsage == 65535) {
        // модуль загружен неявно, и его нельзя выгрузить
        AddText(hwnd, TEXT(" Fixed"));
    } else {
        AddText(hwnd, TEXT("%5d"), me.ProccntUsage);
    }
    PVOID pvPreferredBaseAddr =
        GetModulePreferredBaseAddr(pe.th32ProcessID, me.modBaseAddr);
    if (me.modBaseAddr == pvPreferredBaseAddr) {
        AddText(hwnd, TEXT("%p %*s %8u %s\r\n"),
            me.modBaseAddr, cchAddress, TEXT(""),
            me.modBaseSize, me.szExePath);
    } else {
        AddText(hwnd, TEXT("%p(%p) %8u %s\r\n"),
            me.modBaseAddr, pvPreferredBaseAddr, me.modBaseSize, me.szExePath);
    }
}

// показываем потоки в процессе
AddText(hwnd, TEXT("\r\nThread Information:\r\n"))
    TEXT("      TID      Priority\r\n"));
THREADENTRY32 te = { sizeof(te) };
f0k = th.ThreadFirst(&te);
for (; f0k; f0k = th.ThreadNext(&te)) {
    if (te.th32OwnerProcessID == dwProcessID) {
        int nPriority = te.tpBasePri + te.tpDeltaPri;
        if ((te.tpBasePri < 16) && (nPriority > 15)) nPriority = 15;
        if ((te.tpBasePri > 15) && (nPriority > 31)) nPriority = 31;
        if ((te.tpBasePri < 16) && (nPriority < 1)) nPriority = 1;
        if ((te.tpBasePri > 15) && (nPriority < 16)) nPriority = 16;
        AddText(hwnd, TEXT("%08X      %2d\r\n"),
            te.th32ThreadID, nPriority);
    }
}
////////////////////////////////////////////////////////////////

VOID ShowModuleInfo(HWND hwnd, LPCTSTR pszModulePath) {

    SetWindowText(hwnd, TEXT("")); // очищаем поле вывода

    CToolhelp thProcesses(TH32CS_SNAPPROCESS);
    PROCESSENTRY32 pe = { sizeof(pe) };
    BOOL f0k = thProcesses.ProcessFirst(&pe);

```

см. след. стр.

Рис. 4-6. продолжение

```
AddText(hwnd, TEXT("Pathname: %s\r\nr\n"), pszModulePath);
AddText(hwnd, TEXT("Process Information:\r\n"));
AddText(hwnd, TEXT("      PID      BaseAddr Process\r\n"));
for ( ; f0k; f0k = thProcesses.ProcessNext(&pe)) {
    CToolhelp thModules(TH32CS_SNAPMODULE, pe.th32ProcessID);
    MODULEENTRY32 me = { sizeof(me) };
    BOOL f0k = thModules.ModuleFirst(&me);
    for ( ; f0k; f0k = thModules.ModuleNext(&me)) {
        if (_tcscmp(me.szExePath, pszModulePath) == 0) {
            AddText(hwnd, TEXT(" %08X %p %s\r\n"),
                    pe.th32ProcessID, me.modBaseAddr, pe.szExeFile);
        }
    }
}
////////////////////////////////////////////////////////////////

BOOL Dlg_OnInitDialog(HWND hwnd, HWND hwndFocus, LPARAM lParam) {
    chSETDLGICONS(hwnd, IDI_PROCESSINFO);

    // убираем окно списка модулей
    ShowWindow(GetDlgItem(hwnd, IDC_MODULEHELP), SW_HIDE);

    // пусть в окне результатов используется фиксированный шрифт
    SetWindowFont(GetDlgItem(hwnd, IDC_RESULTS),
                  GetStockFont(ANSI_FIXED_FONT), FALSE);

    // по умолчанию показываем список выполняемых процессов
    Dlg_PopulateProcessList(hwnd);

    return(TRUE);
}
////////////////////////////////////////////////////////////////

BOOL Dlg_OnSize(HWND hwnd, UINT state, int cx, int cy) {
    RECT rc;
    int n = LOWORD(GetDialogBaseUnits());

    HWND hwndCtl = GetDlgItem(hwnd, IDC_PROCESSMODULELIST);
    GetClientRect(hwndCtl, &rc);
    SetWindowPos(hwndCtl, NULL, n, n, cx - n - n, rc.bottom, SWP_NOZORDER);

    hwndCtl = GetDlgItem(hwnd, IDC_RESULTS);
    SetWindowPos(hwndCtl, NULL, n, n + rc.bottom + n,
                 cx - n - n, cy - (n + rc.bottom + n) - n, SWP_NOZORDER);

    return(0);
}
```

Рис. 4-6. продолжение

```
//////////  

void Dlg_OnCommand(HWND hwnd, int id, HWND hwndCtl, UINT codeNotify) {  

    static BOOL s_fProcesses = TRUE;  

    switch (id) {  

        case IDCANCEL:  

            EndDialog(hwnd, id);  

            break;  

        case ID_PROCESSES:  

            s_fProcesses = TRUE;  

            EnableMenuItem(GetMenu(hwnd), ID_VMMAP, MF_BYCOMMAND | MF_ENABLED);  

            DrawMenuBar(hwnd);  

            Dlg_PopulateProcessList(hwnd);  

            break;  

        case ID_MODULES:  

            EnableMenuItem(GetMenu(hwnd), ID_VMMAP, MF_BYCOMMAND | MF_GRAYED);  

            DrawMenuBar(hwnd);  

            s_fProcesses = FALSE;  

            Dlg_PopulateModuleList(hwnd);  

            break;  

        case IDC_PROCESSMODULELIST:  

            if (codeNotify == CBN_SELCHANGE) {  

                DWORD dw = ComboBox_GetCurSel(hwndCtl);  

                if (s_fProcesses) {  

                    dw = (DWORD) ComboBox_GetItemData(hwndCtl, dw); // ID процесса  

                    ShowProcessInfo(GetDlgItem(hwnd, IDC_RESULTS), dw);  

                } else {  

                    // индекс в окне вспомогательного списка  

                    dw = (DWORD) ComboBox_GetItemData(hwndCtl, dw);  

                    TCHAR szModulePath[1024];  

                    ListBox_GetText(GetDlgItem(hwnd, IDC_MODULEHELP), dw, szModulePath);  

                    ShowModuleInfo(GetDlgItem(hwnd, IDC_RESULTS), szModulePath);  

                }  

            }  

            break;  

        case ID_VMMAP:  

            STARTUPINFO si = { sizeof(si) };  

            PROCESS_INFORMATION pi;  

            TCHAR szCmdLine[1024];  

            HWND hwndCB = GetDlgItem(hwnd, IDC_PROCESSMODULELIST);  

            DWORD dwProcessId = (DWORD)  

                ComboBox_GetItemData(hwndCB, ComboBox_GetCurSel(hwndCB));  

            wsprintf(szCmdLine, TEXT("\\"14 VMMAP\" %d"), dwProcessId);  

            BOOL fOk = CreateProcess(NULL, szCmdLine, NULL, NULL,  

                FALSE, 0, NULL, NULL, &si, &pi);  

    }  

}
```

см. след. стр.

Рис. 4-6. продолжение

```
if (fOk) {
    CloseHandle(pi.hProcess);
    CloseHandle(pi.hThread);
} else {
    chMB("Failed to execute VMMAP.EXE.");
}
break;
}

///////////////////////////////



INT_PTR WINAPI Dlg_Proc(HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam) {

switch (uMsg) {
    chHANDLE_DLGMMSG(hwnd, WM_INITDIALOG, Dlg_OnInitDialog);
    chHANDLE_DLGMMSG(hwnd, WM_SIZE,           Dlg_OnSize);
    chHANDLE_DLGMMSG(hwnd, WM_COMMAND,        Dlg_OnCommand);
}
return(FALSE);
}

///////////////////////////////



int WINAPI _tWinMain(HINSTANCE hinstExe, HINSTANCE, PTSTR pszCmdLine, int) {

CToolhelp::EnableDebugPrivilege(TRUE);
DialogBox(hinstExe, MAKEINTRESOURCE(IDD_PROCESSINFO), NULL, Dlg_Proc);
CToolhelp::EnableDebugPrivilege(FALSE);
return(0);
}

/////////////////////////////// Конец файла ////////////////////////
```

Toolhelp.h

```
*****
Модуль: Toolhelp.h
Автор: Copyright (c) 2000, Джейфри Рихтер (Jeffrey Richter)
*****



#include "..\CmnHdr.h"      /* см. приложение A */
#include <tlhelp32.h>
#include <tchar.h>

///////////////////////////////



class CToolhelp {
private:
    HANDLE m_hSnapshot;
```

Рис. 4-6. продолжение

```

public:
    CToolhelp(DWORD dwFlags = 0, DWORD dwProcessID = 0);
    ~CToolhelp();

    BOOL CreateSnapshot(DWORD dwFlags, DWORD dwProcessID = 0);

    BOOL ProcessFirst(PPROCESSENTRY32 ppe) const;
    BOOL ProcessNext(PPROCESSENTRY32 ppe) const;
    BOOL ProcessFind(DWORD dwProcessId, PPROCESSENTRY32 ppe) const;

    BOOL ModuleFirst(PMODULEENTRY32 pme) const;
    BOOL ModuleNext(PMODULEENTRY32 pme) const;
    BOOL ModuleFind(PVOID pvBaseAddr, PMODULEENTRY32 pme) const;
    BOOL ModuleFind(PTSTR pszModName, PMODULEENTRY32 pme) const;

    BOOL ThreadFirst(PTHREADENTRY32 pte) const;
    BOOL ThreadNext(PTHREADENTRY32 pte) const;

    BOOL HeapListFirst(PHEAPLIST32 phl) const;
    BOOL HeapListNext(PHEAPLIST32 phl) const;
    int HowManyHeaps() const;

    // Примечание: функции, оперирующие с блоками памяти в куче, не ссылаются
    // на "снимок", а просто каждый раз с самого начала просматривают кучу
    // процесса. Если исследуемый процесс изменит свою кучу, когда любая
    // из этих функций будет перечислять блоки в его куче, возможно вхождение
    // в бесконечный цикл.
    BOOL HeapFirst(PHEAPENTRY32 phe, DWORD dwProcessID,
        UINT_PTR dwHeapID) const;
    BOOL HeapNext(PHEAPENTRY32 phe) const;
    int HowManyBlocksInHeap(DWORD dwProcessID, DWORD dwHeapId) const;
    BOOL IsAHeap(HANDLE hProcess, PVOID pvBlock, PDWORD pdwFlags) const;

public:
    static BOOL EnableDebugPrivilege(BOOL fEnable = TRUE);
    static BOOL ReadProcessMemory(DWORD dwProcessID, LPCVOID pvBaseAddress,
        PVOID pvBuffer, DWORD cbRead, PDWORD pdwNumberOfBytesRead = NULL);
};

///////////////////////////////
inline CToolhelp::CToolhelp(DWORD dwFlags, DWORD dwProcessID) {
    m_hSnapshot = INVALID_HANDLE_VALUE;
    CreateSnapshot(dwFlags, dwProcessID);
}

///////////////////////////////
inline CToolhelp::~CToolhelp() {

```

см. след. стр.

Рис. 4-6. продолжение

```
if (m_hSnapshot != INVALID_HANDLE_VALUE)
    CloseHandle(m_hSnapshot);
}

///////////////////////////////



inline CToolhelp::CreateSnapshot(DWORD dwFlags, DWORD dwProcessID) {

    if (m_hSnapshot != INVALID_HANDLE_VALUE)
        CloseHandle(m_hSnapshot);

    if (dwFlags == 0) {
        m_hSnapshot = INVALID_HANDLE_VALUE;
    } else {
        m_hSnapshot = CreateToolhelp32Snapshot(dwFlags, dwProcessID);
    }
    return(m_hSnapshot != INVALID_HANDLE_VALUE);
}

///////////////////////////////



inline BOOL CToolhelp::EnableDebugPrivilege(BOOL fEnable) {

    // передавая приложению полномочия отладчика, мы разрешаем ему
    // видеть информацию о сервисных приложениях
    BOOL fOk = FALSE; // предполагаем худшее
    HANDLE hToken;

    // пытаемся открыть маркер доступа (access token) для этого процесса
    if (OpenProcessToken(GetCurrentProcess(), TOKEN_ADJUST_PRIVILEGES,
        &hToken)) {

        TOKEN_PRIVILEGES tp;
        tp.PrivilegeCount = 1;
        LookupPrivilegeValue(NULL, SE_DEBUG_NAME, &tp.Privileges[0].Luid);
        tp.Privileges[0].Attributes = fEnable ? SE_PRIVILEGE_ENABLED : 0;
        AdjustTokenPrivileges(hToken, FALSE, &tp, sizeof(tp), NULL, NULL);
        fOk = (GetLastError() == ERROR_SUCCESS);
        CloseHandle(hToken);
    }
    return(fOk);
}

///////////////////////////////



inline BOOL CToolhelp::ReadProcessMemory(DWORD dwProcessID,
    LPCVOID pvBaseAddress, PVOID pvBuffer, DWORD cbRead,
    PDWORD pdwNumberOfBytesRead) {

    return(Toolhelp32ReadProcessMemory(dwProcessID, pvBaseAddress, pvBuffer,
        cbRead, pdwNumberOfBytesRead));
}
```

Рис. 4-6. продолжение

```
//////////  

inline BOOL CToolhelp::ProcessFirst(PPROCESSENTRY32 ppe) const {  

    BOOL f0k = Process32First(m_hSnapshot, ppe);  

    if (f0k && (ppe->th32ProcessID == 0))  

        f0k = ProcessNext(ppe); // удаляем "[System Process]" (PID = 0)  

    return(f0k);  

}  

inline BOOL CToolhelp::ProcessNext(PPROCESSENTRY32 ppe) const {  

    BOOL f0k = Process32Next(m_hSnapshot, ppe);  

    if (f0k && (ppe->th32ProcessID == 0))  

        f0k = ProcessNext(ppe); // удаляем "[System Process]" (PID = 0)  

    return(f0k);  

}  

inline BOOL CToolhelp::ProcessFind(DWORD dwProcessId, PPROCESSENTRY32 ppe) const {  

    BOOL fFound = FALSE;  

    for (BOOL f0k = ProcessFirst(ppe); f0k; f0k = ProcessNext(ppe)) {  

        fFound = (ppe->th32ProcessID == dwProcessId);  

        if (fFound) break;  

    }  

    return(fFound);  

}  

//////////  

inline BOOL CToolhelp::ModuleFirst(PMODULEENTRY32 pme) const {  

    return(Module32First(m_hSnapshot, pme));  

}  

inline BOOL CToolhelp::ModuleNext(PMODULEENTRY32 pme) const {  

    return(Module32Next(m_hSnapshot, pme));  

}  

inline BOOL CToolhelp::ModuleFind(PVOID pvBaseAddr, PMODULEENTRY32 pme) const {  

    BOOL fFound = FALSE;  

    for (BOOL f0k = ModuleFirst(pme); f0k; f0k = ModuleNext(pme)) {  

        fFound = (pme->modBaseAddr == pvBaseAddr);  

        if (fFound) break;  

    }  

    return(fFound);  

}
```

см. след. стр.

Рис. 4-6. продолжение

```
inline BOOL CToolhelp::ModuleFind(PTSTR pszModName, PMODULEENTRY32 pme) const {
    BOOL fFound = FALSE;
    for (BOOL f0k = ModuleFirst(pme); f0k; f0k = ModuleNext(pme)) {
        fFound = (lstrcmpi(pme->szModule, pszModName) == 0) ||
            (lstrcmpi(pme->szExePath, pszModName) == 0);
        if (fFound) break;
    }
    return(fFound);
}

///////////////////////////////
inline BOOL CToolhelp::ThreadFirst(PTHREADENTRY32 pte) const {
    return(Thread32First(m_hSnapshot, pte));
}

inline BOOL CToolhelp::ThreadNext(PTHREADENTRY32 pte) const {
    return(Thread32Next(m_hSnapshot, pte));
}

///////////////////////////////
inline int CToolhelp::HowManyHeaps() const {

    int nHowManyHeaps = 0;
    HEAPLIST32 hl = { sizeof(hl) };
    for (BOOL f0k = HeapListFirst(&hl); f0k; f0k = HeapListNext(&hl))
        nHowManyHeaps++;
    return(nHowManyHeaps);
}

inline int CToolhelp::HowManyBlocksInHeap(DWORD dwProcessID,
    DWORD dwHeapID) const {

    int nHowManyBlocksInHeap = 0;
    HEAPENTRY32 he = { sizeof(he) };
    BOOL f0k = HeapFirst(&he, dwProcessID, dwHeapID);
    for (; f0k; f0k = HeapNext(&he))
        nHowManyBlocksInHeap++;
    return(nHowManyBlocksInHeap);
}

inline BOOL CToolhelp::HeapListFirst(PHEAPLIST32 phl) const {

    return(Heap32ListFirst(m_hSnapshot, phl));
}

inline BOOL CToolhelp::HeapListNext(PHEAPLIST32 phl) const {
```

Рис. 4-6. продолжение

```
    return(Heap32ListNext(m_hSnapshot, ph1));  
}  
  
inline BOOL CToolhelp::HeapFirst(PHEAPENTRY32 phe, DWORD dwProcessID,  
    UINT_PTR dwHeapID) const {  
  
    return(Heap32First(phe, dwProcessID, dwHeapID));  
}  
  
inline BOOL CToolhelp::HeapNext(PHEAPENTRY32 phe) const {  
  
    return(Heap32Next(phe));  
}  
  
inline BOOL CToolhelp::IsAHeap(HANDLE hProcess, PVOID pvBlock,  
    PDWORD pdwFlags) const {  
  
    HEAPLIST32 hl = { sizeof(hl) };  
    for (BOOL f0kHL = HeapListFirst(&hl); f0kHL; f0kHL = HeapListNext(&hl)) {  
        HEAPENTRY32 he = { sizeof(he) };  
        BOOL f0kHE = HeapFirst(&he, hl.th32ProcessID, hl.th32HeapID);  
        for (; f0kHE; f0kHE = HeapNext(&he)) {  
            MEMORY_BASIC_INFORMATION mbi;  
            VirtualQueryEx(hProcess, (PVOID) he.dwAddress, &mbi, sizeof(mbi));  
            if (chINRANGE(mbi.AllocationBase, pvBlock,  
                (PBYTE) mbi.AllocationBase + mbi.RegionSize)) {  
  
                *pdwFlags = hl.dwFlags;  
                return(TRUE);  
            }  
        }  
    }  
    return(FALSE);  
}  
  
/////////////////////////////// Конец файла ///////////////////////////////
```

Задания

Группу процессов зачастую нужно рассматривать как единую сущность. Например, когда Вы командуете Microsoft Developer Studio собрать проект, он порождает процесс Cl.exe, а тот в свою очередь может создать другие процессы (скажем, для дополнительных проходов компилятора). Но, если Вы пожелаете прервать сборку, Developer Studio должен каким-то образом завершить Cl.exe и все его дочерние процессы. Решение этой простой (и распространенной) проблемы в Windows было весьма затруднительно, поскольку она не отслеживает родственные связи между процессами. В частности, выполнение дочерних процессов продолжается даже после завершения родительского.

При разработке сервера тоже бывает полезно группировать процессы. Допустим, клиентская программа просит сервер выполнить приложение (которое создает ряд дочерних процессов) и сообщить результаты. Поскольку к серверу может обратиться сразу несколько клиентов, было бы неплохо, если бы он умел как-то ограничивать ресурсы, выделяемые каждому клиенту, и тем самым не давал бы одному клиенту монопольно использовать все серверные ресурсы. Под ограничения могли бы подпадать такие ресурсы, как процессорное время, выделяемое на обработку клиентского запроса, и размеры рабочего набора (working set). Кроме того, у клиентской программы не должно быть возможности завершить работу сервера и т. д.

В Windows 2000 введен новый объект ядра — задание (job). Он позволяет группировать процессы и помещать их в нечто вроде песочницы, которая определенным образом ограничивает их действия. Относитесь к этому объекту как к контейнеру процессов. Кстати, очень полезно создавать задание и с одним процессом — это позволяет налагать на процесс ограничения, которые иначе указать нельзя.

Взгляните на мою функцию *StartRestrictedProcess* (рис. 5-1). Она включает процесс в задание, которое ограничивает возможность выполнения определенных операций.

WINDOWS 98 Windows 98 не поддерживает задания.

```
void StartRestrictedProcess() {
    // создаем объект ядра "задание"
    HANDLE hjob = CreateJobObject(NULL, NULL);

    // вводим ограничения для процессов в задании

    // сначала определяем некоторые базовые ограничения
    JOBOBJECT_BASIC_LIMIT_INFORMATION jobli = { 0 };
```

Рис. 5-1. Функция *StartRestrictedProcess*

Рис. 5-1. продолжение

```

// процесс всегда выполняется с классом приоритета idle
jobli.PriorityClass = IDLE_PRIORITY_CLASS;

// задание не может использовать более одной секунды процессорного времени
jobli.PerJobUserTimeLimit.QuadPart = 10000000; // 1 секунда, выраженная в
                                                // 100-наносекундных интервалах

// два ограничения, которые я налагаю на задание (процесс)
jobli.LimitFlags = JOB_OBJECT_LIMIT_PRIORITY_CLASS
    | JOB_OBJECT_LIMIT_JOB_TIME;
SetInformationJobObject(hjob, JobObjectBasicLimitInformation, &jobli,
    sizeof(jobli));

// теперь вводим некоторые ограничения по пользовательскому интерфейсу
JOBOBJECT_BASIC_UI_RESTRICTIONS jobuir;
jobuir.UIRestrictionsClass = JOB_OBJECT_UILIMIT_NONE; // "замысловатый" нуль

// процесс не имеет права останавливать систему
jobuir.UIRestrictionsClass |= JOB_OBJECT_UILIMIT_EXITWINDOWS;

// процесс не имеет права обращаться к USER-объектам в системе
// (например, к другим окнам)
jobuir.UIRestrictionsClass |= JOB_OBJECT_UILIMIT_HANDLES;

SetInformationJobObject(hjob, JobObjectBasicUIRestrictions, &jobuir,
    sizeof(jobuir));

// Порождаем процесс, который будет размещен в задании.
// ПРИМЕЧАНИЕ: процесс нужно сначала создать и только потом поместить
// в задание. А это значит, что поток процесса должен быть создан
// и тут же приостановлен, чтобы он не смог выполнить какой-нибудь код
// еще до введения ограничений.
STARTUPINFO si = { sizeof(si) };
PROCESS_INFORMATION pi;
CreateProcess(NULL, "CMD", NULL, NULL, FALSE,
    CREATE_SUSPENDED, NULL, NULL, &si, &pi);
// Включаем процесс в задание.
// ПРИМЕЧАНИЕ: дочерние процессы, порождаемые этим процессом,
// автоматически становятся частью того же задания.
AssignProcessToJobObject(hjob, pi.hProcess);

// теперь потоки дочерних процессов могут выполнять код
ResumeThread(pi.hThread);
CloseHandle(pi.hThread);

// ждем, когда процесс завершится или будет исчерпан
// лимит процессорного времени, указанный для задания
HANDLE h[2];
h[0] = pi.hProcess;
h[1] = hjob;

```

см. след. стр.

Рис. 5-1. продолжение

```
DWORD dw = WaitForMultipleObjects(2, h, FALSE, INFINITE);
switch (dw - WAIT_OBJECT_0) {
    case 0:
        // процесс завершился...
        break;
    case 1:
        // лимит процессорного времени исчерпан...
        break;
}
// проводим очистку
CloseHandle(pi.hProcess);
CloseHandle(hjob);
```

А теперь я объясню, как работает *StartRestrictedProcess*. Сначала я создаю новый объект ядра «задание», вызывая:

```
HANDLE CreateJobObject(
    PSECURITY_ATTRIBUTES psa,
    PCTSTR pszName);
```

Как и любая функция, создающая объекты ядра, *CreateJobObject* принимает в первом параметре информацию о защите и сообщает системе, должна ли она вернуть наследуемый описатель. Параметр *pszName* позволяет присвоить заданию имя, чтобы к нему могли обращаться другие процессы через функцию *OpenJobObject*.

```
HANDLE OpenJobObject(
    DWORD dwDesiredAccess,
    BOOL bInheritHandle,
    PCTSTR pszName);
```

Закончив работу с объектом-заданием, закройте его описатель, вызвав, как всегда, *CloseHandle*. Именно так я и делаю в конце своей функции *StartRestrictedProcess*. Имейте в виду, что закрытие объекта-задания не приводит к автоматическому завершению всех его процессов. На самом деле этот объект просто помечается как подлежащий разрушению, и система уничтожает его только после завершения всех включенных в него процессов.

Заметьте, что после закрытия описателя объект-задание становится недоступным для процессов, даже несмотря на то что объект все еще существует. Этот факт иллюстрирует следующий код:

```
// создаем именованный объект-задание
HANDLE hjob = CreateJobObject(NULL, TEXT("Jeff"));

// включаем в него наш процесс
AssignProcessToJobObject(hjob, GetCurrentProcess());

// закрытие объекта-задания не убивает ни наш процесс, ни само задание,
// но присвоенное ему имя ("Jeff") моментально удаляется
CloseHandle(hjob);

// пробуем открыть существующее задание
hjob = OpenJobObject(JOB_OBJECT_ALL_ACCESS, FALSE, TEXT("Jeff"));
// OpenJobObject терпит неудачу и возвращает NULL, поскольку имя ("Jeff")
```

```
// уже не указывает на объект-задание после вызова CloseHandle;
// получить описатель этого объекта больше нельзя
```

Определение ограничений, налагаемых на процессы в задании

Создав задание, Вы обычно строите «песочницу» (набор ограничений) для включаемых в него процессов. Ограничения бывают нескольких видов:

- базовые и расширенные базовые ограничения — не дают процессам в задании монопольно захватывать системные ресурсы;
- базовые ограничения по пользовательскому интерфейсу (UI) — блокируют возможность его изменения;
- ограничения, связанные с защитой, — перекрывают процессам в задании доступ к защищенным ресурсам (файлам, подразделам реестра и т. д.).

Ограничения на задание вводятся вызовом:

```
BOOL SetInformationJobObject(
    HANDLE hJob,
    JOBOBJECTINFOCLASS JobObjectInformationClass,
    PVOID pJobObjectInformation,
    DWORD cbJobObjectInformationLength);
```

Первый параметр определяет нужное Вам задание, второй параметр (перечисленного типа) — вид ограничений, третий — адрес структуры данных, содержащей подробную информацию о задаваемых ограничениях, а четвертый — размер этой структуры (используется для указания версии). Следующая таблица показывает, как устанавливаются ограничения.

Вид ограничений	Значение второго параметра	Структура, указываемая в третьем параметре
Базовые ограничения	<i>JobObjectBasicLimitInformation</i>	<code>JOBOBJECT_BASIC_LIMIT_INFORMATION</code>
Расширенные базовые ограничения	<i>JobObjectExtendedLimitInformation</i>	<code>JOBOBJECT_EXTENDED_LIMIT_INFORMATION</code>
Базовые ограничения по пользовательскому интерфейсу	<i>JobObjectBasicUIRestrictions</i>	<code>JOBOBJECT_BASIC_UI_RESTRICTIONS</code>
Ограничения, связанные с защитой	<i>JobObjectSecurityLimitInformation</i>	<code>JOBOBJECT_SECURITY_LIMIT_INFORMATION</code>

В функции *StartRestrictedProcess* я устанавливаю для задания лишь несколько базовых ограничений. Для этого я создаю структуру `JOB_OBJECT_BASIC_LIMIT_INFORMATION`, инициализирую ее и вызываю функцию *SetInformationJobObject*. Данная структура выглядит так:

```
typedef struct _JOBOBJECT_BASIC_LIMIT_INFORMATION {
    LARGE_INTEGER PerProcessUserTimeLimit;
    LARGE_INTEGER PerJobUserTimeLimit;
    DWORD        LimitFlags;
    DWORD        MinimumWorkingSetSize;
```

см. след. стр.

```
    DWORD      MaximumWorkingSetSize;
    DWORD      ActiveProcessLimit;
    DWORD_PTR   Affinity;
    DWORD      PriorityClass;
    DWORD      SchedulingClass;
} JOBOBJECT_BASIC_LIMIT_INFORMATION, *PJOBOBJECT_BASIC_LIMIT_INFORMATION;
```

Все элементы этой структуры кратко описаны в таблице 5-1.

Элементы	Описание	Примечание
<i>PerProcessUser-TimeLimit</i>	Максимальное время в пользовательском режиме, выделяемое каждому процессу (в порциях по 100 нс)	Система автоматически завершает любой процесс, который пытается использовать больше отведенного времени. Это ограничение вводится флагом <i>JOB_OBJECT_LIMIT_PROCESS_TIME</i> в <i>LimitFlags</i> .
<i>PerJobUser-TimeLimit</i>	Максимальное время в пользовательском режиме для всех процессов в данном задании (в порциях по 100 нс)	По умолчанию система автоматически завершает все процессы, когда заканчивается это время. Данное значение можно изменять в процессе выполнения задания. Это ограничение вводится флагом <i>JOB_OBJECT_LIMIT_JOB_TIME</i> в <i>LimitFlags</i> .
<i>LimitFlags</i>	Виды ограничений для задания	См. раздел после таблицы.
<i>MinimumWorking-SetSize</i> и <i>MaximumWorking-SetSize</i>	Верхний и нижний предел рабочего набора для каждого процесса (а не для всех процессов в задании)	Обычно рабочий набор процесса может расширяться за стандартный предел; указав <i>MaximumWorking-SetSize</i> , Вы введете жесткое ограничение. Когда размер рабочего набора какого-либо процесса достигнет заданного предела, процесс начнет сбрасывать свои страницы на диск. Вызовы функции <i>SetProcessWorking-SetSize</i> этим процессом будут игнорироваться, если только он не обращается к ней для того, чтобы очистить свой рабочий набор. Это ограничение вводится флагом <i>JOB_OBJECT_LIMIT_WORKINGSET</i> в <i>LimitFlags</i> .
<i>ActiveProcessLimit</i>	Максимальное количество процессов, одновременно выполняемых в задании	Любая попытка обойти такое ограничение приведет к завершению нового процесса с ошибкой «not enough quota» («превышение квоты»). Это ограничение вводится флагом <i>JOB_OBJECT_LIMIT_ACTIVE_PROCESS</i> в <i>LimitFlags</i> .
<i>Affinity</i>	Подмножество процессоров, на которых можно выполнять процессы этого задания	Для индивидуальных процессов это ограничение можно еще больше детализировать. Вводится флагом <i>JOB_OBJECT_LIMIT_AFFINITY</i> в <i>LimitFlags</i> .

Таблица 5-1. Элементы структуры *JOBOBJECT_BASIC_LIMIT_INFORMATION*

продолжение

Элементы	Описание	Примечание
<i>PriorityClass</i>	Класс приоритета для всех процессов в задании	Вызванная процессом функция <i>SetPriorityClass</i> сообщает об успехе даже в том случае, если на самом деле она не выполнила свою задачу, а <i>GetPriorityClass</i> возвращает класс приоритета, каковой и пытался установить процесс, хотя в реальности его класс может быть совсем другим. Кроме того, <i>SetThreadPriority</i> не может поднять приоритет потоков выше <i>normal</i> , но позволяет понизить его. Это ограничение вводится флагом <i>JOB_OBJECT_LIMIT_PRIORITY_CLASS</i> в <i>LimitFlags</i> .
<i>SchedulingClass</i>	Относительная продолжительность кванта времени, выделяемого всем потокам в задании	Этот элемент может принимать значения от 0 до 9; по умолчанию устанавливается 5. Подробнее о его назначении см. ниже. Это ограничение вводится флагом <i>JOB_OBJECT_LIMIT_SCHEDULING_CLASS</i> в <i>LimitFlags</i> .

Хочу пояснить некоторые вещи, связанные с этой структурой, которые, по-моему довольно туманно изложены в документации Platform SDK. Указывая ограничения для задания, Вы устанавливаете те или иные биты в элементе *LimitFlags*. Например, в *StartRestrictedProcess* я использовал флаги *JOB_OBJECT_LIMIT_PRIORITY_CLASS* и *JOB_OBJECT_LIMIT_JOB_TIME*, т. е. определил всего два ограничения.

При выполнении задание ведет учет по нескольким показателям — например, сколько процессорного времени уже использовали его процессы. Всякий раз, когда Вы устанавливаете базовые ограничения с помощью флага *JOB_OBJECT_LIMIT_JOB_TIME*, из общего процессорного времени, израсходованного всеми процессами, вычитается то, которое использовали завершившиеся процессы. Этот показатель сообщает, сколько процессорного времени израсходовали активные на данный момент процессы. А что если Вам понадобится изменить ограничения на доступ к подмножеству процессоров, не сбрасывая при этом учетную информацию по процессорному времени? Для этого Вы должны ввести новое базовое ограничение флагом *JOB_OBJECT_LIMIT_AFFINITY* и отказаться от флага *JOB_OBJECT_LIMIT_JOB_TIME*. Но тогда получится, что Вы снимаете ограничения на процессорное время.

Вы хотели другого: ограничить доступ к подмножеству процессоров, сохранив существующее ограничение на процессорное время, и не вычитать время, израсходованное завершенными процессами, из общего времени. Чтобы решить эту проблему, используйте специальный флаг *JOB_OBJECT_LIMIT_PRESERVE_JOB_TIME*. Этот флаг и *JOB_OBJECT_LIMIT_JOB_TIME* являются взаимоисключающими. Флаг *JOB_OBJECT_LIMIT_PRESERVE_JOB_TIME* указывает системе изменить ограничения, не вычитая процессорное время, использованное уже завершенными процессами.

Обсудим также элемент *SchedulingClass* структуры *JOBOBJECT_BASIC_LIMIT_INFORMATION*. Представьте, что для двух заданий определен класс приоритета *NORMAL_PRIORITY_CLASS*, а Вы хотите, чтобы процессы одного задания получали больше процессорного времени, чем процессы другого. Так вот, элемент *SchedulingClass* позволяет изменять распределение процессорного времени между заданиями с одинаковым

классом приоритета. Вы можете присвоить ему любое значение в пределах 0–9 (по умолчанию он равен 5). Увеличивая его значение, Вы заставляете Windows 2000 выделять потокам в процессах конкретного задания более длительный квант времени, а снижая — напротив, уменьшаете этот квант.

Допустим, у меня есть два задания с обычным (normal) классом приоритета: в каждом задании — по одному процессу, а в каждом процессе — по одному потоку (тоже с обычным приоритетом). В нормальной ситуации эти два потока обрабатывались бы процессором по принципу карусели и получали бы равные кванты процессорного времени. Но если я запишу в элемент *SchedulingClass* для первого задания значение 3, система будет выделять его потокам более короткий квант процессорного времени, чем потокам второго задания.

Используя *SchedulingClass*, избегайте слишком больших его значений, иначе Вы замедлите общую реакцию других заданий, процессов и потоков на какие-либо события в системе. Кроме того, учтите, что все сказанное здесь относится только к Windows 2000. В будущих версиях Windows планировщик потоков предполагается существенно изменить, чтобы операционная система могла более гибко планировать потоки в заданиях и процессах.

И последнее ограничение, которое заслуживает отдельного упоминания, связано с флагом *JOB_OBJECT_LIMIT_DIE_ON_UNHANDLED_EXCEPTION*. Он отключает для всех процессов в задании вывод диалогового окна с сообщением о необработанном исключении. Система реагирует на этот флаг вызовом *SetErrorMode* с флагом *SEM_NOGPFAULTERRORBOX* для каждого из процессов в задании. Процесс, в котором возникнет необрабатываемое им исключение, немедленно завершается без уведомления пользователя. Этот флаг полезен в сервисных и других пакетных заданиях. В его отсутствие один из процессов в задании мог бы вызвать исключение и не завершиться, впустую расходуя системные ресурсы.

Помимо базовых ограничений, Вы можете устанавливать расширенные, для чего применяется структура *JOBOBJECT_EXTENDED_LIMIT_INFORMATION*:

```
typedef struct _JOBOBJECT_EXTENDED_LIMIT_INFORMATION {
    JOBOBJECT_BASIC_LIMIT_INFORMATION BasicLimitInformation;
    IO_COUNTERS IoInfo;
    SIZE_T ProcessMemoryLimit;
    SIZE_T JobMemoryLimit;
    SIZE_T PeakProcessMemoryUsed;
    SIZE_T PeakJobMemoryUsed;
} JOBOBJECT_EXTENDED_LIMIT_INFORMATION, *PJOBOBJECT_EXTENDED_LIMIT_INFORMATION;
```

Как видите, она включает структуру *JOBOBJECT_BASIC_LIMIT_INFORMATION*, являясь фактически ее надстройкой. Это несколько странная структура, потому что в ней есть элементы, не имеющие никакого отношения к определению ограничений для задания. Во-первых, элемент *IoInfo* зарезервирован, и Вы ни в коем случае не должны обращаться к нему. О том, как узнать значение счетчика ввода-вывода, я расскажу позже. Кроме того, элементы *PeakProcessMemoryUsed* и *PeakJobMemoryUsed* предназначены только для чтения и сообщают о максимальном объеме памяти, переданной соответственно одному из процессов или всем процессам в задании.

Остальные два элемента, *ProcessMemoryLimit* и *JobMemoryLimit*, ограничивают соответственно объем переданной памяти, который может быть использован одним из процессов или всеми процессами в задании. Чтобы задать любое из этих ограничений, укажите в элементе *LimitFlags* флаг *JOB_OBJECT_LIMIT_JOB_MEMORY* или *JOB_OBJECT_LIMIT_PROCESS_MEMORY*.

А теперь вернемся к прочим ограничениям, которые можно налагать на задания. Структура JOBOBJECT_BASIC_UI_RESTRICTIONS выглядит так:

```
typedef struct _JOBOBJECT_BASIC_UI_RESTRICTIONS {
    DWORD UIRestrictionsClass;
} JOBOBJECT_BASIC_UI_RESTRICTIONS, *PJOBOBJECT_BASIC_UI_RESTRICTIONS;
```

В этой структуре всего один элемент, *UIRestrictionsClass*, который содержит набор битовых флагов, кратко описанных в таблице 5-2.

Флаг	Описание
JOB_OBJECT_UILIMIT_EXITWINDOWS	Запрещает выдачу команд из процессов на выход из системы, завершение ее работы, перезагрузку или выключение компьютера через функцию <i>ExitWindowsEx</i>
JOB_OBJECT_UILIMIT_READCLIPBOARD	Запрещает процессам чтение из буфера обмена
JOB_OBJECT_UILIMIT_WRITECLIPBOARD	Запрещает процессам стирание буфера обмена
JOB_OBJECT_UILIMIT_SYSTEMPARAMETERS	Запрещает процессам изменение системных параметров через <i>SystemParametersInfo</i>
JOB_OBJECT_UILIMIT_DISPLAYSETTINGS	Запрещает процессам изменение параметров экрана через <i>ChangeDisplaySettings</i>
JOB_OBJECT_UILIMIT_GLOBALATOMS	Предоставляет заданию отдельную глобальную таблицу атомарного доступа (global atom table) и разрешает его процессам пользоваться только этой таблицей
JOB_OBJECT_UILIMIT_DESKTOP	Запрещает процессам создание новых рабочих столов или переключение между ними через функции <i>CreateDesktop</i> или <i>SwitchDesktop</i>
JOB_OBJECT_UILIMIT_HANDLES	Запрещает процессам в задании использовать USER-объекты (например, HWND), созданные внешними по отношению к этому заданию процессами

Таблица 5-2. Битовые флаги базовых ограничений по пользовательскому интерфейсу для объекта-задания

Последний флаг, JOB_OBJECT_UILIMIT_HANDLES, представляет особый интерес: он запрещает процессам в задании обращаться к USER-объектам, созданным внешними по отношению к этому заданию процессами. Так, запустив утилиту Microsoft Spy++ из задания, Вы не обнаружите никаких окон, кроме тех, которые создаст сама Spy++. На рис. 5-2 показано окно Microsoft Spy++ с двумя открытыми дочерними MDI-окнами. Заметьте, что в левой секции (Threads 1) содержится список потоков в системе. Кажется, что лишь у одного из них, 000006AC SPYXX, есть дочерние окна. А все дело в том, что я запустил Microsoft Spy++ из задания и ограничил ему права на использование описателей USER-объектов. В том же окне сообщается о потоках MSDEV и EXPLORER, но никаких упоминаний о созданных ими окнах нет. Уверяю Вас, эти потоки наверняка создали какие-нибудь окна — просто Spy++ лишена возможности их видеть. В правой секции (Windows 3) утилиты Spy++ должна показывать иерархию окон на рабочем столе, но там нет ничего, кроме одного элемента — 00000000. (Это не настоящий элемент, но Spy++ была обязана поместить сюда хоть что-нибудь.)

Обратите внимание, что такие ограничения односторонни, т. е. внешние процессы все равно видят USER-объекты, которые созданы процессами, включенными в задание. Например, если запустить Notepad в задании, а Spy++ — вне его, последняя увидит окно Notepad, даже если для задания указан флаг `JOB_OBJECT_UILIMIT_HANDLES`. Кроме того, Spy++, запущенная в отдельном задании, все равно увидит это окно Notepad, если только для ее задания не установлен флаг `JOB_OBJECT_UILIMIT_HANDLES`.

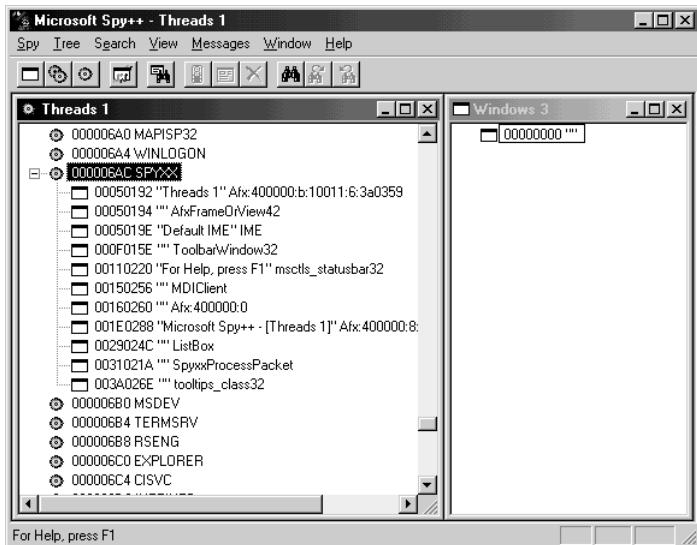


Рис. 5-2. Microsoft Spy++ работает в задании, которому ограничен доступ к описателям USER-объектов

Ограничение доступа к описателям USER-объектов — вещь изумительная, если Вы хотите создать по-настоящему безопасную песочницу, в которой будут «копаться» процессы Вашего задания. Однако часто бывает нужно, чтобы процесс в задании взаимодействовал с внешними процессами. Одно из самых простых решений здесь — использовать оконные сообщения, но, если процессам в задании доступ к описателям пользовательского интерфейса запрещен, ни один из них не сможет послать сообщение (синхронно или асинхронно) окну, созданному внешним процессом. К счастью, теперь есть функция, которая поможет решить эту проблему:

```
BOOL UserHandleGrantAccess(
    HANDLE hUserObj,
    HANDLE hjob,
    BOOL fGrant);
```

Параметр `hUserObj` идентифицирует конкретный USER-объект, доступ к которому Вы хотите предоставить или запретить процессам в задании. Это почти всегда описатель окна, но USER-объектом может быть, например, рабочий стол, программная ловушка, ярлык или меню. Последние два параметра, `hjob` и `fGrant`, указывают на задание и вид ограничения. Обратите внимание, что функция не сработает, если ее вызвать из процесса в том задании, на которое указывает `hjob`, — процесс не имеет права сам себе предоставлять доступ к объекту.

И последний вид ограничений, устанавливаемых для задания, относится к защите. (Введя в действие такие ограничения, Вы не сможете их отменить.) Структура `JOBOBJECT_SECURITY_LIMIT_INFORMATION` выглядит так:

```

typedef struct _JOBOBJECT_SECURITY_LIMIT_INFORMATION {
    DWORD SecurityLimitFlags;
    HANDLE JobToken;
    PTOKEN_GROUPS SidsToDelete;
    PTOKEN_PRIVILEGES PrivilegesToDelete;
    PTOKEN_GROUPS RestrictedSids;
} JOBOBJECT_SECURITY_LIMIT_INFORMATION, *PJOBOBJECT_SECURITY_LIMIT_INFORMATION;

```

Ее элементы описаны в следующей таблице.

Элемент	Описание
<i>SecurityLimitFlags</i>	Набор флагов, которые закрывают доступ администратору, запрещают маркер неограниченного доступа, принудительно назначают заданный маркер доступа, блокируют доступ по каким-либо идентификаторам защиты (security ID, SID) и отменяют указанные привилегии
<i>JobToken</i>	Маркер доступа, связываемый со всеми процессами в задании
<i>SidsToDelete</i>	Указывает, по каким SID не разрешается доступ
<i>PrivilegesToDelete</i>	Определяет привилегии, которые снимаются с маркера доступа
<i>RestrictedSids</i>	Задает набор SID, по которым запрещается доступ к любому защищенному объекту (deny-only SIDs); этот набор добавляется к маркеру доступа

Естественно, если Вы налагаете ограничения, то потом Вам, наверное, понадобится информация о них. Для этого вызовите:

```

BOOL QueryInformationJobObject(
    HANDLE hJob,
    JOBOBJECTINFOCLASS JobObjectInformationClass,
    PVOID pvJobObjectInformation,
    DWORD cbJobObjectInformationLength,
    PDWORD pdwReturnLength);

```

В эту функцию, как и в *SetInformationJobObject*, передается описатель задания, переменная перечислимого типа *JOBOBJECTINFOCLASS*. Она сообщает информацию об ограничениях, адрес и размер структуры данных, инициализируемой функцией. Последний параметр, *pdwReturnLength*, заполняется самой функцией и указывает, сколько байтов помещено в буфер. Если эти сведения Вас не интересуют (что обычно и бывает), передавайте в этом параметре NULL.



Процесс может получить информацию о своем задании, передав при вызове *QueryInformationJobObject* вместо описателя задания значение NULL. Это позволит ему выяснить установленные для него ограничения. Однако аналогичный вызов *SetInformationJobObject* даст ошибку, так как процесс не имеет права самостоятельно изменять заданные для него ограничения.

Включение процесса в задание

О'кэй, с ограничениями на этом закончим. Вернемся к *StartRestrictedProcess*. Установив ограничения для задания, я вызываю *CreateProcess* и создаю процесс, который помещаю в это задание. Я использую здесь флаг *CREATE_SUSPENDED*, и он приводит к тому, что процесс порождается, но код пока не выполняет. Поскольку *StartRestrictedProcess* вызывается из процесса, внешнего по отношению к заданию, его дочерний

процесс тоже не входит в это задание. Если бы я разрешил дочернему процессу немедленно начать выполнение кода, он проигнорировал бы мою песочницу со всеми ее ограничениями. Поэтому сразу после создания дочернего процесса и перед началом его работы я должен явно включить этот процесс в только что сформированное задание, вызвав:

```
BOOL AssignProcessToJobObject(  
    HANDLE hJob,  
    HANDLE hProcess);
```

Эта функция заставляет систему рассматривать процесс, идентифицируемый параметром *hProcess*, как часть существующего задания, на которое указывает *hJob*. Обратите внимание, что *AssignProcessToJobObject* позволяет включить в задание только тот процесс, который еще не относится ни к одному заданию. Как только процесс стал частью какого-нибудь задания, его нельзя переместить в другое задание или отпустить на волю. Кроме того, когда процесс, включенный в задание, порождает новый процесс, последний автоматически помещается в то же задание. Однако этот порядок можно изменить.

- Включая в *LimitFlags* структуры *JOBOBJECT_BASIC_LIMIT_INFORMATION* флаг *JOB_OBJECT_BREAKAWAY_OK*, Вы сообщаете системе, что новый процесс может выполняться вне задания. Потом Вы должны вызвать *CreateProcess* с новым флагом *CREATE_BREAKAWAY_FROM_JOB*. (Если Вы сделаете это без флага *JOB_OBJECT_BREAKAWAY_OK* в *LimitFlags*, функция *CreateProcess* завершится с ошибкой.) Такой механизм пригодится на случай, если новый процесс тоже управляет заданиями.
- Включая в *LimitFlags* структуры *JOBOBJECT_BASIC_LIMIT_INFORMATION* флаг *JOB_OBJECT_SILENT_BREAKAWAY_OK*, Вы тоже сообщаете системе, что новый процесс не является частью задания. Но указывать в *CreateProcess* какие-либо флаги на этот раз не потребуется. Данный механизм полезен для процессов, которым ничего не известно об объектах-заданиях.

Что касается *StartRestrictedProcess*, то после вызова *AssignProcessToJobObject* новый процесс становится частью задания. Далее язываю *ResumeThread*, чтобы поток нового процесса начал выполняться в рамках ограничений, установленных для задания. В этот момент я также закрываю описатель потока, поскольку он мне больше не нужен.

Завершение всех процессов в задании

Уверен, именно это Вы и будете делать чаще всего. В начале главы я упомянул о том, как непросто остановить сборку в Developer Studio, потому что для этого ему должны быть известны все процессы, которые успел создать его самый первый процесс. (Это очень каверзная задача. Как Developer Studio справляется с ней, я объяснял в своей колонке «Вопросы и ответы по Win32» в июньском выпуске Microsoft Systems Journal за 1998 год.) Подозреваю, что следующие версии Developer Studio будут использовать механизм заданий, и решать задачу, о которой мы с Вами говорили, станет гораздо легче.

Чтобы уничтожить все процессы в задании, Вы просто вызываете:

```
BOOL TerminateJobObject(  
    HANDLE hJob,  
    UINT uExitCode);
```

Вызов этой функции похож на вызов *TerminateProcess* для каждого процесса в задании и присвоение всем кодам завершения одного значения — *uExitCode*.

Получение статистической информации о задании

Мы уже обсудили, как с помощью *QueryInformationJobObject* получить информацию о текущих ограничениях, установленных для задания. Этой функцией можно пользоваться и для получения статистической информации. Например, чтобы выяснить базовые учетные сведения, вызовите ее, передав *JobObjectBasicAccountingInformation* во втором параметре и адрес структуры *JOBOBJECT_BASIC_ACCOUNTING_INFORMATION*:

```
typedef struct _JOBOBJECT_BASIC_ACCOUNTING_INFORMATION {
    LARGE_INTEGER TotalUserTime;
    LARGE_INTEGER TotalKernelTime;
    LARGE_INTEGER ThisPeriodTotalUserTime;
    LARGE_INTEGER ThisPeriodTotalKernelTime;
    DWORD TotalPageFaultCount;
    DWORD TotalProcesses;
    DWORD ActiveProcesses;
    DWORD TotalTerminatedProcesses;
} JOBOBJECT_BASIC_ACCOUNTING_INFORMATION, *PJOBOBJECT_BASIC_ACCOUNTING_INFORMATION;
```

Элементы этой структуры кратко описаны в таблице 5-3.

Элемент	Описание
<i>TotalUserTime</i>	Процессорное время, израсходованное процессами задания в пользовательском режиме
<i>TotalKernelTime</i>	Процессорное время, израсходованное процессами задания в режиме ядра
<i>ThisPeriodTotalUserTime</i>	То же, что <i>TotalUserTime</i> , но обнуляется, когда базовые ограничения изменяются вызовом <i>SetInformationJobObject</i> , а флаг <i>JOB_OBJECT_LIMIT_PRESERVE_JOB_TIME</i> не используется
<i>ThisPeriodTotalKernelTime</i>	То же, что <i>ThisPeriodTotalUserTime</i> , но относится к процессорному времени, израсходованному в режиме ядра
<i>TotalPageFaultCount</i>	Общее количество ошибок страниц, вызванных процессами задания
<i>TotalProcesses</i>	Общее число процессов, когда-либо выполнявшихся в этом задании
<i>ActiveProcesses</i>	Текущее количество процессов в задании
<i>TotalTerminatedProcesses</i>	Количество процессов, завершенных из-за превышения ими отведенного лимита процессорного времени

Таблица 5-3. Элементы структуры *JOBOBJECT_BASIC_ACCOUNTING_INFORMATION*

Вы можете извлечь те же сведения вместе с учетной информацией по вводу-выводу, передав *JobObjectBasicAndIoAccountingInformation* во втором параметре и адрес структуры *JOBOBJECT_BASIC_AND_IO_ACCOUNTING_INFORMATION*:

```
typedef struct _JOBOBJECT_BASIC_AND_IO_ACCOUNTING_INFORMATION {
    JOBOBJECT_BASIC_ACCOUNTING_INFORMATION BasicInfo;
    IO_COUNTERS IoInfo;
} JOBOBJECT_BASIC_AND_IO_ACCOUNTING_INFORMATION;
```

Как видите, она просто возвращает *JOBOBJECT_BASIC_ACCOUNTING_INFORMATION* и *IO_COUNTERS*. Последняя структура показана на следующей странице.

```
typedef struct _IO_COUNTERS {
    ULONGLONG ReadOperationCount;
    ULONGLONG WriteOperationCount;
    ULONGLONG OtherOperationCount;
    ULONGLONG ReadTransferCount;
    ULONGLONG WriteTransferCount;
    ULONGLONG OtherTransferCount;
} IO_COUNTERS;
```

Она сообщает о числе операций чтения, записи и перемещения (а также о количестве байтов, переданных при выполнении этих операций). Данные относятся ко всем процессам в задании. Кстати, новая функция *GetProcessIoCounters* позволяет получить ту же информацию о процессах, не входящих ни в какие задания.

```
BOOL GetProcessIoCounters(
    HANDLE hProcess,
    PIO_COUNTERS pIoCounters);
```

QueryInformationJobObject также возвращает набор идентификаторов текущих процессов в задании. Но перед этим Вы должны прикинуть, сколько их там может быть, и выделить соответствующий блок памяти, где поместятся массив идентификаторов и структура *JOBOBJECT_BASIC_PROCESS_ID_LIST*:

```
typedef struct _JOBOBJECT_BASIC_PROCESS_ID_LIST {
    DWORD NumberOfAssignedProcesses;
    DWORD NumberOfProcessIdsInList;
    DWORD ProcessIdList[1];
} JOBOBJECT_BASIC_PROCESS_ID_LIST, *PJOBOBJECT_BASIC_PROCESS_ID_LIST;
```

В итоге, чтобы получить набор идентификаторов текущих процессов в задании, нужно написать примерно такой код:

```
void EnumProcessIdsInJob(HANDLE hjob) {

    // я исхожу из того, что количество процессов
    // в этом задании никогда не превысит 10
    #define MAX_PROCESS_IDS      10

    // определяем размер блока памяти (в байтах)
    // для хранения идентификаторов и структуры
    DWORD cb = sizeof(JOBOBJECT_BASIC_PROCESS_ID_LIST) +
        (MAX_PROCESS_IDS - 1) * sizeof(DWORD);

    // выделяем этот блок памяти
    PJOBOBJECT_BASIC_PROCESS_ID_LIST pjobpil = _alloca(cb);

    // сообщаем функции, на какое максимальное число процессов
    // рассчитана выделенная нами память
    pjobpil->NumberOfAssignedProcesses = MAX_PROCESS_IDS;

    // запрашиваем текущий список идентификаторов процессов
    QueryInformationJobObject(hjob, JobObjectBasicProcessIdList,
        pjobpil, cb, &cb);

    // перечисляем идентификаторы процессов
    for (int x = 0; x < pjobpil->NumberOfProcessIdsInList; x++) {
```

```

    // используем pjobpil->ProcessIdList[x]...
}

// так как для выделения памяти мы вызывали _alloc,
// освобождать память нам не потребуется
}

```

Вот и все, что Вам удастся получить через эти функции, хотя на самом деле операционная система знает о заданиях гораздо больше. Эту информацию, которая хранится в специальных счетчиках, можно извлечь с помощью функций из библиотеки Performance Data Helper (PDH.dll) или через модуль Performance Monitor, подключаемый к Microsoft Management Console (MMC). Рис. 5-3 иллюстрирует некоторые из доступных в системе счетчиков заданий (job object counters), а рис. 5-4 — счетчики, относящиеся к отдельным параметрам заданий (job object details counters). Заметьте, что в задании Jeff содержится четыре процесса: calc, cmd, notepad и wordpad.

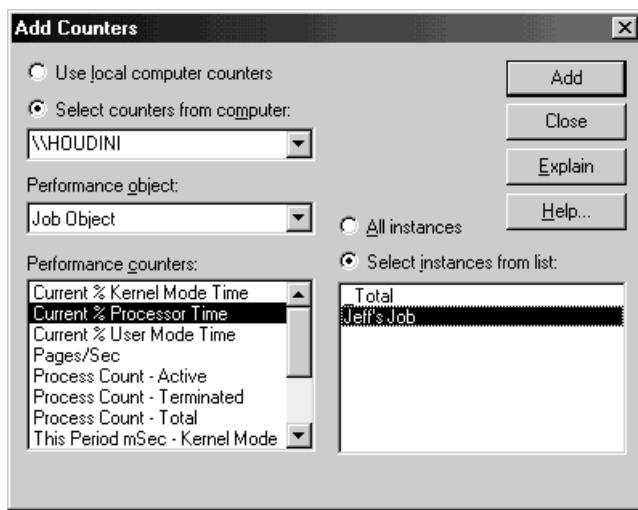


Рис. 5-3. MMC Performance Monitor: счетчики задания

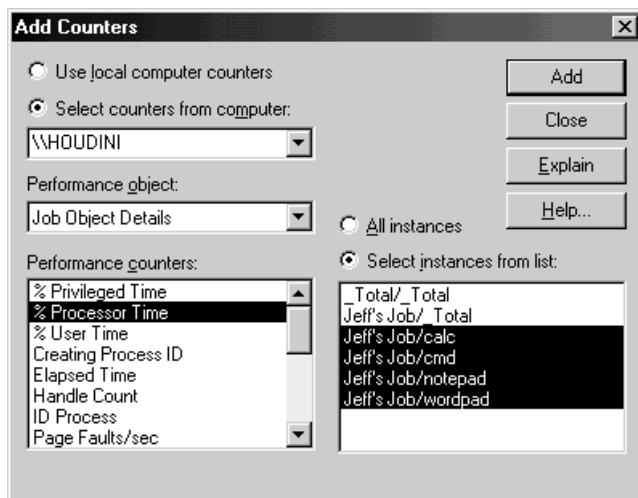


Рис. 5-4. MMC Performance Monitor: счетчики, относящиеся к отдельным параметрам задания

Извлечь сведения из этих счетчиков Вы сможете только для тех заданий, которым были присвоены имена при вызове *CreateJobObject*. По этой причине, наверное, лучше всегда именовать задания, даже если Вы и не собираетесь ссылаться на них по именам из других процессов.

Уведомления заданий

Итак, базовые сведения об объектах-заданиях я изложил. Единственное, что осталось рассмотреть, — уведомления. Допустим, Вам нужно знать, когда завершаются все процессы в задании или заканчивается все отпущенное им процессорное время. Либо выяснить, когда в задании порождается или уничтожается очередной процесс. Если такие уведомления Вас не интересуют (а во многих приложениях они и не нужны), работать с заданиями будет очень легко — не сложнее, чем я уже рассказывал. Но если они все же понадобятся, Вам придется копнуть чуть поглубже.

Информацию о том, все ли выделенное процессорное время исчерпано, получить нетрудно. Объекты-задания не переходят в свободное состояние до тех пор, пока их процессы не израсходуют отведенное процессорное время. Как только оно заканчивается, система уничтожает все процессы в задании и переводит его объект в свободное состояние (*signaled state*). Это событие легко перехватить с помощью *WaitForSingleObject* (или похожей функции). Кстати, потом Вы можете вернуть объект-задание в состояние «занято» (*nonsignaled state*), вызвав *SetInformationJobObject* и выделив ему дополнительное процессорное время.

Когда я только начинал разбираться с заданиями, мне казалось, что объект-задание должен переходить в свободное состояние после завершения всех его процессов. В конце концов, прекращая свою работу, объекты процессов и потоков освобождаются; то же самое вроде бы должно происходить и с заданиями. Но Microsoft предпочла сделать по-другому: объект-задание переходит в свободное состояние после того, как исчерпает выделенное ему время. Поскольку большинство заданий начинает свою работу с одним процессом, который существует, пока не завершатся все его дочерние процессы, Вам нужно просто следить за описателем родительского процесса — он освободится, как только завершится все задание. Моя функция *StartRestrictedProcess* как раз и демонстрирует данный прием.

Но это были лишь простейшие уведомления — более «продвинутые», например о создании или разрушении процесса, получать гораздо сложнее. В частности, Вам придется создать объект ядра «порт завершения ввода-вывода» и связать с ним объект или объекты «задание». После этого нужно будет перевести один или больше потоков в режим ожидания порта завершения.

Создав порт завершения ввода-вывода, Вы сопоставляете с ним задание, вызывая *SetInformationJobObject* следующим образом:

```
JOBOBJECT_ASSOCIATE_COMPLETION_PORT joacp;
joacp.CompletionKey = 1; // любое значение, уникально идентифицирующее
                        // это задание
joacp.CompletionPort = hIOCP; // описатель порта завершения, принимающего
                            // уведомления
SetInformationJobObject(hJob, JobObjectAssociateCompletionPortInformation,
    &joacp, sizeof(joacp));
```

После выполнения этого кода система начнет отслеживать задание и при возникновении событий передавать их порту завершения. (Кстати, Вы можете вызывать *QueryInformationJobObject* и получать ключ завершения и описатель порта, но вряд ли

это Вам когда-нибудь понадобится.) Потоки следят за портом завершения ввода-вывода, вызывая *GetQueuedCompletionStatus*:

```
BOOL GetQueuedCompletionStatus(
    HANDLE hIOCP,
    PDWORD pNumBytesTransferred,
    PULONG_PTR pCompletionKey,
    POVERLAPPED *pOverlapped,
    DWORD dwMilliseconds);
```

Когда эта функция возвращает уведомление о событии задания, **pCompletionKey* содержит значение ключа завершения, заданное при вызове *SetInformationJobObject* для связывания задания с портом завершения. По нему Вы узнаете, в каком из заданий возникло событие. Значение в **pNumBytesTransferred* указывает, какое именно событие произошло (таблица 5-4). В зависимости от конкретного события в **pOverlapped* может возвращаться идентификатор процесса.

Событие	Описание
JOB_OBJECT_MSG_ACTIVE_PROCESS_ZERO	В задании нет работающих процессов
JOB_OBJECT_MSG_END_OF_PROCESS_TIME	Процессорное время, выделенное процессу, исчерпано; процесс завершается, и сообщается его идентификатор
JOB_OBJECT_MSG_ACTIVE_PROCESS_LIMIT	Была попытка превысить ограничение на число активных процессов в задании
JOB_OBJECT_MSG_PROCESS_MEMORY_LIMIT	Была попытка превысить ограничение на объем памяти, которая может быть передана процессу; сообщается идентификатор процесса
JOB_OBJECT_MSG_JOB_MEMORY_LIMIT	Была попытка превысить ограничение на объем памяти, которая может быть передана заданию; сообщается идентификатор процесса
JOB_OBJECT_MSG_NEW_PROCESS	В задание добавлен процесс; сообщается идентификатор процесса
JOB_OBJECT_MSG_EXIT_PROCESS	Процесс завершен; сообщается идентификатор процесса
JOB_OBJECT_MSG_ABNORMAL_EXIT_PROCESS	Процесс завершен из-за необработанного им исключения; сообщается идентификатор процесса
JOB_OBJECT_MSG_END_OF_JOB_TIME	Процессорное время, выделенное заданию, исчерпано; процессы не завершаются, и Вы можете либо возобновить их работу, задав новый лимит по времени, либо самостоятельно завершить процессы, вызвав <i>TerminateJobObject</i>

Таблица 5-4. Уведомления о событиях задания, посылаемые системой связанным с этим заданием порту завершения

И последнее замечание: по умолчанию объект-задание настраивается системой на автоматическое завершение всех его процессов по истечении выделенного ему процессорного времени, а уведомление *JOB_OBJECT_MSG_END_OF_JOB_TIME* не посылается. Если Вы хотите, чтобы объект-задание не уничтожал свои процессы, а просто сообщал о превышении лимита на процессорное время, Вам придется написать примерно такой код:

```
// создаем структуру JOB_OBJECT_END_OF_JOB_TIME_INFORMATION
// и инициализируем ее единственный элемент
```

см. след. стр.

```
JOBOBJECT_END_OF_JOB_TIME_INFORMATION joeobji;
joeobji.EndOfJobTimeAction = JOB_OBJECT_POST_AT_END_OF_JOB;
```

```
// сообщаем заданию, что ему нужно делать по истечении его времени
SetInformationJobObject(hJob, JobObjectEndOfJobTimeInformation,
    &joeobji, sizeof(joeobji));
```

Вы можете указать и другое значение, JOB_OBJECT_TERMINATE_AT_END_OF_JOB, но оно задается по умолчанию, еще при создании задания.

Программа-пример JobLab

Эта программа, «05 JobLab.exe» (см. листинг на рис. 5-6), позволяет легко экспериментировать с заданиями. Ее файлы исходного кода и ресурсов находятся в каталоге 05-JobLab на компакт-диске, прилагаемом к книге. После запуска JobLab открывается окно, показанное на рис. 5-5.

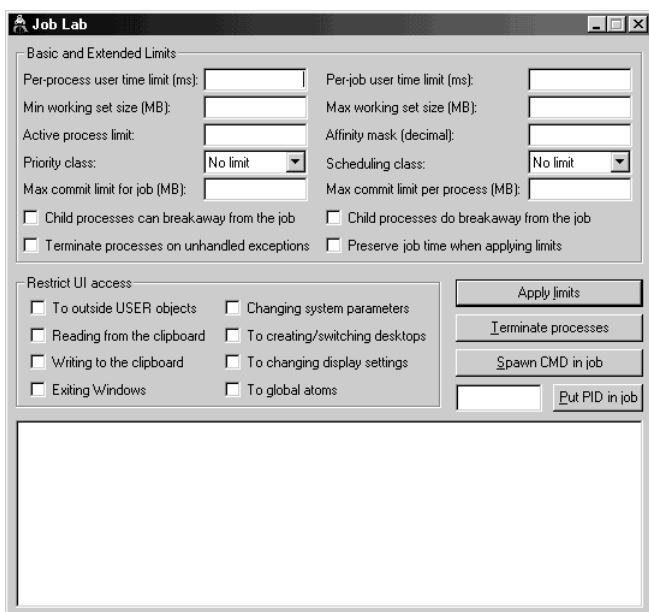


Рис. 5-5. Программа-пример JobLab

Когда процесс инициализируется, он создает объект «задание». Я присваиваю ему имя JobLab, чтобы Вы могли наблюдать за ним с помощью MMC Performance Monitor. Моя программа также создает порт завершения ввода-вывода и связывает с ним объект-задание. Это позволяет отслеживать уведомления от задания и отображать их в списке в нижней части окна.

Изначально в задании нет процессов, и никаких ограничений для него не установлено. Поля в верхней части окна позволяют задавать базовые и расширенные ограничения. Все, что от Вас требуется, — ввести в них допустимые значения и щелкнуть кнопку Apply Limits. Если Вы оставляете поле пустым, соответствующие ограничения не вводятся. Кроме базовых и расширенных, Вы можете задавать ограничения по пользовательскому интерфейсу. Обратите внимание: помечая флажок Preserve Job Time When Applying Limits, Вы не устанавливаете ограничение, а просто получаете возможность изменять ограничения, не сбрасывая значения элементов ThisPeriod-

TotalUserTime и *ThisPeriodTotalKernelTime* при запросе базовой учетной информации. Этот флагок становится недоступен при наложении ограничений на процессорное время для отдельных заданий.

Остальные кнопки позволяют управлять заданием по-другому. Кнопка Terminate Processes уничтожает все процессы в задании. Кнопка Spawn CMD In Job запускает командный процессор, сопоставляемый с заданием. Из этого процесса можно запускать дочерние процессы и наблюдать, как они ведут себя, став частью задания. И последняя кнопка, Put PID In Job, позволяет связать существующий свободный процесс с заданием (т. е. включить его в задание).

Список в нижней части окна отображает обновляемую каждые 10 секунд информацию о статусе задания: базовые и расширенные сведения, статистику ввода-вывода, а также пиковые объемы памяти, занимаемые процессом и заданием.

Кроме этой информации, в списке показываются уведомления, поступающие от задания в порт завершения ввода-вывода. (Кстати, вся информация обновляется и при приеме уведомления.)

И еще одно: если Вы измените исходный код и будете создавать безымянный объект ядра «задание», то сможете запускать несколько копий этой программы, создавая тем самым два и более объектов-заданий на одной машине. Это расширит Ваши возможности в экспериментах с заданиями.

Что касается исходного кода, то специально обсуждать его нет смысла — в нем и так достаточно комментариев. Замечу лишь, что в файле Job.h я определил C++-класс CJob, инкапсулирующий объект «задание» операционной системы. Это избавило меня от необходимости передавать туда-сюда описатель задания и позволило уменьшить число операций приведения типов, которые обычно приходится выполнять при вызове функций *QueryInformationJobObject* и *SetInformationJobObject*.



JobLab.cpp

```

/*****
Modуль: JobLab.cpp
Автор: Copyright (c) 2000, Джейфри Рихтер (Jeffrey Richter)
*****/

#include "..\CmnHdr.h"
#include <windowsx.h>
#include <process.h> // для доступа к _beginthreadex
#include <tchar.h>
#include <stdio.h>
#include "Resource.h"
#include "Job.h"

///////////////////////////////
CJob g_job;           // задание
HWND g_hwnd;          // описатель диалогового окна (доступен всем потокам)
HANDLE g_hIOCP;        // порт завершения, принимающий уведомления от задания
HANDLE g_hThreadIOCP; // поток порта завершения

```

Рис. 5-6. Программа-пример JobLab

см. след. стр.

Рис. 5-6. продолжение

```
// ключи завершения, относящиеся к порту завершения
#define COMPKEY_TERMINATE ((UINT_PTR) 0)
#define COMPKEY_STATUS   ((UINT_PTR) 1)
#define COMPKEY_JOBOBJECT ((UINT_PTR) 2)

///////////////////////////////



DWORD WINAPI JobNotify(PVOID) {
    TCHAR sz[2000];
    BOOL fDone = FALSE;

    while (!fDone) {
        DWORD dwBytesXferred;
        ULONG_PTR CompKey;
        LPOVERLAPPED po;
        GetQueuedCompletionStatus(g_hIOCP,
            &dwBytesXferred, &CompKey, &po, INFINITE);

        // приложение закрывается, выходим из этого потока
        fDone = (CompKey == COMPKEY_TERMINATE);

        HWND hwndLB = FindWindow(NULL, TEXT("Job Lab"));
        hwndLB = GetDlgItem(hwndLB, IDC_STATUS);

        if (CompKey == COMPKEY_JOBOBJECT) {
            lstrcpy(sz, TEXT("--> Notification: "));
            LPTSTR psz = sz + lstrlen(sz);
            switch (dwBytesXferred) {
                case JOB_OBJECT_MSG_END_OF_JOB_TIME:
                    wsprintf(psz, TEXT("Job time limit reached"));
                    break;

                case JOB_OBJECT_MSG_END_OF_PROCESS_TIME:
                    wsprintf(psz, TEXT("Job process (Id=%d) time limit reached"), po);
                    break;

                case JOB_OBJECT_MSG_ACTIVE_PROCESS_LIMIT:
                    wsprintf(psz, TEXT("Too many active processes in job"));
                    break;

                case JOB_OBJECT_MSG_ACTIVE_PROCESS_ZERO:
                    wsprintf(psz, TEXT("Job contains no active processes"));
                    break;

                case JOB_OBJECT_MSG_NEW_PROCESS:
                    wsprintf(psz, TEXT("New process (Id=%d) in job"), po);
                    break;

                case JOB_OBJECT_MSG_EXIT_PROCESS:
                    wsprintf(psz, TEXT("Process (Id=%d) terminated"), po);
                    break;
            }
        }
    }
}
```

Рис. 5-6. продолжение

```

case JOB_OBJECT_MSG_ABNORMAL_EXIT_PROCESS:
    wsprintf(psz, TEXT("Process (Id=%d) terminated abnormally"), po);
    break;

case JOB_OBJECT_MSG_PROCESS_MEMORY_LIMIT:
    wsprintf(psz, TEXT("Process (Id=%d) exceeded memory limit"), po);
    break;

case JOB_OBJECT_MSG_JOB_MEMORY_LIMIT:
    wsprintf(psz,
        TEXT("Process (Id=%d) exceeded job memory limit"), po);
    break;

default:
    wsprintf(psz, TEXT("Unknown notification: %d"), dwBytesXferred);
    break;
}
ListBox_SetCurSel(hwndLB, ListBox_AddString(hwndLB, sz));
CompKey = 1; // принудительное обновление списка
// при получении уведомления

if (CompKey == COMPKEY_STATUS) {
    static int s_nStatusCount = 0;
    _stprintf(sz, TEXT("--> Status Update (%u)'), s_nStatusCount++);
    ListBox_SetCurSel(hwndLB, ListBox_AddString(hwndLB, sz));

    // выводим базовую учетную информацию
    JOBOBJECT_BASIC_AND_IO_ACCOUNTING_INFORMATION jobai;
    g_job.QueryBasicAccountingInfo(&jobai);

    _stprintf(sz, TEXT("Total Time: User=%I64u, Kernel=%I64u      "))
        TEXT("Period Time: User=%I64u, Kernel=%I64u"),
        jobai.BasicInfo.TotalUserTime.QuadPart,
        jobai.BasicInfo.TotalKernelTime.QuadPart,
        jobai.BasicInfo.ThisPeriodTotalUserTime.QuadPart,
        jobai.BasicInfo.ThisPeriodTotalKernelTime.QuadPart);
    ListBox_SetCurSel(hwndLB, ListBox_AddString(hwndLB, sz));

    _stprintf(sz, TEXT("Page Faults=%u, Total Processes=%u, "))
        TEXT("Active Processes=%u, Terminated Processes=%u"),
        jobai.BasicInfo.TotalPageFaultCount,
        jobai.BasicInfo.TotalProcesses,
        jobai.BasicInfo.ActiveProcesses,
        jobai.BasicInfo.TotalTerminatedProcesses);
    ListBox_SetCurSel(hwndLB, ListBox_AddString(hwndLB, sz));

    // показываем учетную информацию по вводу-выводу
    _stprintf(sz, TEXT("Reads=%I64u (%I64u bytes), "))
        TEXT("Write=%I64u (%I64u bytes), Other=%I64u (%I64u bytes"),
        jobai.IoInfo.ReadOperationCount, jobai.IoInfo.ReadTransferCount,
        jobai.IoInfo.WriteOperationCount, jobai.IoInfo.WriteTransferCount,

```

см. след. стр.

Рис. 5-6. продолжение

```
        jobai.IoInfo.OtherOperationCount, jobai.IoInfo.OtherTransferCount);
    ListBox_SetCurSel(hwndLB, ListBox_AddString(hwndLB, sz));

    // сообщаем пиковые значения объема памяти, использованные
    // процессами и заданиями
    JOBOBJECT_EXTENDED_LIMIT_INFORMATION joeli;
    g_job.QueryExtendedLimitInfo(&joeli);
    _stprintf(sz, TEXT("Peak memory used: Process=%I64u, Job=%I64u"),
              (_int64) joeli.PeakProcessMemoryUsed,
              (_int64) joeli.PeakJobMemoryUsed);
    ListBox_SetCurSel(hwndLB, ListBox_AddString(hwndLB, sz));

    // выводим список идентификаторов процессов
    DWORD dwNumProcesses = 50, dwProcessIdList[50];
    g_job.QueryBasicProcessIdList(dwNumProcesses,
                                  dwProcessIdList, &dwNumProcesses);
    _stprintf(sz, TEXT("PIDs: %s"),
              (dwNumProcesses == 0) ? TEXT("(none)") : TEXT(""));
    for (DWORD x = 0; x < dwNumProcesses; x++) {
        _stprintf(_tcschr(sz, 0), TEXT("%d "), dwProcessIdList[x]);
    }
    ListBox_SetCurSel(hwndLB, ListBox_AddString(hwndLB, sz));
}

return(0);
}

//////////
```

```
BOOL Dlg_OnInitDialog(HWND hwnd, HWND hwndFocus, LPARAM lParam) {

    chSETDLGICONS(hwnd, IDI_JOBLAB);

    // сохраняем описатель нашего окна, чтобы поток
    // порта завершения мог к нему обращаться
    g_hwnd = hwnd;

    HWND hwndPriorityClass = GetDlgItem(hwnd, IDC_PRIORITYCLASS);
    ComboBox_AddString(hwndPriorityClass, TEXT("No limit"));
    ComboBox_AddString(hwndPriorityClass, TEXT("Idle"));
    ComboBox_AddString(hwndPriorityClass, TEXT("Below normal"));
    ComboBox_AddString(hwndPriorityClass, TEXT("Normal"));
    ComboBox_AddString(hwndPriorityClass, TEXT("Above normal"));
    ComboBox_AddString(hwndPriorityClass, TEXT("High"));
    ComboBox_AddString(hwndPriorityClass, TEXT("Realtime"));
    ComboBox_SetCurSel(hwndPriorityClass, 0); // по умолчанию - "No limit"

    HWND hwndSchedulingClass = GetDlgItem(hwnd, IDC_SCHEDULINGCLASS);
    ComboBox_AddString(hwndSchedulingClass, TEXT("No limit"));
    for (int n = 0; n <= 9; n++) {
        TCHAR szSchedulingClass[2] = { (TCHAR) (TEXT('0') + n), 0 };
        ComboBox_SetString(hwndSchedulingClass, szSchedulingClass);
    }
}
```

Рис. 5-6. продолжение

```

    ComboBox_AddString(hwndSchedulingClass, szSchedulingClass);
}
ComboBox_SetCurSel(hwndSchedulingClass, 0); // по умолчанию - "No limit"
SetTimer(hwnd, 1, 10000, NULL); // обновление каждые 10 секунд
return(TRUE);
}

///////////////////////////////



void Dlg_ApplyLimits(HWND hwnd) {
const int nNanosecondsPerSecond = 100000000;
const int nMillisecondsPerSecond = 1000;
const int nNanosecondsPerMillisecond =
    nNanosecondsPerSecond / nMillisecondsPerSecond;
BOOL f;
__int64 q;
SIZE_T s;
DWORD d;

// устанавливаем базовые и расширенные ограничения
JOBOBJECT_EXTENDED_LIMIT_INFORMATION joeli = { 0 };
joeli.BasicLimitInformation.LimitFlags = 0;

q = GetDlgItemInt(hwnd, IDC_PERPROCESSUSERTIMELIMIT, &f, FALSE);
if (f) {
    joeli.BasicLimitInformation.LimitFlags |= JOB_OBJECT_LIMIT_PROCESS_TIME;
    joeli.BasicLimitInformation.PerProcessUserTimeLimit.QuadPart =
        q * nNanosecondsPerMillisecond / 100;
}

q = GetDlgItemInt(hwnd, IDC_PERJOBUSERTIMELIMIT, &f, FALSE);
if (f) {
    joeli.BasicLimitInformation.LimitFlags |= JOB_OBJECT_LIMIT_JOB_TIME;
    joeli.BasicLimitInformation.PerJobUserTimeLimit.QuadPart =
        q * nNanosecondsPerMillisecond / 100;
}

s = GetDlgItemInt(hwnd, IDC_MINWORKINGSETSIZE, &f, FALSE);
if (f) {
    joeli.BasicLimitInformation.LimitFlags |= JOB_OBJECT_LIMIT_WORKINGSET;
    joeli.BasicLimitInformation.MinimumWorkingSetSize = s * 1024 * 1024;
    s = GetDlgItemInt(hwnd, IDC_MAXWORKINGSETSIZE, &f, FALSE);
    if (f) {
        joeli.BasicLimitInformation.MaximumWorkingSetSize = s * 1024 * 1024;
    } else {
        joeli.BasicLimitInformation.LimitFlags &= ~JOB_OBJECT_LIMIT_WORKINGSET;
        chMB("Both minimum and maximum working set sizes must be set.\n"
             "The working set limits will NOT be in effect.");
    }
}
}

```

см. след. стр.

Рис. 5-6. продолжение

```
d = GetDlgItemInt(hwnd, IDC_ACTIVEPROCESSLIMIT, &f, FALSE);
if (f) {
    joeli.BasicLimitInformation.LimitFlags |=
        JOB_OBJECT_LIMIT_ACTIVE_PROCESS;
    joeli.BasicLimitInformation.ActiveProcessLimit = d;
}

s = GetDlgItemInt(hwnd, IDC_AFFINITYMASK, &f, FALSE);
if (f) {
    joeli.BasicLimitInformation.LimitFlags |= JOB_OBJECT_LIMIT_AFFINITY;
    joeli.BasicLimitInformation.Affinity = s;
}

joeli.BasicLimitInformation.LimitFlags |= JOB_OBJECT_LIMIT_PRIORITY_CLASS;
switch (ComboBox_GetCurSel(GetDlgItem(hwnd, IDC_PRIORITYCLASS))) {
    case 0:
        joeli.BasicLimitInformation.LimitFlags &=
            ~JOB_OBJECT_LIMIT_PRIORITY_CLASS;
        break;

    case 1:
        joeli.BasicLimitInformation.PriorityClass =
            IDLE_PRIORITY_CLASS;
        break;

    case 2:
        joeli.BasicLimitInformation.PriorityClass =
            BELOW_NORMAL_PRIORITY_CLASS;
        break;

    case 3:
        joeli.BasicLimitInformation.PriorityClass =
            NORMAL_PRIORITY_CLASS;
        break;

    case 4:
        joeli.BasicLimitInformation.PriorityClass =
            ABOVE_NORMAL_PRIORITY_CLASS;
        break;

    case 5:
        joeli.BasicLimitInformation.PriorityClass =
            HIGH_PRIORITY_CLASS;
        break;

    case 6:
        joeli.BasicLimitInformation.PriorityClass =
            REALTIME_PRIORITY_CLASS;
        break;
}
```

Рис. 5-6. продолжение

```

int nSchedulingClass =
    ComboBox_GetCurSel(GetDlgItem(hwnd, IDC_SCHEDULINGCLASS));
if (nSchedulingClass > 0) {
    joeli.BasicLimitInformation.LimitFlags |=
        JOB_OBJECT_LIMIT_SCHEDULING_CLASS;
    joeli.BasicLimitInformation.SchedulingClass = nSchedulingClass - 1;
}

s = GetDlgItemInt(hwnd, IDC_MAXCOMMITPERJOB, &f, FALSE);
if (f) {
    joeli.BasicLimitInformation.LimitFlags |= JOB_OBJECT_LIMIT_JOB_MEMORY;
    joeli.JobMemoryLimit = s * 1024 * 1024;
}

s = GetDlgItemInt(hwnd, IDC_MAXCOMMITPERPROCESS, &f, FALSE);
if (f) {
    joeli.BasicLimitInformation.LimitFlags |=
        JOB_OBJECT_LIMIT_PROCESS_MEMORY;
    joeli.ProcessMemoryLimit = s * 1024 * 1024;
}

if (IsDlgButtonChecked(hwnd, IDC_CHILDPROCESSESCANBREAKAWAYFROMJOB))
    joeli.BasicLimitInformation.LimitFlags |= JOB_OBJECT_LIMIT_BREAKAWAY_OK;

if (IsDlgButtonChecked(hwnd, IDC_CHILDPROCESSESDOBREAKAWAYFROMJOB))
    joeli.BasicLimitInformation.LimitFlags |=
        JOB_OBJECT_LIMIT_SILENT_BREAKAWAY_OK;

if (IsDlgButtonChecked(hwnd, IDC_TERMINATEPROCESSONEXCEPTIONS))
    joeli.BasicLimitInformation.LimitFlags |=
        JOB_OBJECT_LIMIT_DIE_ON_UNHANDLED_EXCEPTION;

f = g_job.SetExtendedLimitInfo(&joeli,
    ((joeli.BasicLimitInformation.LimitFlags & JOB_OBJECT_LIMIT_JOB_TIME)
     != 0) ? FALSE :
    IsDlgButtonChecked(hwnd, IDC_PRESERVEJOBTIMEWHENAPPLYINGLIMITS));
chASSERT(f);

// устанавливаем ограничения, связанные с пользовательским интерфейсом
DWORD jobuir = JOB_OBJECT_UILIMIT_NONE; // "замысловатый" нуль (0)
if (IsDlgButtonChecked(hwnd, IDC_RESTRICTACCESSTOUTSIDEUSEROBJECTS))
    jobuir |= JOB_OBJECT_UILIMIT_HANDLES;

if (IsDlgButtonChecked(hwnd, IDC_RESTRICTREADINGCLIPBOARD))
    jobuir |= JOB_OBJECT_UILIMIT_READCLIPBOARD;

if (IsDlgButtonChecked(hwnd, IDC_RESTRICTWRITINGCLIPBOARD))
    jobuir |= JOB_OBJECT_UILIMIT_WRITECLIPBOARD;

if (IsDlgButtonChecked(hwnd, IDC_RESTRICTEXITWINDOW))
    jobuir |= JOB_OBJECT_UILIMIT_EXITWINDOWS;

```

см. след. стр.

Рис. 5-6. продолжение

```
if (IsDlgButtonChecked(hwnd, IDC_RESTRICTCHANGINGSYSTEMPARAMETERS))
    jobuir |= JOB_OBJECT_UILIMIT_SYSTEMPARAMETERS;

if (IsDlgButtonChecked(hwnd, IDC_RESTRICTDESKTOPS))
    jobuir |= JOB_OBJECT_UILIMIT_DESKTOP;

if (IsDlgButtonChecked(hwnd, IDC_RESTRICTDISPLAYSETTINGS))
    jobuir |= JOB_OBJECT_UILIMIT_DISPLAYSETTINGS;

if (IsDlgButtonChecked(hwnd, IDC_RESTRICTGLOBALATOMS))
    jobuir |= JOB_OBJECT_UILIMIT_GLOBALATOMS;

chVERIFY(g_job.SetBasicUIRestrictions(jobuir));
}

///////////////////////////////
void Dlg_OnCommand(HWND hwnd, int id, HWND hwndCtl, UINT codeNotify) {

    switch (id) {
        case IDCANCEL:
            // пользователь закрывает наше приложение – закрываем и задание
            KillTimer(hwnd, 1);
            g_job.Terminate(0);
            EndDialog(hwnd, id);
            break;

        case IDC_PERJOBUSERTIMELIMIT:
            {
                // устанавливая новое ограничение по времени,
                // нужно сбросить ранее указанное время для задания
                BOOL f;
                GetDlgItemInt(hwnd, IDC_PERJOBUSERTIMELIMIT, &f, FALSE);
                EnableWindow(
                    GetDlgItem(hwnd, IDC_PRESERVEJOBTIMEWHENAPPLYINGLIMITS), !f);
            }
            break;

        case IDC_APPLYLIMITS:
            Dlg_ApplyLimits(hwnd);
            PostQueuedCompletionStatus(g_hIOCP, 0, COMPKEY_STATUS, NULL);
            break;

        case IDC_TERMINATE:
            g_job.Terminate(0);
            PostQueuedCompletionStatus(g_hIOCP, 0, COMPKEY_STATUS, NULL);
            break;

        case IDC_SPAWNCMDINJOB:
            {
                // порождаем процесс командного процессора и помещаем его в задание
```

Рис. 5-6. продолжение

```

STARTUPINFO si = { sizeof(si) };
PROCESS_INFORMATION pi;
TCHAR sz[] = TEXT("CMD");
CreateProcess(NULL, sz, NULL, NULL,
    FALSE, CREATE_SUSPENDED, NULL, NULL, &si, &pi);
g_job.AssignProcess(pi.hProcess);
ResumeThread(pi.hThread);
CloseHandle(pi.hProcess);
CloseHandle(pi.hThread);
}
PostQueuedCompletionStatus(g_hIOCP, 0, COMPKEY_STATUS, NULL);
break;

case IDC_ASSIGNPROCESSTOJOB:
{
    DWORD dwProcessId = GetDlgItemInt(hwnd, IDC_PROCESSID, NULL, FALSE);
    HANDLE hProcess = OpenProcess(
        PROCESS_SET_QUOTA | PROCESS_TERMINATE, FALSE, dwProcessId);
    if (hProcess != NULL) {
        chVERIFY(g_job.AssignProcess(hProcess));
        CloseHandle(hProcess);
    } else chMB("Could not assign process to job.");
}
PostQueuedCompletionStatus(g_hIOCP, 0, COMPKEY_STATUS, NULL);
break;
}

///////////////////////////////
void WINAPI Dlg_OnTimer(HWND hwnd, UINT id) {
    PostQueuedCompletionStatus(g_hIOCP, 0, COMPKEY_STATUS, NULL);
}

/////////////////////////////
INT_PTR WINAPI Dlg_Proc(HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam) {
    switch (uMsg) {
        chHANDLE_DLGMMSG(hwnd, WM_INITDIALOG, Dlg_OnInitDialog);
        chHANDLE_DLGMMSG(hwnd, WM_TIMER, Dlg_OnTimer);
        chHANDLE_DLGMMSG(hwnd, WM_COMMAND, Dlg_OnCommand);
    }
    return(FALSE);
}

/////////////////////////////

```

см. след. стр.

Рис. 5-6. продолжение

```
int WINAPI _tWinMain(HINSTANCE hinstExe, HINSTANCE, LPTSTR pszCmdLine, int) {  
  
    // создаем порт завершения, который будет принимать уведомления от задания  
    g_hIOCP = CreateIoCompletionPort(INVALID_HANDLE_VALUE, NULL, 0, 0);  
  
    // создаем поток, ждущий порт завершения  
    g_hThreadIOCP = chBEGINTHREADEX(NULL, 0, JobNotify, NULL, 0, NULL);  
  
    // создаем объект-задание  
    g_job.Create(NULL, TEXT("JobLab"));  
    g_job.SetEndOfJobInfo(JOB_OBJECT_POST_AT_END_OF_JOB);  
    g_job.AssociateCompletionPort(g_hIOCP, COMPKEY_JOBOBJECT);  
  
    DialogBox(hinstExe, MAKEINTRESOURCE(IDD_JOBLAB), NULL, Dlg_Proc);  
  
    // передаем специальный ключ, заставляющий поток  
    // порта завершения закончить работу  
    PostQueuedCompletionStatus(g_hIOCP, 0, COMPKEY_TERMINATE, NULL);  
  
    // ждем, когда завершится его поток  
    WaitForSingleObject(g_hThreadIOCP, INFINITE);  
  
    // проводим должную очистку  
    CloseHandle(g_hIOCP);  
    CloseHandle(g_hThreadIOCP);  
  
    // ПРИМЕЧАНИЕ: задание закрывается вызовом деструктора g_job  
    return(0);  
}  
  
//////////////////////////// Конец файла //////////////////////////////
```

Job.h

```
*****  
Модуль: Job.h  
Автор: Copyright (c) 2000, Джейфри Рихтер (Jeffrey Richter)  
*****  
  
#pragma once  
  
//////////////////////////////  
  
#include <malloc.h>      // для доступа к _alloca  
  
//////////////////////////////  
  
class CJob {  
public:  
    CJob(HANDLE hJob = NULL);  
    ~CJob();
```

Рис. 5-6. продолжение

```

operator HANDLE() const { return(m_hJob); }

// функции, создающие или открывающие объект-задание
BOOL Create(LPSECURITY_ATTRIBUTES psa = NULL, LPCTSTR pszName = NULL);
BOOL Open(LPCTSTR pszName, DWORD dwDesiredAccess,
          BOOL fInheritHandle = FALSE);

// функции, манипулирующие объектом-заданием
BOOL AssignProcess(HANDLE hProcess);
BOOL Terminate(UINT uExitCode = 0);

// функции, налагающие ограничения на задания
BOOL SetExtendedLimitInfo(PJOBOBJECT_EXTENDED_LIMIT_INFORMATION pjoeli,
                           BOOL fPreserveJobTime = FALSE);
BOOL SetBasicUIRestrictions(DWORD fdwLimits);
BOOL GrantUserHandleAccess(HANDLE hUserObj, BOOL fGrant = TRUE);
BOOL SetSecurityLimitInfo(PJOBOBJECT_SECURITY_LIMIT_INFORMATION pjosli);

// функции, запрашивающие сведения об ограничениях
BOOL QueryExtendedLimitInfo(PJOBOBJECT_EXTENDED_LIMIT_INFORMATION pjoeli);
BOOL QueryBasicUIRestrictions(PDWORD pfdwRestrictions);
BOOL QuerySecurityLimitInfo(PJOBOBJECT_SECURITY_LIMIT_INFORMATION pjosli);

// функции, запрашивающие статусную информацию о задании
BOOL QueryBasicAccountingInfo(
    PJOBOBJECT_BASIC_AND_IO_ACCOUNTING_INFORMATION pjobai);
BOOL QueryBasicProcessIdList(DWORD dwMaxProcesses,
                             PDWORD pdwProcessIdList, PDWORD pdwProcessesReturned = NULL);

// функции, работающие с уведомлениями задания
BOOL AssociateCompletionPort(HANDLE hIOCP, ULONG_PTR CompKey);
BOOL QueryAssociatedCompletionPort(
    PJOBOBJECT_ASSOCIATE_COMPLETION_PORT pjoacp);
BOOL SetEndOfJobInfo(
    DWORD fdwEndOfJobInfo = JOB_OBJECT_TERMINATE_AT_END_OF_JOB);
BOOL QueryEndOfJobTimeInfo(PDWORD pfdwEndOfJobTimeInfo);

private:
    HANDLE m_hJob;
};

/////////////////////////////// inline CJob::CJob(HANDLE hJob) {
    m_hJob = hJob;
}

///////////////////////////////

```

см. след. стр.

Рис. 5-6. продолжение

```
inline CJob::~CJob() {

    if (m_hJob != NULL)
        CloseHandle(m_hJob);
}

///////////////////////////////



inline BOOL CJob::Create(PSECURITY_ATTRIBUTES psa, PCTSTR pszName) {

    m_hJob = CreateJobObject(psa, pszName);
    return(m_hJob != NULL);
}

///////////////////////////////



inline BOOL CJob::Open(
    PCTSTR pszName, DWORD dwDesiredAccess, BOOL fInheritHandle) {

    m_hJob = OpenJobObject(dwDesiredAccess, fInheritHandle, pszName);
    return(m_hJob != NULL);
}

///////////////////////////////



inline BOOL CJob::AssignProcess(HANDLE hProcess) {

    return(AssignProcessToJobObject(m_hJob, hProcess));
}

///////////////////////////////



inline BOOL CJob::AssociateCompletionPort(HANDLE hIOCP, ULONG_PTR CompKey) {

    JOBOBJECT_ASSOCIATE_COMPLETION_PORT joacp = { (PVOID) CompKey, hIOCP };
    return(SetInformationJobObject(m_hJob,
        JobObjectAssociateCompletionPortInformation, &joacp, sizeof(joacp)));
}

///////////////////////////////



inline BOOL CJob::SetExtendedLimitInfo(
    PJOBOBJECT_EXTENDED_LIMIT_INFORMATION pjoeli, BOOL fPreserveJobTime) {

    if (fPreserveJobTime)
        pjoeli->BasicLimitInformation.LimitFlags |=
            JOB_OBJECT_LIMIT_PRESERVE_JOB_TIME;

    // если Вы хотите сохранить информацию о времени задания,
    // флаг JOB_OBJECT_LIMIT_JOB_TIME нужно убрать
    const DWORD fdwFlagTest =
```

Рис. 5-6. продолжение

```

(JOB_OBJECT_LIMIT_PRESERVE_JOB_TIME | JOB_OBJECT_LIMIT_JOB_TIME);

if ((pjoeli->BasicLimitInformation.LimitFlags & fdwFlagTest)
    == fdwFlagTest) {
    // ошибка, так как указаны два взаимоисключающих флага
    DebugBreak();
}

return(SetInformationJobObject(m_hJob,
    JobObjectExtendedLimitInformation, pjoeli, sizeof(*pjoeli)));
}

///////////////////////////////
inline BOOL CJob::SetBasicUIRestrictions(DWORD fdwLimits) {

    JOBOBJECT_BASIC_UI_RESTRICTIONS jobuir = { fdwLimits };
    return(SetInformationJobObject(m_hJob,
        JobObjectBasicUIRestrictions, &jobuir, sizeof(jobuir)));
}

///////////////////////////////
inline BOOL CJob::SetEndOfJobInfo(DWORD fdwEndOfJobInfo) {

    JOBOBJECT_END_OF_JOB_TIME_INFORMATION joeoji = { fdwEndOfJobInfo };
    joeoji.EndOfJobTimeAction = fdwEndOfJobInfo;
    return(SetInformationJobObject(m_hJob,
        JobObjectEndOfJobTimeInformation, &joeoji, sizeof(joeoji)));
}

///////////////////////////////
inline BOOL CJob::SetSecurityLimitInfo(
    PJOBOBJECT_SECURITY_LIMIT_INFORMATION pjosli) {

    return(SetInformationJobObject(m_hJob,
        JobObjectSecurityLimitInformation, pjosli, sizeof(*pjosl)));
}

///////////////////////////////
inline BOOL CJob::QueryAssociatedCompletionPort(
    PJOBOBJECT_ASSOCIATE_COMPLETION_PORT pjoacp) {

    return(QueryInformationJobObject(m_hJob,
        JobObjectAssociateCompletionPortInformation, pjoacp, sizeof(*pjoacp),
        NULL));
}

/////////////////////////////

```

см. след. стр.

Рис. 5-6. продолжение

```
inline BOOL CJob::QueryBasicAccountingInfo(
    PJOBOBJECT_BASIC_AND_IO_ACCOUNTING_INFORMATION pjobai) {

    return(QueryInformationJobObject(m_hJob,
        JobObjectBasicAndIoAccountingInformation, pjobai, sizeof(*pjobai),
        NULL));
}

///////////////////////////////



inline BOOL CJob::QueryExtendedLimitInfo(
    PJOBOBJECT_EXTENDED_LIMIT_INFORMATION pjoeli) {

    return(QueryInformationJobObject(m_hJob, JobObjectExtendedLimitInformation,
        pjoeli, sizeof(*pjoeli), NULL));
}

///////////////////////////////



inline BOOL CJob::QueryBasicProcessIdList(DWORD dwMaxProcesses,
    PDWORD pdwProcessIdList, PDWORD pdwProcessesReturned) {

    // определяем требуемый объем памяти в байтах
    DWORD cb = sizeof(JOBOBJECT_BASIC_PROCESS_ID_LIST) +
        (sizeof(DWORD) * (dwMaxProcesses - 1));

    // выделяем эти байты из стека
    PJOBOBJECT_BASIC_PROCESS_ID_LIST pjobpil =
        (PJOBOBJECT_BASIC_PROCESS_ID_LIST) _alloca(cb);

    // Успешно ли прошла эта операция? Если да, идем дальше.
    BOOL fOk = (pjobpil != NULL);

    if (fOk) {
        pjobpil->NumberofProcessIdsInList = dwMaxProcesses;
        fOk = ::QueryInformationJobObject(m_hJob, JobObjectBasicProcessIdList,
            pjobpil, cb, NULL);

        if (fOk) {
            // у нас появилась информация, возвращаем ее тому, кто ее запрашивал
            if (pdwProcessesReturned != NULL)
                *pdwProcessesReturned = pjobpil->NumberofProcessIdsInList;

            CopyMemory(pdwProcessIdList, pjobpil->ProcessIdList,
                sizeof(DWORD) * pjobpil->NumberofProcessIdsInList);
        }
    }
    return(fOk);
}

///////////////////////////////
```

Рис. 5-6. продолжение

```

inline BOOL CJob::QueryBasicUIRestrictions(PDWORD pfdwRestrictions) {

    JOBOBJECT_BASIC_UI_RESTRICTIONS jobuir;
    BOOL fOk = QueryInformationJobObject(m_hJob, JobObjectBasicUIRestrictions,
        &jobuir, sizeof(jobuir), NULL);
    if (fOk)
        *pfdwRestrictions = jobuir.UIRestrictionsClass;
    return(fOk);
}

/////////////////////////////// Конец файла ///////////////////////////////

```

```

inline BOOL CJob::QueryEndOfJobTimeInfo(PDWORD pfdwEndOfJobTimeInfo) {

    JOBOBJECT_END_OF_JOB_TIME_INFORMATION joeoji;
    BOOL fOk = QueryInformationJobObject(m_hJob, JobObjectBasicUIRestrictions,
        &joeoji, sizeof(joeoji), NULL);
    if (fOk)
        *pfdwEndOfJobTimeInfo = joeoji.EndOfJobTimeAction;
    return(fOk);
}

/////////////////////////////// Конец файла ///////////////////////////////

```

```

inline BOOL CJob::QuerySecurityLimitInfo(
    PJOBOBJECT_SECURITY_LIMIT_INFORMATION pjosli) {

    return(QueryInformationJobObject(m_hJob, JobObjectSecurityLimitInformation,
        pjosli, sizeof(*pjosl), NULL));
}

```

```

/////////////////////////////// Конец файла ///////////////////////////////

```

```

inline BOOL CJob::Terminate(UINT uExitCode) {

    return(TerminateJobObject(m_hJob, uExitCode));
}

```

```

/////////////////////////////// Конец файла ///////////////////////////////

```

```

inline BOOL CJob::GrantUserHandleAccess(HANDLE hUserObj, BOOL fGrant) {

    return(UserHandleGrantAccess(hUserObj, m_hJob, fGrant));
}

```

```

/////////////////////////////// Конец файла ///////////////////////////////

```

Базовые сведения о потоках

Тематика, связанная потоками, очень важна, потому что в любом процессе должен быть хотя бы один поток. В этой главе концепции потоков будут рассмотрены гораздо подробнее. В частности, я объясню, в чем разница между процессами и потоками и для чего они предназначены. Также я расскажу о том, как система использует объекты ядра «поток» для управления потоками. Подобно процессам, потоки обладают определенными свойствами, поэтому мы поговорим о функциях, позволяющих обращаться к этим свойствам и при необходимости модифицировать их. Кроме того, Вы узнаете о функциях, предназначенных для создания (порождения) дополнительных потоков в системе.

В главе 4 я говорил, что процесс фактически состоит из двух компонентов: объекта ядра «процесс» и адресного пространства. Так вот, любой поток тоже состоит из двух компонентов:

- объекта ядра, через который операционная система управляет потоком. Там же хранится статистическая информация о потоке;
- стека потока, который содержит параметры всех функций и локальные переменные, необходимые потоку для выполнения кода. (О том, как система управляет стеком потока, я расскажу в главе 16.)

В той же главе 4 я упомянул, что процессы инертны. Процесс ничего не исполняет, он просто служит контейнером потоков. Потоки всегда создаются в контексте какого-либо процесса, и вся их жизнь проходит только в его границах. На практике это означает, что потоки исполняют код и манипулируют данными в адресном пространстве процесса. Поэтому, если два и более потоков выполняется в контексте одного процесса, все они делят одно адресное пространство. Потоки могут исполнять один и тот же код и манипулировать одними и теми же данными, а также совместно использовать описатели объектов ядра, поскольку таблица описателей создается не в отдельных потоках, а в процессах.

Как видите, процессы используют куда больше системных ресурсов, чем потоки. Причина кроется в адресном пространстве. Создание виртуального адресного пространства для процесса требует значительных системных ресурсов. При этом ведется масса всяческой статистики, на что уходит немало памяти. В адресное пространство загружаются EXE- и DLL-файлы, а значит, нужны файловые ресурсы. С другой стороны, потоку требуются лишь соответствующий объект ядра и стек; объем статистических сведений о потоке невелик и много памяти не занимает.

Так как потоки расходуют существенно меньше ресурсов, чем процессы, старайтесь решать свои задачи за счет использования дополнительных потоков и избегайте создания новых процессов. Только не принимайте этот совет за жесткое правило — многие проекты как раз лучше реализовать на основе множества процессов. Нужно просто помнить об издержках и соразмерять цель и средства.

Прежде чем мы углубимся в скучные, но крайне важные концепции, давайте обсудим, как правильно пользоваться потоками, разрабатывая архитектуру приложения.

В каких случаях потоки создаются

Поток (thread) определяет последовательность исполнения кода в процессе. При инициализации процесса система всегда создает первичный поток. Начинаясь со стартового кода из библиотеки C/C++, который в свою очередь вызывает входную функцию (*WinMain*, *wWinMain*, *main* или *wmain*) из Вашей программы, он живет до того момента, когда входная функция возвращает управление стартовому коду и тот вызывает функцию *ExitProcess*. Большинство приложений обходится единственным, первичным потоком. Однако процессы могут создавать дополнительные потоки, что позволяет им эффективнее выполнять свою работу.

У каждого компьютера есть чрезвычайно мощный ресурс — центральный процессор. И нет абсолютно никаких причин тому, чтобы этот процессор простоявал (не считая экономии электроэнергии). Чтобы процессор всегда был при деле, Вы нагружаете его самыми разнообразными задачами. Вот несколько примеров.

- Вы активизируете службу индексации данных (content indexing service) Windows 2000. Она создает поток с низким приоритетом, который, периодически пробуждаясь, индексирует содержимое файлов на дисковых устройствах Вашего компьютера. Чтобы найти какой-либо файл, Вы открываете окно Search Results (щелкнув кнопку Start и выбрав из меню Search команду For Files Or Folders) и вводите в поле Containing Text нужные критерии поиска. После этого начинается поиск по индексу, и на экране появляется список файлов, удовлетворяющих этим критериям. Служба индексации данных значительно увеличивает скорость поиска, так как при ее использовании больше не требуется открывать, сканировать и закрывать каждый файл на диске.
- Вы запускаете программу для дефрагментации дисков, поставляемую с Windows 2000. Обычно утилиты такого рода предлагают массу настроек для администрирования, в которых средний пользователь совершенно не разбирается, — например, когда и как часто проводить дефрагментацию. Благодаря потокам с более низким приоритетом Вы можете пользоваться этой программой в фоновом режиме и дефрагментировать диски в те моменты, когда других дел у системы нет.
- Нетрудно представить будущую версию компилятора, способную автоматически компилировать файлы исходного кода в паузах, возникающих при наборе текста программы. Тогда предупреждения и сообщения об ошибках появлялись бы практически в режиме реального времени, и Вы тут же видели бы, в чем Вы ошиблись. Самое интересное, что Microsoft Visual Studio в какой-то мере уже умеет это делать, — обратите внимание на секцию ClassView в Workspace.
- Электронные таблицы пересчитывают данные в фоновом режиме.
- Текстовые процессоры разбивают текст на страницы, проверяют его на орографические и грамматические ошибки, а также печатают в фоновом режиме.
- Файлы можно копировать на другие носители тоже в фоновом режиме.
- Web-браузеры способны взаимодействовать с серверами в фоновом режиме. Благодаря этому пользователь может перейти на другой Web-узел, не дожидаясь, когда будут получены результаты с текущего Web-узла.

Одна важная вещь, на которую Вы должны были обратить внимание во всех этих примерах, заключается в том, что поддержка многопоточности позволяет упростить пользовательский интерфейс приложения. Если компилятор ведет сборку Вашей программы в те моменты, когда Вы делаете паузы в наборе ее текста, отпадает необходимость в командах меню Build. То же самое относится к командам Check Spelling и Check Grammar в текстовых процессорах.

В примере с Web-браузером выделение ввода-вывода (сетевого, файлового или какого-то другого) в отдельный поток обеспечивает « отзывчивость » пользовательского интерфейса приложения даже при интенсивной передаче данных. Вообразите приложение, которое сортирует записи в базе данных, печатает документ или копирует файлы. Возложив любую из этих задач, так или иначе связанных с вводом-выводом, на отдельный поток, пользователь может по-прежнему работать с интерфейсом приложения и при необходимости отменить операцию, выполняемую в фоновом режиме.

Многопоточное приложение легче масштабируется. Как Вы увидите в следующей главе, каждый поток можно закрепить за определенным процессором. Так что, если в Вашем компьютере имеется два процессора, а в приложении — два потока, оба процессора будут при деле. И фактически Вы сможете выполнять две задачи одновременно.

В каждом процессе есть хотя бы один поток. Даже не делая ничего особенного в приложении, Вы уже выигрываете только от того, что оно выполняется в многопоточной операционной системе. Например, Вы можете собирать программу и одновременно пользоваться текстовым процессором (довольно часто я так и работаю). Если в компьютере установлено два процессора, то сборка выполняется на одном из них, а документ обрабатывается на другом. Иначе говоря, какого-либо падения производительности не наблюдается. И кроме того, если компилятор из-за той или иной ошибки входит в бесконечный цикл, на остальных процессах это никак не отражается. (Конечно, о программах для MS-DOS и 16-разрядной Windows речь не идет.)

И в каких случаях потоки не создаются

До сих пор я пел одни дифирамбы многопоточным приложениям. Но, несмотря на все преимущества, у них есть и свои недостатки. Некоторые разработчики почему-то считают, будто *любую* проблему можно решить, разбив программу на отдельные потоки. Трудно совершить большую ошибку!

Потоки — вещь невероятно полезная, когда ими пользуются с умом. Увы, решая старые проблемы, можно создать себе новые. Допустим, Вы разрабатываете текстовый процессор и хотите выделить функциональный блок, отвечающий за распечатку, в отдельный поток. Идея вроде неплоха: пользователь, отправив документ на распечатку, может сразу вернуться к редактированию. Но задумайтесь вот над чем: значит, информация в документе может быть изменена *при* распечатке документа? Как видите, теперь перед Вами совершенно новая проблема, с которой прежде сталкиваться не приходилось. Тут-то и подумаешь, а стоит ли выделять печать в отдельный поток, зачем искать лишних приключений? Но давайте разрешим при распечатке редактирование любых документов, кроме того, который печатается в данный момент. Или так: скопируем документ во временный файл и отправим на печать именно его, а пользователь пусть редактирует оригинал в свое удовольствие. Когда распечатка временного файла закончится, мы его удалим — вот и все.

Еще одно узкое место, где неправильное применение потоков может привести к появлению проблем, — разработка пользовательского интерфейса в приложении. В подавляющем большинстве программ все компоненты пользовательского интерфей-

са (окна) обрабатываются одним и тем же потоком. И дочерние окна любого окна определенно должен создавать только один поток. Создание разных окон в разных потоках иногда имеет смысл, но такие случаи действительно редки.

Обычно в приложении существует один поток, отвечающий за поддержку пользовательского интерфейса, — он создает все окна и содержит цикл *GetMessage*. Любые другие потоки в процессе являются рабочими (т. е. отвечают за вычисления, ввод-вывод и другие операции) и не создают никаких окон. Поток пользовательского интерфейса, как правило, имеет более высокий приоритет, чем рабочие потоки, — это нужно для того, чтобы он всегда быстро реагировал на действия пользователя.

Несколько потоков пользовательского интерфейса в одном процессе можно обнаружить в таких приложениях, как Windows Explorer. Он создает отдельный поток для каждого окна папки. Это позволяет копировать файлы из одной папки в другую и попутно просматривать содержимое еще какой-то папки. Кроме того, если какая-то ошибка в Explorer приводит к краху одного из его потоков, прочие потоки остаются работоспособны, и Вы можете пользоваться соответствующими окнами, пока не сделаете что-нибудь такое, из-за чего рухнут и они. (Подробнее о потоках и пользовательском интерфейсе см. главы 26 и 27.)

В общем, мораль этого вступления такова: многопоточность следует использовать разумно. Не создавайте несколько потоков только потому, что это возможно. Многие полезные и мощные программы по-прежнему строятся на основе одного первично-го потока, принадлежащего процессу.

Ваша первая функция потока

Каждый поток начинает выполнение с некоей входной функции. В первичном пото-ке таковой является *main*, *wmain*, *WinMain* или *wWinMain*. Если Вы хотите создать вторичный поток, в нем тоже должна быть входная функция, которая выглядит пример-но так:

```
DWORD WINAPI ThreadFunc(PVOID pvParam) {
    DWORD dwResult = 0;
    :
    return(dwResult);
}
```

Функция потока может выполнять любые задачи. Рано или поздно она закончит свою работу и вернет управление. В этот момент Ваш поток остановится, память, от-веденная под его стек, будет освобождена, а счетчик пользователей его объекта ядра «поток» уменьшится на 1. Когда счетчик обнулится, этот объект ядра будет разрушен. Но, как и объект ядра «процесс», он может жить гораздо дольше, чем сопоставленный с ним поток.

А теперь поговорим о самых важных вещах, касающихся функций потоков.

- В отличие от входной функции первичного потока, у которой должно быть одно из четырех имен: *main*, *wmain*, *WinMain* или *wWinMain*, — функцию пото-ка можно назвать как угодно. Однако, если в программе несколько функций потоков, Вы должны присвоить им разные имена, иначе компилятор или компоновщик решит, что Вы создаете несколько реализаций единственной функции.
- Поскольку входным функциям первичного потока передаются строковые па-раметры, они существуют в ANSI- и Unicode-версиях: *main* — *wmain* и *WinMain* —

wWinMain. Но функциям потоков передается единственный параметр, смысл которого определяется Вами, а не операционной системой. Поэтому здесь нет проблем с ANSI/Unicode.

- Функция потока должна возвращать значение, которое будет использоваться как код завершения потока. Здесь полная аналогия с библиотекой C/C++: код завершения первичного потока становится кодом завершения процесса.
- Функции потоков (да и все Ваши функции) должны по мере возможности обходиться своими параметрами и локальными переменными. Так как к статической или глобальной переменной могут одновременно обратиться несколько потоков, есть риск повредить ее содержимое. Однако параметры и локальные переменные создаются в стеке потока, поэтому они в гораздо меньшей степени подвержены влиянию другого потока.

Вот Вы и узнали, как должна быть реализована функция потока. Теперь рассмотрим, как заставить операционную систему создать поток, который выполнит эту функцию.

Функция *CreateThread*

Мы уже говорили, как при вызове функции *CreateProcess* появляется на свет первичный поток процесса. Если Вы хотите создать дополнительные потоки, нужно вызвать из первичного потока функцию *CreateThread*:

```
HANDLE CreateThread(  
    PSECURITY_ATTRIBUTES psa;  
    DWORD cbStack;  
    PTHREAD_START_ROUTINE pfnStartAddr;  
    PVOID pvParam;  
    DWORD fdwCreate;  
    PDWORD pdwThreadID);
```

При каждом вызове этой функции система создает объект ядра «поток». Это не сам поток, а компактная структура данных, которая используется операционной системой для управления потоком и хранит статистическую информацию о потоке. Так что объект ядра «поток» — полный аналог объекта ядра «процесс».

Система выделяет память под стек потока из адресного пространства процесса. Новый поток выполняется в контексте того же процесса, что и родительский поток. Поэтому он получает доступ ко всем описателям объектов ядра, всей памяти и стекам всех потоков в процессе. За счет этого потоки в рамках одного процесса могут легко взаимодействовать друг с другом.



CreateThread — это Windows-функция, создающая поток. Но никогда не вызывайте ее, если Вы пишете код на C/C++. Вместо нее Вы должны использовать функцию *_beginthreadex* из библиотеки Visual C++. (Если Вы работаете с другим компилятором, он должен поддерживать свой эквивалент функции *CreateThread*.) Что именно делает *_beginthreadex* и почему это так важно, я объясню потом.

О'кэй, общее представление о функции *CreateThread* Вы получили. Давайте рассмотрим все ее параметры.

Параметр *psa*

Параметр *psa* является указателем на структуру SECURITY_ATTRIBUTES. Если Вы хотите, чтобы объекту ядра «поток» были присвоены атрибуты защиты по умолчанию (что чаще всего и бывает), передайте в этом параметре NULL. А чтобы дочерние процессы смогли наследовать описатель этого объекта, определите структуру SECURITY_ATTRIBUTES и инициализируйте ее элемент *bInheritHandle* значением TRUE (см. главу 3).

Параметр *cbStack*

Этот параметр определяет, какую часть адресного пространства поток сможет использовать под свой стек. Каждому потоку выделяется отдельный стек. Функция *CreateProcess*, запуская приложение, вызывает *CreateThread*, и та инициализирует первичный поток процесса. При этом *CreateProcess* заносит в параметр *cbStack* значение, хранящееся в самом исполняемом файле. Управлять этим значением позволяет ключ /STACK компоновщика:

/STACK:[*reserve*] [,*commit*]

Аргумент *reserve* определяет объем адресного пространства, который система должна зарезервировать под стек потока (по умолчанию — 1 Мб). Аргумент *commit* задает объем физической памяти, который изначально передается области, зарезервированной под стек (по умолчанию — 1 страница). По мере исполнения кода в потоке Вам, весьма вероятно, понадобится отвести под стек больше одной страницы памяти. При переполнении стека возникнет исключение. (О стеке потока и исключениях, связанных с его переполнением, см. главу 16, а об общих принципах обработки исключений — главу 23.) Перехватив это исключение, система передаст зарезервированному пространству еще одну страницу (или столько, сколько указано в аргументе *commit*). Такой механизм позволяет динамически увеличивать размер стека лишь по необходимости.

Если Вы, обращаясь к *CreateThread*, передаете в параметре *cbStack* ненулевое значение, функция резервирует всю указанную Вами память. Ее объем определяется либо значением параметра *cbStack*, либо значением, заданным в ключе /STACK компоновщика (выбирается большее из них). Но передается стеку лишь тот объем памяти, который соответствует значению в *cbStack*. Если же Вы передаете в параметре *cbStack* нулевое значение, *CreateThread* создает стек для нового потока, используя информацию, встроенную компоновщиком в EXE-файл.

Значение аргумента *reserve* устанавливает верхний предел для стека, и это ограничение позволяет прекращать деятельность функций с бесконечной рекурсией. Допустим, Вы пишете функцию, которая рекурсивно вызывает сама себя. Предположим также, что в функции есть «жучок», приводящий к бесконечной рекурсии. Всякий раз, когда функция вызывает сама себя, в стеке создается новый стековый фрейм. Если бы система не позволяла ограничивать максимальный размер стека, рекурсивная функция так и вызывала бы сама себя до бесконечности, а стек поглотил бы все адресное пространство процесса. Задавая же определенный предел, Вы, во-первых, предотвращаете разрастание стека до гигантских объемов и, во-вторых, гораздо быстрее узнаете о наличии ошибки в своей программе. (Программа-пример Summation в главе 16 продемонстрирует, как перехватывать и обрабатывать переполнение стека в приложениях.)

Параметры *pfnStartAddr* и *pvParam*

Параметр *pfnStartAddr* определяет адрес функции потока, с которой должен будет начать работу создаваемый поток, а параметр *pvParam* идентичен параметру *pvParam* функции потока. *CreateThread* лишь передает этот параметр по эстафете той функции, с которой начинается выполнение создаваемого потока. Таким образом, данный параметр позволяет передавать функции потока какое-либо инициализирующее значение. Оно может быть или просто числовым значением, или указателем на структуру данных с дополнительной информацией.

Вполне допустимо и даже полезно создавать несколько потоков, у которых в качестве входной точки используется адрес одной и той же функции. Например, можно реализовать Web-сервер, который обрабатывает каждый клиентский запрос в отдельном потоке. При создании каждому потоку передается свое значение *pvParam*.

Учтите, что Windows — операционная система с вытесняющей многозадачностью, а следовательно, новый поток и поток, вызвавший *CreateThread*, могут выполняться одновременно. В связи с этим возможны проблемы. Остерегайтесь, например, такого кода:

```
DWORD WINAPI FirstThread(PVOID pvParam) {
    // инициализируем переменную, которая содержится в стеке
    int x = 0;
    DWORD dwThreadId;

    // создаем новый поток
    HANDLE hThread = CreateThread(NULL, 0, SecondThread, (PVOID) &x,
        0, &dwThreadId);

    // мы больше не ссылаемся на новый поток,
    // поэтому закрываем свой описатель этого потока
    CloseHandle(hThread);

    // Наш поток закончил работу.
    // ОШИБКА: его стек будет разрушен, но SecondThread
    // может попытаться обратиться к нему.
    return(0);
}

DWORD WINAPI SecondThread(PVOID pvParam) {
    // здесь выполняется какая-то длительная обработка
    :
    // Пытаемся обратиться к переменной в стеке FirstThread.
    // ПРИМЕЧАНИЕ: это может привести к ошибке общей защиты –
    // нарушению доступа!
    * ((int *) pvParam) = 5;
    :
    return(0);
}
```

Не исключено, что в приведенном коде *FirstThread* закончит свою работу до того, как *SecondThread* присвоит значение 5 переменной *x* из *FirstThread*. Если так и будет, *SecondThread* не узнает, что *FirstThread* больше не существует, и попытается изменить содержимое какого-то участка памяти с недействительным теперь адресом. Это неизбежно вызовет нарушение доступа: стек первого потока уничтожен по завершении

FirstThread. Что же делать? Можно объявить *x* статической переменной, и компилятор отведет память для хранения переменной *x* не в стеке, а в разделе данных приложения (application's data section). Но тогда функция станет нереентрабельной. Иначе говоря, в этом случае Вы не смогли бы создать два потока, выполняющих одну и ту же функцию, так как оба потока совместно использовали бы статическую переменную. Другое решение этой проблемы (и его более сложные варианты) базируется на методах синхронизации потоков, речь о которых пойдет в главах 8, 9 и 10.

Параметр *fdwCreate*

Этот параметр определяет дополнительные флаги, управляющие созданием потока. Он принимает одно из двух значений: 0 (исполнение потока начинается немедленно) или CREATE_SUSPENDED. В последнем случае система создает поток, инициализирует его и приостанавливает до последующих указаний.

Флаг CREATE_SUSPENDED позволяет программе изменить какие-либо свойства потока перед тем, как он начнет выполнять код. Правда, необходимость в этом возникает довольно редко. Одно из применений этого флага демонстрирует программа-пример JobLab из главы 5.

Параметр *pdwThreadID*

Последний параметр функции *CreateThread* — это адрес переменной типа DWORD, в которой функция возвращает идентификатор, приписанный системой новому потоку. (Идентификаторы процессов и потоков рассматривались в главе 4.)



В Windows 2000 и Windows NT 4 в этом параметре можно передавать NULL (обычно так и делается). Тем самым Вы сообщаете функции, что Вас не интересует идентификатор потока. Но в Windows 95/98 это приведет к ошибке, так как функция попытается записать идентификатор потока по нулевому адресу, что недопустимо. И поток не будет создан.

Такое несоответствие между операционными системами может создать разработчикам приложений массу проблем. Допустим, Вы пишете и тестируете программу в Windows 2000 (которая создает поток, даже если Вы передаете NULL в *pdwThreadID*). Но вот Вы запускаете приложение в Windows 98, и функция *CreateThread*, естественно, дает ошибку. Вывод один: тщательно тестируйте свое приложение во всех операционных системах, в которых оно будет работать.

Завершение потока

Поток можно завершить четырьмя способами:

- функция потока возвращает управление (рекомендуемый способ);
- поток самоуничтожается вызовом функции *ExitThread* (нежелательный способ);
- один из потоков данного или стороннего процесса вызывает функцию *TerminateThread* (нежелательный способ);
- завершается процесс, содержащий данный поток (тоже нежелательно).

В этом разделе мы обсудим перечисленные способы завершения потока, а также рассмотрим, что на самом деле происходит в момент его окончания.

Возврат управления функцией потока

Функцию потока следует проектировать так, чтобы поток завершался только после того, как она возвращает управление. Это единственный способ, гарантирующий корректную очистку всех ресурсов, принадлежавших Вашему потоку. При этом:

- любые C++-объекты, созданные данным потоком, уничтожаются соответствующими деструкторами;
- система корректно освобождает память, которую занимал стек потока;
- система устанавливает код завершения данного потока (поддерживаемый объектом ядра «поток») — его и возвращает Ваша функция потока;
- счетчик пользователей данного объекта ядра «поток» уменьшается на 1.

Функция *ExitThread*

Поток можно завершить принудительно, вызвав:

```
VOID ExitThread(DWORD dwExitCode);
```

При этом освобождаются все ресурсы операционной системы, выделенные данному потоку, но C/C++-ресурсы (например, объекты, созданные из C++-классов) не очищаются. Именно поэтому лучше возвращать управление из функции потока, чем самому вызывать функцию *ExitThread*. (Подробнее на эту тему см. раздел «Функция *ExitProcess*» в главе 4.)

В параметр *dwExitCode* Вы помещаете значение, которое система рассматривает как код завершения потока. Возвращаемого значения у этой функции нет, потому что после ее вызова поток перестает существовать.



ExitThread — это Windows-функция, которая уничтожает поток. Но никогда не вызывайте ее, если Вы пишете код на C/C++. Вместо нее Вы должны использовать функцию *_endthreadex* из библиотеки Visual C++. (Если Вы работаете с другим компилятором, он должен поддерживать свой эквивалент функции *ExitThread*.) Что именно делает *_endthreadex* и почему это так важно, я объясню потом.

Функция *TerminateThread*

Вызов этой функции также завершает поток:

```
BOOL TerminateThread(  
    HANDLE hThread,  
    DWORD dwExitCode);
```

В отличие от *ExitThread*, которая уничтожает только вызывающий поток, эта функция завершает поток, указанный в параметре *hThread*. В параметр *dwExitCode* Вы помещаете значение, которое система рассматривает как код завершения потока. После того как поток будет уничтожен, счетчик пользователей его объекта ядра «поток» уменьшится на 1.



TerminateThread — функция асинхронная, т. е. она сообщает системе, что Вы хотите завершить поток, но к тому времени, когда она вернет управление, поток может быть еще не уничтожен. Так что, если Вам нужно точно знать момент завершения потока, используйте *WaitForSingleObject* (см. главу 9) или аналогичную функцию, передав ей описатель этого потока.

Корректно написанное приложение не должно вызывать эту функцию, поскольку поток не получает никакого уведомления о завершении; из-за этого он не может выполнить должную очистку ресурсов.



Уничтожение потока при вызове *ExitThread* или возврате управления из функции потока приводит к разрушению его стека. Но если он завершен функцией *TerminateThread*, система не уничтожает стек, пока не завершится и процесс, которому принадлежал этот поток. Так сделано потому, что другие потоки могут использовать указатели, ссылающиеся на данные в стеке завершенного потока. Если бы они обратились к несуществующему стеку, произошло бы нарушение доступа.

Кроме того, при завершении потока система уведомляет об этом все DLL, подключенные к процессу — владельцу завершенного потока. Но при вызове *TerminateThread* такого не происходит, и процесс может быть завершен некорректно. (Подробнее на эту тему см. главу 20.)

Если завершается процесс

Функции *ExitProcess* и *TerminateProcess*, рассмотренные в главе 4, тоже завершают потоки. Единственное отличие в том, что они прекращают выполнение всех потоков, принадлежавших завершенному процессу. При этом гарантируется высвобождение любых выделенных процессу ресурсов, в том числе стеков потоков. Однако эти две функции уничтожают потоки принудительно — так, будто для каждого из них вызывается функция *TerminateThread*. А это означает, что очистка проводится некорректно: деструкторы C++-объектов не вызываются, данные на диск не сбрасываются и т. д.

Что происходит при завершении потока

А происходит вот что.

- Освобождаются все описатели User-объектов, принадлежавших потоку. В Windows большинство объектов принадлежит процессу, содержащему поток, из которого они были созданы. Сам поток владеет только двумя User-объектами: окнами и ловушками (hooks). Когда поток, создавший такие объекты, завершается, система уничтожает их автоматически. Прочие объекты разрушаются, только когда завершается владевший ими процесс.
- Код завершения потока меняется со `STILL_ACTIVE` на код, переданный в функцию *ExitThread* или *TerminateThread*.
- Объект ядра «поток» переводится в свободное состояние.
- Если данный поток является последним активным потоком в процессе, завершается и сам процесс.
- Счетчик пользователей объекта ядра «поток» уменьшается на 1.

При завершении потока сопоставленный с ним объект ядра «поток» не освобождается до тех пор, пока не будут закрыты все внешние ссылки на этот объект.

Когда поток завершился, толку от его описателя другим потокам в системе в общем немного. Единственное, что они могут сделать, — вызвать функцию *GetExitCodeThread*, проверить, завершен ли поток, идентифицируемый описателем *hThread*, и, если да, определить его код завершения.

```
BOOL GetExitCodeThread(
    HANDLE hThread,
    PDWORD pdwExitCode);
```

Код завершения возвращается в переменной типа DWORD, на которую указывает *pdwExitCode*. Если поток не завершен на момент вызова *GetExitCodeThread*, функция записывает в эту переменную идентификатор STILL_ACTIVE (0x103). При успешном вызове функция возвращает TRUE. К использованию описателя для определения факта завершения потока мы еще вернемся в главе 9.

Кое-что о внутреннем устройстве потока

Я уже объяснил Вам, как реализовать функцию потока и как заставить систему создать поток, который выполнит эту функцию. Теперь мы попробуем разобраться, как система справляется с данной задачей.

На рис. 6-1 показано, что именно должна сделать система, чтобы создать и инициализировать поток. Давайте приглядимся к этой схеме повнимательнее. Вызов *CreateThread* заставляет систему создать объект ядра «поток». При этом счетчику числа его пользователей присваивается начальное значение, равное 2. (Объект ядра «поток» уничтожается только после того, как прекращается выполнение потока и закрывается описатель, возвращенный функцией *CreateThread*.) Также инициализируются другие свойства этого объекта: счетчик числа простоев (suspension count) получает значение 1, а код завершения — значение STILL_ACTIVE (0x103). И, наконец, объект переводится в состояние «занято».

Создав объект ядра «поток», система выделяет стеку потока память из адресного пространства процесса и записывает в его самую верхнюю часть два значения. (Стеки потоков всегда строятся от старших адресов памяти к младшим.) Первое из них является значением параметра *pvParam*, переданного Вами функции *CreateThread*, а второе — это содержимое параметра *pfnStartAddr*, который Вы тоже передаете в *CreateThread*.

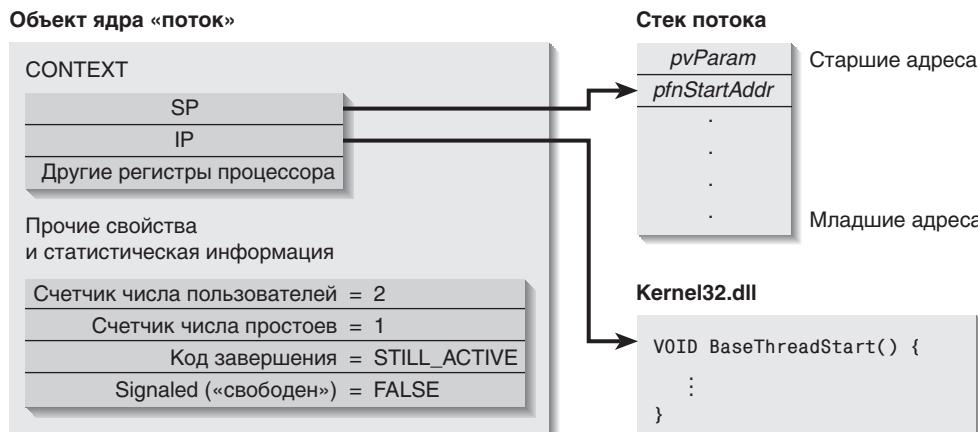


Рис. 6-1. Так создается и инициализируется поток

У каждого потока собственный набор регистров процессора, называемый **контекстом** потока. Контекст отражает состояние регистров процессора на момент последнего исполнения потока и записывается в структуру CONTEXT (она определена в заголовочном файле WinNT.h). Эта структура содержится в объекте ядра «поток».

Указатель команд (IP) и указатель стека (SP) — два самых важных регистра в контексте потока. Вспомните: потоки выполняются в контексте процесса. Соответственно эти регистры всегда указывают на адреса памяти в адресном пространстве процесса. Когда система инициализирует объект ядра «поток», указателю стека в структуре CONTEXT присваивается тот адрес, по которому в стек потока было записано значение *pfnStartAddr*, а указателю команд — адрес недокументированной (и неэкспортируемой) функции *BaseThreadStart*. Эта функция содержится в модуле Kernel32.dll, где, кстати, реализована и функция *CreateThread*.

Вот главное, что делает *BaseThreadStart*:

```
VOID BaseThreadStart(PTHREAD_START_ROUTINE pfnStartAddr, PVOID pvParam) {
    __try {
        ExitThread((pfnStartAddr)(pvParam));
    }
    __except(UnhandledExceptionFilter(GetExceptionInformation())) {
        ExitProcess(GetExceptionCode());
    }
    // ПРИМЕЧАНИЕ: мы никогда не попадем сюда
}
```

После инициализации потока система проверяет, был ли передан функции *CreateThread* флаг CREATE_SUSPENDED. Если нет, система обнуляет его счетчик числа простоев, и потоку может быть выделено процессорное время. Далее система загружает в регистры процессора значения, сохраненные в контексте потока. С этого момента поток может выполнять код и манипулировать данными в адресном пространстве своего процесса.

Поскольку указатель команд нового потока установлен на *BaseThreadStart*, именно с этой функции и начнется выполнение потока. Глядя на ее прототип, можно подумать, будто *BaseThreadStart* передаются два параметра, а значит, она вызывается из какой-то другой функции, но это не так. Новый поток просто начинает с нее свою работу. *BaseThreadStart* получает доступ к двум параметрам, которые появляются у нее потому, что операционная система записывает соответствующие значения в стек потока (а через него параметры как раз и передаются функциям). Правда, на некоторых аппаратных платформах параметры передаются не через стек, а с использованием определенных регистров процессора. Поэтому на таких аппаратных платформах система — прежде чем разрешить потоку выполнение функции *BaseThreadStart* — инициализирует нужные регистры процессора.

Когда новый поток выполняет *BaseThreadStart*, происходит следующее.

- Ваша функция потока включается во фрейм структурной обработки исключений (далее для краткости — SEH-фрейм), благодаря чему любое исключение, если оно происходит в момент выполнения Вашего потока, получает хоть какую-то обработку, предлагаемую системой по умолчанию. Подробнее о структурной обработке исключений см. главы 23, 24 и 25.
- Система обращается к Вашей функции потока, передавая ей параметр *pvParam*, который Вы ранее передали функции *CreateThread*.
- Когда Ваша функция потока возвращает управление, *BaseThreadStart* вызывает *ExitThread*, передавая ей значение, возвращенное Вашей функцией. Счетчик числа пользователей объекта ядра «поток» уменьшается на 1, и выполнение потока прекращается.

- Если Ваш поток вызывает необрабатываемое им исключение, его обрабатывает SEH-фрейм, построенный функцией *BaseThreadStart*. Обычно в результате этого появляется окно с каким-нибудь сообщением, и, когда пользователь закрывает его, *BaseThreadStart* вызывает *ExitProcess* и завершает весь процесс, а не только тот поток, в котором произошло исключение.

Обратите внимание, что из *BaseThreadStart* поток вызывает либо *ExitThread*, либо *ExitProcess*. А это означает, что поток никогда не выходит из данной функции; он всегда уничтожается внутри нее. Вот почему у *BaseThreadStart* нет возвращаемого значения — она просто ничего не возвращает.

Кстати, именно благодаря *BaseThreadStart* Ваша функция потока получает возможность вернуть управление по окончании своей работы. *BaseThreadStart*, вызывая функцию потока, заталкивает в стек свой адрес возврата и тем самым сообщает ей, куда надо вернуться. Но сама *BaseThreadStart* не возвращает управление. Иначе возникло бы нарушение доступа, так как в стеке потока нет ее адреса возврата.

При инициализации первичного потока его указатель команд устанавливается на другую недокументированную функцию — *BaseProcessStart*. Она почти идентична *BaseThreadStart* и выглядит примерно так:

```
VOID BaseProcessStart(PPROCESS_START_ROUTINE pfnStartAddr) {  
    __try {  
        ExitThread((pfnStartAddr)());  
    }  
    __except(UnhandledExceptionFilter(GetExceptionInformation())) {  
        ExitProcess(GetExceptionCode());  
    }  
    // ПРИМЕЧАНИЕ: мы никогда не попадем сюда  
}
```

Единственное различие между этими функциями в отсутствии ссылки на параметр *pvParam*. Функция *BaseProcessStart* обращается к стартовому коду библиотеки C/C++, который выполняет необходимую инициализацию, а затем вызывает Вашу входную функцию *main*, *wmain*, *WinMain* или *wWinMain*. Когда входная функция возвращает управление, стартовый код библиотеки C/C++ вызывает *ExitProcess*. Поэтому первичный поток приложения, написанного на C/C++, никогда не возвращается в *BaseProcessStart*.

Некоторые соображения по библиотеке C/C++

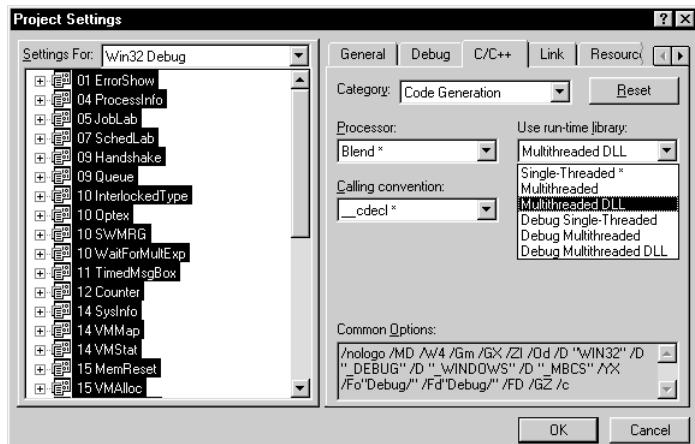
Microsoft поставляет с Visual C++ шесть библиотек C/C++. Их краткое описание представлено в следующей таблице.

Имя библиотеки	Описание
LibC.lib	Статически подключаемая библиотека для однопоточных приложений (используется по умолчанию при создании нового проекта)
LibCD.lib	Отладочная версия статически подключаемой библиотеки для однопоточных приложений
LibCMt.lib	Статически подключаемая библиотека для многопоточных приложений
LibCMtD.lib	Отладочная версия статически подключаемой библиотеки для многопоточных приложений
MSVCRT.lib	Библиотека импорта для динамического подключения рабочей версии MSVCRT.dll; поддерживает как одно-, так и многопоточные приложения

продолжение

Имя библиотеки	Описание
MSVCRTD.lib	Библиотека импорта для динамического подключения отладочной версии MSVCRTD.dll; поддерживает как одно-, так и многопоточные приложения

При реализации любого проекта нужно знать, с какой библиотекой его следует связать. Конкретную библиотеку можно выбрать в диалоговом окне Project Settings: на вкладке C/C++ в списке Category укажите Code Generation, а в списке Use Run-Time Library — одну из шести библиотек.



Наверное, Вам уже хочется спросить: «А зачем мне отдельные библиотеки для однопоточных и многопоточных программ?» Отвечаю. Стандартная библиотека C была разработана где-то в 1970 году — задолго до появления самого понятия многопоточности. Авторы этой библиотеки, само собой, не задумывались о проблемах, связанных с многопоточными приложениями.

Возьмем, к примеру, глобальную переменную *errno* из стандартной библиотеки C. Некоторые функции, если происходит какая-нибудь ошибка, записывают в эту переменную соответствующий код. Допустим, у Вас есть такой фрагмент кода:

```
BOOL fFailure = (system("NOTE PAD. EXE README. TXT") == -1);

if (fFailure) {
    switch (errno) {
        case E2BIG:           // список аргументов или размер окружения слишком велик
            break;
        case ENOENT:          // командный интерпретатор не найден
            break;
        case ENOEXEC:          // неверный формат командного интерпретатора
            break;
        case ENOMEM:          // недостаточно памяти для выполнения команды
            break;
    }
}
```

Теперь представим, что поток, выполняющий показанный выше код, прерван после вызова функции *system* и до оператора *if*. Допустим также, поток прерван для выпол-

нения другого потока (в том же процессе), который обращается к одной из функций библиотеки С, и та тоже заносит какое-то значение в глобальную переменную *errno*. Смотрите, что получается: когда процессор вернется к выполнению первого потока, в переменной *errno* окажется вовсе не то значение, которое было записано функцией *system*. Поэтому для решения этой проблемы нужно закрепить за каждым потоком свою переменную *errno*. Кроме того, понадобится какой-то механизм, который позволит каждому потоку ссыльаться на свою переменную *errno* и не трогать чужую.

Это лишь один пример того, что стандартная библиотека С/С++ не рассчитана на многопоточные приложения. Кроме *errno*, в ней есть еще целый ряд переменных и функций, с которыми возможны проблемы в многопоточной среде: *_doserrno*, *strtok*, *_wcstok*, *strerror*, *_strerror*, *tmpnam*, *tmpfile*, *asctime*, *_wasctime*, *gmtime*, *_ecvt*, *_fcvt* — список можно продолжить.

Чтобы многопоточные программы, использующие библиотеку С/С++, работали корректно, требуется создать специальную структуру данных и связать ее с каждым потоком, из которого вызываются библиотечные функции. Более того, они должны знать, что, когда Вы к ним обращаетесь, нужно просматривать этот блок данных в вызывающем потоке, чтобы не повредить данные в каком-нибудь другом потоке.

Так откуда же система знает, что при создании нового потока надо создать и этот блок данных? Ответ очень прост: не знает и знать не хочет. Вся ответственность — исключительно на Вас. Если Вы пользуетесь небезопасными в многопоточной среде функциями, то должны создавать потоки библиотечной функцией *_beginthreadex*, а не Windows-функцией *CreateThread*:

```
unsigned long _beginthreadex(
    void *security,
    unsigned stack_size,
    unsigned (*start_address)(void *),
    void *arglist,
    unsigned initflag,
    unsigned *thrdaddr);
```

У функции *_beginthreadex* тот же список параметров, что и у *CreateThread*, но их имена и типы несколько отличаются. (Группа, которая отвечает в Microsoft за разработку и поддержку библиотеки С/С++, считает, что библиотечные функции не должны зависеть от типов данных Windows.) Как и *CreateThread*, функция *_beginthreadex* возвращает описатель только что созданного потока. Поэтому, если Вы раньше пользовались функцией *CreateThread*, ее вызовы в исходном коде несложно заменить на вызовы *_beginthreadex*. Однако из-за некоторого расхождения в типах данных Вам придется позаботиться об их приведении к тем, которые нужны функции *_beginthreadex*, и тогда компилятор будет счастлив. Лично я создал небольшой макрос, *chBEGINTHREADEX*, который и делает всю эту работу в исходном коде.

```
typedef unsigned (_ _stdcall *PTHREAD_START) (void *);

#define chBEGINTHREADEX(psa, cbStack, pfnStartAddr, \
    pvParam, fdwCreate, pdwThreadID) \
        ((HANDLE) _beginthreadex( \
            (void *) (psa), \
            (unsigned) (cbStack), \
            (PTHREAD_START) (pfnStartAddr), \
            (void *) (pvParam), \
            (unsigned) (fdwCreate), \
            (unsigned *) (pdwThreadID)))
```

Заметьте, что функция `_beginthreadex` существует только в многопоточных версиях библиотеки C/C++. Связав проект с однопоточной библиотекой, Вы получите от компоновщика сообщение об ошибке «unresolved external symbol». Конечно, это сделано специально, потому что однопоточная библиотека не может корректно работать в многопоточном приложении. Также обратите внимание на то, что при создании нового проекта Visual Studio по умолчанию выбирает однопоточную библиотеку. Этот вариант не самый безопасный, и для многопоточных приложений Вы должны сами выбрать одну из многопоточных версий библиотеки C/C++.

Поскольку Microsoft поставляет исходный код библиотеки C/C++, несложно разобраться в том, что такого делает `_beginthreadex`, чего не делает `CreateThread`. На дистрибутивном компакт-диске Visual Studio ее исходный код содержится в файле Threadex.c. Чтобы не перепечатывать весь код, я решил дать Вам ее версию в псевдокоде, выделив самые интересные места.

```
unsigned long __cdecl _beginthreadex (
    void *psa,
    unsigned cbStack,
    unsigned (_stdcall * pfnStartAddr) (void *),
    void * pvParam,
    unsigned fdwCreate,
    unsigned *pdwThreadID) {

    _ptiddata ptd;           // указатель на блок данных потока
    unsigned long thdl;      // описатель потока

    // выделяется блок данных для нового потока
    if ((ptd = _calloc_crt(1, sizeof(struct tiddata))) == NULL)
        goto error_return;

    // инициализация блока данных
    initptd(ptd);

    // здесь запоминается нужная функция потока и параметр,
    // который мы хотим поместить в блок данных
    ptd->_initaddr = (void *) pfnStartAddr;
    ptd->_initarg = pvParam;

    // создание нового потока
    thdl = (unsigned long) CreateThread(psa, cbStack,
        _threadstartex, (PVOID) ptd, fdwCreate, pdwThreadID);
    if (thdl == NULL) {
        // создать поток не удалось; проводится очистка и сообщается об ошибке
        goto error_return;
    }

    // поток успешно создан; возвращается его описатель
    return(thdl);

error_return:
    // ошибка: не удалось создать блок данных или сам поток
    _free_crt(ptd);
    return((unsigned long)0L);
}
```

Несколько важных моментов, связанных с `_beginthreadex`.

- Каждый поток получает свой блок памяти `tidata`, выделяемый из кучи, которая принадлежит библиотеке C/C++. (Структура `tidata` определена в файле Mtdll.h. Она довольно любопытна, и я привел ее на рис. 6-2.)
- Адрес функции потока, переданный `_beginthreadex`, запоминается в блоке памяти `tidata`. Там же сохраняется и параметр, который должен быть передан этой функции.
- Функция `_beginthreadex` вызывает `CreateThread`, так как лишь с ее помощью операционная система может создать новый поток.
- При вызове `CreateThread` сообщается, что она должна начать выполнение нового потока с функции `_threadstartex`, а не с того адреса, на который указывает `pfnStartAddr`. Кроме того, функции потока передается не параметр `pvParam`, а адрес структуры `tidata`.
- Если все проходит успешно, `_beginthreadex`, как и `CreateThread`, возвращает описатель потока. В ином случае возвращается NULL.

```
struct _tidata {  
    unsigned long    _tid;           /* идентификатор потока */  
    unsigned long    _thandle;        /* описатель потока */  
    int              _terrno;         /* значение errno */  
    unsigned long    _tdoserrno;      /* значение _doserrno */  
    unsigned int     _fpds;           /* сегмент данных Floating Point */  
    unsigned long    _holdrand;       /* зародышевое значение для rand() */  
    char *           _token;          /* указатель (ptr) на метку strtok() */  
#ifdef _WIN32  
    wchar_t *        _wtoken;         /* ptr на метку wcstok() */  
#endif /* _WIN32 */  
    unsigned char *   _mtoken;         /* ptr на метку _mbstok() */  
  
    /* следующие указатели обрабатываются функцией malloc в период выполнения */  
    char *           _errmsg;         /* ptr на буфер strerror()/_strerror() */  
    char *           _namebuf0;        /* ptr на буфер tmpnam() */  
#ifdef _WIN32  
    wchar_t *        _wnamebuf0;       /* ptr на буфер _wttmpnam() */  
#endif /* _WIN32 */  
    char *           _namebuf1;        /* ptr на буфер tmpfile() */  
#ifdef _WIN32  
    wchar_t *        _wnamebuf1;       /* ptr на буфер _wttmpfile() */  
#endif /* _WIN32 */  
    char *           _asctimebuf;      /* ptr на буфер asctime() */  
#ifdef _WIN32  
    wchar_t *        _wasctimebuf;     /* ptr на буфер _wasctime() */  
#endif /* _WIN32 */  
    void *           _gmtimebuf;       /* ptr на структуру gmtime() */  
    char *           _cvtbuf;          /* ptr на буфер ecvt()/fcvt */  
  
    /* следующие поля используются кодом _beginthread */  
    void *           _initaddr;        /* начальный адрес пользовательского потока */  
    void *           _initarg;         /* начальный аргумент пользовательского потока */
```

Рис. 6-2. Локальная структура `tidata` потока, определенная в библиотеке C/C++

Рис. 6-2. продолжение

```

/* следующие три поля нужны для поддержки функции signal и обработки ошибок,
 * возникающих в период выполнения */
void *      _pxcptacttab;    /* ptr на таблицу "исключение-действие" */
void *      _tpxcptinfoptrs; /* ptr на указатели к информации об исключении */
int         _tfpecode;       /* код исключения для операций над числами
                                * с плавающей точкой */

/* следующее поле нужно подпрограммам NLG */
unsigned long _NLG_dwCode;

/*
 * данные для отдельного потока, используемые при обработке исключений в C++
 */
void *      _terminate;      /* подпрограмма terminate() */
void *      _unexpected;     /* подпрограмма unexpected() */
void *      _translator;     /* транслятор S.E. */
void *      _curexception;   /* текущее исключение */
void *      _curcontext;     /* контекст текущего исключения */
#if defined (_M_MRX000)
void *      _pFrameInfoChain;
void *      _pUnwindContext;
void *      _pExitContext;
int         _MipsPtdDelta;
int         _MipsPtdEpsilon;
#elif defined (_M_PPC)
void *      _pExitContext;
void *      _pUnwindContext;
void *      _pFrameInfoChain;
int         _FrameInfo[6];
#endif /* defined (_M_PPC) */
};

typedef struct _tiddata * _ptiddata;

```

Выяснив, как создается и инициализируется структура *tiddata* для нового потока, посмотрим, как она сопоставляется с этим потоком. Взгляните на исходный код функции *_threadstartex* (который тоже содержится в файле Threadex.c библиотеки C/C++). Вот моя версия этой функции в псевдокоде:

```

static unsigned long WINAPI threadstartex (void* ptd) {
    // Примечание: ptd - это адрес блока tiddata данного потока

    // блок tiddata сопоставляется с данным потоком
    TlsSetValue(_tlsindex, ptd);

    // идентификатор этого потока записывается в tiddata
    ((_ptiddata) ptd)->_tid = GetCurrentThreadId();

    // здесь инициализируется поддержка операций над числами с плавающей точкой
    // (код не показан)

```

см. след. стр.

```
// пользовательская функция потока включается в SEH-фрейм для обработки
// ошибок периода выполнения и поддержки signal
__try {
    // здесь вызывается функция потока, которой передается нужный параметр;
    // код завершения потока передается _endthreadex
    _endthreadex(
        ( (unsigned (WINAPI * )(void * ))((_ptidata)ptd)->_initaddr) )
        ( ((_ptidata)ptd)->_initarg ) );
}

__except(_XcptFilter(GetExceptionCode(), GetExceptionInformation())){
    // обработчик исключений из библиотеки С не даст нам попасть сюда
    _exit(GetExceptionCode());
}

// здесь мы тоже никогда не будем, так как в этой функции поток умирает
return(0L);
}
```

Несколько важных моментов, связанных со *_threadstartex*.

- Новый поток начинает выполнение с *BaseThreadStart* (в Kernel32.dll), а затем переходит в *_threadstartex*.
- В качестве единственного параметра функции *_threadstartex* передается адрес блока *tidata* нового потока.
- Windows-функция *TlsSetValue* сопоставляет с вызывающим потоком значение, которое называется локальной памятью потока (Thread Local Storage, TLS) (о ней я расскажу в главе 21), а *_threadstartex* сопоставляет блок *tidata* с новым потоком.
- Функция потока заключается в SEH-фрейм. Он предназначен для обработки ошибок периода выполнения (например, не перехваченных исключений C++), поддержки библиотечной функции *signal* и др. Этот момент, кстати, очень важен. Если бы Вы создали поток с помощью *CreateThread*, а потом вызвали библиотечную функцию *signal*, она работала бы некорректно.
- Далее вызывается функция потока, которой передается нужный параметр. Адрес этой функции и ее параметр были сохранены в блоке *tidata* функцией *_beginthreadex*.
- Значение, возвращаемое функцией потока, считается кодом завершения этого потока. Обратите внимание: *_threadstartex* не возвращается в *BaseThreadStart*. Иначе после уничтожения потока его блок *tidata* так и остался бы в памяти. А это привело бы к утечке памяти в Вашем приложении. Чтобы избежать этого, *_threadstartex* вызывает другую библиотечную функцию, *_endthreadex*, и передает ей код завершения.

Последняя функция, которую нам нужно рассмотреть, — это *_endthreadex* (ее исходный код тоже содержится в файле Threadex.c). Вот как она выглядит в моей версии (в псевдокоде):

```
void __cdecl _endthreadex (unsigned retcode) {
    _ptidata ptd;      // указатель на блок данных потока

    // здесь проводится очистка ресурсов, выделенных для поддержки операций
    // над числами с плавающей точкой (код не показан)
```

```

// определение адреса блока tiddata данного потока
ptd = _getptd();

// высвобождение блока tiddata
_freeptd(ptd);

// завершение потока
ExitThread(retcode);
}

```

Несколько важных моментов, связанных с *_endthreadex*.

- Библиотечная функция *_getptd* обращается к Windows-функции *TlsGetValue*, которая сообщает адрес блока памяти *tiddata* вызывающего потока.
- Этот блок освобождается, и вызовом *ExitThread* поток разрушается. При этом, конечно, передается корректный код завершения.

Где-то в начале главы я уже говорил, что прямого обращения к функции *ExitThread* следует избегать. Это правда, и я не отказываюсь от своих слов. Тогда же я сказал, что это приводит к уничтожению вызывающего потока и не позволяет ему вернуться из выполняемой в данный момент функции. А поскольку она не возвращает управление, любые созданные Вами C++-объекты не разрушаются. Так вот, теперь у Вас есть еще одна причина не вызывать *ExitThread*: она не дает освободить блок памяти *tiddata* потока, из-за чего в Вашем приложении может наблюдаться утечка памяти (до его завершения).

Разработчики Microsoft Visual C++, конечно, прекрасно понимают, что многие все равно будут пользоваться функцией *ExitThread*, поэтому они кое-что сделали, чтобы свести к минимуму вероятность утечки памяти. Если Вы действительно так хотите самостоятельно уничтожить свой поток, можете вызвать из него *_endthreadex* (вместо *ExitThread*) и тем самым освободить его блок *tiddata*. И все же я не рекомендую этого.

Сейчас Вы уже должны понимать, зачем библиотечным функциям нужен отдельный блок данных для каждого порожденного потока и каким образом после вызова *_beginthreadex* происходит создание и инициализация этого блока данных, а также его связывание с только что созданным потоком. Кроме того, Вы уже должны разбираться в том, как функция *_endthreadex* освобождает этот блок по завершении потока.

Как только блок данных инициализирован и сопоставлен с конкретным потоком, любая библиотечная функция, к которой обращается поток, может легко узнать адрес его блока и таким образом получить доступ к данным, принадлежащим этому потоку.

Ладно, с функциями все ясно, теперь попробуем проследить, что происходит с глобальными переменными вроде *errno*. В заголовочных файлах С эта переменная определена так:

```

#if defined(_MT) || defined(_DLL)
extern int * __cdecl _errno(void);
#define errno (*_errno())
#else /* !defined(_MT) && !defined(_DLL) */
extern int errno;
#endif /* _MT || _DLL */

```

Создавая многопоточное приложение, надо указывать в командной строке компилятора один из ключей: /MT (многопоточное приложение) или /MD (многопоточ-

ная DLL); тогда компилятор определит идентификатор `_MT`. После этого, ссылаясь на `errno`, Вы будете на самом деле вызывать внутреннюю функцию `_errno` из библиотеки C/C++. Она возвращает адрес элемента данных `errno` в блоке, сопоставленном с вызывающим потоком. Кстати, макрос `errno` составлен так, что позволяет получать содержимое памяти по этому адресу. А сделано это для того, чтобы можно было писать, например, такой код:

```
int *p = &errno;
if (*p == ENOMEM) {
    :
}
```

Если бы внутренняя функция `_errno` просто возвращала значение `errno`, этот код не удалось бы скомпилировать.

Многопоточная версия библиотеки C/C++, кроме того, «обертывает» некоторые функции синхронизирующими примитивами. Ведь если бы два потока одновременно вызывали функцию `malloc`, куча могла бы быть повреждена. Поэтому в многопоточной версии библиотеки потоки не могут одновременно выделять память из кучи. Второй поток она заставляет ждать до тех пор, пока первый не выйдет из функции `malloc`, и лишь тогда второй поток получает доступ к `malloc`. (Подробнее о синхронизации потоков мы поговорим в главах 8, 9 и 10.)

Конечно, все эти дополнительные операции не могли не отразиться на быстродействии многопоточной версии библиотеки. Поэтому Microsoft, кроме многопоточной, поставляет и однопоточную версию статически подключаемой библиотеки C/C++.

Динамически подключаемая версия библиотеки C/C++ вполне универсальна: ее могут использовать любые выполняемые приложения и DLL, которые обращаются к библиотечным функциям. По этой причине данная библиотека существует лишь в многопоточной версии. Поскольку она поставляется в виде DLL, ее код не нужно включать непосредственно в EXE- и DLL-модули, что существенно уменьшает их размер. Кроме того, если Microsoft исправляет какую-то ошибку в такой библиотеке, то и программы, построенные на ее основе, автоматически избавляются от этой ошибки.

Как Вы, наверное, и предполагали, стартовый код из библиотеки C/C++ создает и инициализирует блок данных для первичного потока приложения. Это позволяет без всяких опасений вызывать из первичного потока любые библиотечные функции. А когда первичный поток заканчивает выполнение своей входной функции, блок данных завершающего потока освобождается самой библиотекой. Более того, стартовый код делает все необходимое для структурной обработки исключений, благодаря чему из первичного потока можно спокойно обращаться и к библиотечной функции `signal`.

Ой, вместо `_beginthreadex` я по ошибке вызвал `CreateThread`

Вас, наверное, интересует, что случится, если создать поток не библиотечной функцией `_beginthreadex`, а Windows-функцией `CreateThread`. Когда этот поток вызовет какую-нибудь библиотечную функцию, которая манипулирует со структурой `tiddata`, произойдет следующее. (Большинство библиотечных функций реентерабельно и не требует этой структуры.) Сначала эта функция попытается выяснить адрес блока данных потока (вызовом `TlsGetValue`). Получив NULL вместо адреса `tiddata`, она узнает, что вызывающий поток не сопоставлен с таким блоком. Тогда библиотечная функция тут

же создаст и инициализирует блок *tiddata* для вызывающего потока. Далее этот блок будет сопоставлен с потоком (через *TlsSetValue*) и останется при нем до тех пор, пока выполнение потока не прекратится. С этого момента данная функция (как, впрочем, и любая другая из библиотеки C/C++) сможет пользоваться блоком *tiddata* потока.

Как это ни фантастично, но Ваш поток будет работать почти без глюков. Хотя некоторые проблемы все же появятся. Во-первых, если этот поток воспользуется библиотечной функцией *signal*, весь процесс завершится, так как SEH-фрейм не подготовлен. Во-вторых, если поток завершится, не вызвав *_endthreadex*, его блок данных не высвободится и произойдет утечка памяти. (Да и кто, интересно, вызовет *_endthreadex* из потока, созданного с помощью *CreateThread*?)



Если Вы связываете свой модуль с многопоточной DLL-версией библиотеки C/C++, то при завершении потока и высвобождении блока *tiddata* (если он был создан), библиотека получает уведомление DLL_THREAD_DETACH. Даже несмотря на то что это предотвращает утечку памяти, связанную с блоком *tiddata*, я настоятельно советую создавать потоки через *_beginthreadex*, а не с помощью *CreateThread*.

Библиотечные функции, которые лучше не вызывать

В библиотеке C/C++ содержится две функции:

```
unsigned long _beginthread(
    void (*__cdecl *start_address)(void *),
    unsigned stack_size,
    void *arglist);

и

void _endthread(void);
```

Первоначально они были созданы для того, чем теперь занимаются новые функции *_beginthreadex* и *_endthreadex*. Но, как видите, у *_beginthread* параметров меньше, и, следовательно, ее возможности ограничены в сравнении с полнофункциональной *_beginthreadex*. Например, работая с *_beginthread*, нельзя создать поток с атрибутами защиты, отличными от присваиваемых по умолчанию, нельзя создать поток и тут же его задержать — нельзя даже получить идентификатор потока. С функцией *_endthread* та же история: она не принимает никаких параметров, а это значит, что по окончании работы потока его код завершения всегда равен 0.

Однако с функцией *_endthread* дело обстоит куда хуже, чем кажется: перед вызовом *ExitThread* она обращается к *CloseHandle* и передает ей описатель нового потока. Чтобы разобраться, в чем тут проблема, взгляните на следующий код:

```
DWORD dwExitCode;
HANDLE hThread = _beginthread(...);
GetExitCodeThread(hThread, &dwExitCode);
CloseHandle(hThread);
```

Весьма вероятно, что созданный поток отработает и завершится еще до того, как первый поток успеет вызвать функцию *GetExitCodeThread*. Если так и случится, значение в *hThread* окажется недействительным, потому что *_endthread* уже закрыла описатель нового потока. И, естественно, вызов *CloseHandle* даст ошибку.

Новая функция `_endthreadex` не закрывает описатель потока, поэтому фрагмент кода, приведенный выше, будет нормально работать (если мы, конечно, заменим вызов `_beginthread` на вызов `_beginthreadex`). И в заключение, напомню еще раз: как только функция потока возвращает управление, `_beginthreadex` самостоятельно вызывает `_endthreadex`, а `_beginthread` обращается к `_endthread`.

Как узнать о себе

Потоки часто обращаются к Windows-функциям, которые меняют среду выполнения. Например, потоку может понадобиться изменить свой приоритет или приоритет процесса. (Приоритеты рассматриваются в главе 7.) И поскольку это не редкость, когда поток модифицирует среду (собственную или процесса), в Windows предусмотрены функции, позволяющие легко ссылаться на объекты ядра текущего процесса и потока:

```
HANDLE GetCurrentProcess();
HANDLE GetCurrentThread();
```

Обе эти функции возвращают псевдоописатель объекта ядра «процесс» или «поток». Они не создают новые описатели в таблице описателей, которая принадлежит вызывающему процессу, и не влияют на счетчики числа пользователей объектов ядра «процесс» и «поток». Поэтому, если вызвать `CloseHandle` и передать ей псевдоописатель, она проигнорирует вызов и просто вернет FALSE.

Псевдоописатели можно использовать при вызове функций, которым нужен описатель процесса. Так, поток может запросить все временные показатели своего процесса, вызвав `GetProcessTimes`:

```
FILETIME ftCreationTime, ftExitTime, ftKernelTime, ftUserTime;
GetProcessTimes(GetCurrentProcess(),
    &ftCreationTime, &ftExitTime, &ftKernelTime, &ftUserTime);
```

Аналогичным образом поток может выяснить собственные временные показатели, вызвав `GetThreadTimes`:

```
FILETIME ftCreationTime, ftExitTime, ftKernelTime, ftUserTime;
GetThreadTimes(GetCurrentThread(),
    &ftCreationTime, &ftExitTime, &ftKernelTime, &ftUserTime);
```

Некоторые Windows-функции позволяют указывать конкретный процесс или поток по его уникальному в рамках всей системы идентификатору. Вот функции, с помощью которых поток может выяснить такой идентификатор — собственный или своего процесса:

```
DWORD GetCurrentProcessId();
DWORD GetCurrentThreadId();
```

По сравнению с функциями, которые возвращают псевдоописатели, эти функции, как правило, не столь полезны, но когда-то и они могут пригодиться.

Преобразование псевдоописателя в настоящий описатель

Иногда бывает нужно выяснить настоящий, а не псевдоописатель потока. Под «настоящим» я подразумеваю описатель, который однозначно идентифицирует уникальный поток. Вдумайтесь в такой фрагмент кода:

```

DWORD WINAPI ParentThread(PVOID pvParam) {
    HANDLE hThreadParent = GetCurrentThread();
    CreateThread(NULL, 0, ChildThread, (PVOID) hThreadParent, 0, NULL);
    // далее следует какой-то код...
}

DWORD WINAPI ChildThread(PVOID pvParam) {
    HANDLE hThreadParent = (HANDLE) pvParam;
    FILETIME ftCreationTime, ftExitTime, ftKernelTime, ftUserTime;
    GetThreadTimes(hThreadParent,
        &ftCreationTime, &ftExitTime, &ftKernelTime, &ftUserTime);
    // далее следует какой-то код...
}

```

Вы заметили, что здесь не все ладно? Идея была в том, чтобы родительский поток передавал дочернему свой описатель. Но он передает псевдо-, а не настоящий описатель. Начиная выполнение, дочерний поток передает этот псевдоописатель функции *GetThreadTimes*, и она вследствие этого возвращает временные показатели своего — а вовсе не родительского! — потока. Происходит так потому, что псевдоописатель является описателем текущего потока, т. е. того, который вызывает эту функцию.

Чтобы исправить приведенный выше фрагмент кода, превратим псевдоописатель в настоящий через функцию *DuplicateHandle* (о ней я рассказывал в главе 3):

```

BOOL DuplicateHandle(
    HANDLE hSourceProcess,
    HANDLE hSource,
    HANDLE hTargetProcess,
    PHANDLE phTarget,
    DWORD fdwAccess,
    BOOL bInheritHandle,
    DWORD fdwOptions);

```

Обычно она используется для создания нового «процессо-зависимого» описателя из описателя объекта ядра, значение которого увязано с другим процессом. А мы воспользуемся *DuplicateHandle* не совсем по назначению и скорректируем с ее помощью наш фрагмент кода так:

```

DWORD WINAPI ParentThread(PVOID pvParam) {
    HANDLE hThreadParent;

    DuplicateHandle(
        GetCurrentProcess(),           // описатель процесса, к которому
                                       // относится псевдоописатель потока;
        GetCurrentThread(),           // псевдоописатель родительского потока;
        GetCurrentProcess(),           // описатель процесса, к которому
                                       // относится новый, настоящий
                                       // описатель потока;
        &hThreadParent,               // даст новый, настоящий описатель,
                                       // идентифицирующий родительский поток;
        0,                           // игнорируется из-за DUPLICATE_SAME_ACCESS;
        FALSE,                      // новый описатель потока ненаследуемый;
        DUPLICATE_SAME_ACCESS);     // новому описателю потока присваиваются
                                       // те же атрибуты защиты, что и псевдоописателю

```

см. след. стр.

```
CreateThread(NULL, 0, ChildThread, (PVOID) hThreadParent, 0, NULL);
// далее следует какой-то код...
}

DWORD WINAPI ChildThread(PVOID pvParam) {
    HANDLE hThreadParent = (HANDLE) pvParam;
    FILETIME ftCreationTime, ftExitTime, ftKernelTime, ftUserTime;
    GetThreadTimes(hThreadParent,
        &ftCreationTime, &ftExitTime, &ftKernelTime, &ftUserTime);
    CloseHandle(hThreadParent);
    // далее следует какой-то код...
}
```

Теперь родительский поток преобразует свой «двусмысленный» псевдоописатель в настоящий описатель, однозначно определяющий родительский поток, и передает его в *CreateThread*. Когда дочерний поток начинает выполнение, его параметр *pvParam* содержит настоящий описатель потока. В итоге вызов какой-либо функции с этим описателем влияет не на дочерний, а на родительский поток.

Поскольку *DuplicateHandle* увеличивает счетчик пользователей указанного объекта ядра, то, закончив работу с продублированным описателем объекта, очень важно не забыть уменьшить счетчик. Сразу после обращения к *GetThreadTimes* дочерний поток вызывает *CloseHandle*, уменьшая тем самым счетчик пользователей объекта «родительский поток» на 1. В этом фрагменте кода я исходил из того, что дочерний поток не вызывает других функций с передачей этого описателя. Если же ему надо вызвать какие-то функции с передачей описателя родительского потока, то, естественно, к *CloseHandle* следует обращаться только после того, как необходимость в этом описателе у дочернего потока отпадет.

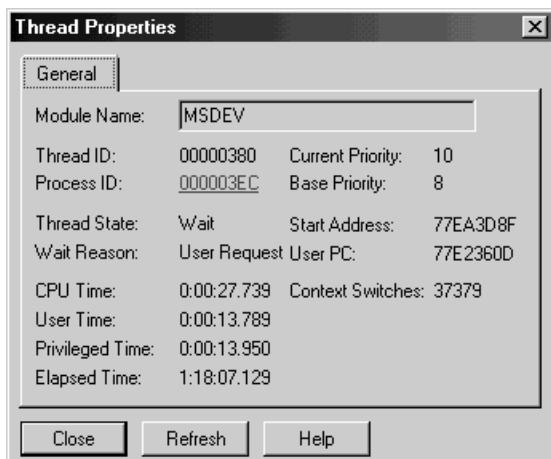
Надо заметить, что *DuplicateHandle* позволяет преобразовать и псевдоописатель процесса. Вот как это сделать:

```
HANDLE hProcess;
DuplicateHandle(
    GetCurrentProcess(),           // описатель процесса, к которому
                                    // относится псевдоописатель;
    GetCurrentProcess(),           // псевдоописатель процесса;
    GetCurrentProcess(),           // описатель процесса, к которому
                                    // относится новый, настоящий описатель;
    &hProcess,                   // даст новый, настоящий описатель,
                                // идентифицирующий процесс;
    0,                          // игнорируется из-за DUPLICATE_SAME_ACCESS;
    FALSE,                      // новый описатель процесса ненаследуемый;
    DUPLICATE_SAME_ACCESS);     // новому описателю процесса присваиваются
                                // те же атрибуты защиты, что и псевдоописателю
```

Планирование потоков, приоритет и привязка к процессорам

Операционная система с вытесняющей многозадачностью должна использовать тот или иной алгоритм, позволяющий ей распределять процессорное время между потоками. Здесь мы рассмотрим алгоритмы, применяемые в Windows 98 и Windows 2000.

В главе 6 мы уже обсудили структуру CONTEXT, поддерживаемую в объекте ядра «поток», и выяснили, что она отражает состояние регистров процессора на момент последнего выполнения потока процессором. Каждые 20 мс (или около того) Windows просматривает все существующие объекты ядра «поток» и отмечает те из них, которые могут получать процессорное время. Далее она выбирает один из таких объектов и загружает в регистры процессора значения из его контекста. Эта операция называется *переключением контекста* (context switching). По каждому потоку Windows ведет учет того, сколько раз он подключался к процессору. Этот показатель сообщают специальные утилиты вроде Microsoft Spy++. Например, на иллюстрации ниже показан список свойств одного из потоков. Обратите внимание, что этот поток подключался к процессору 37379 раз.



Поток выполняет код и манипулирует данными в адресном пространстве своего процесса. Примерно через 20 мс Windows сохранит значения регистров процессора в контексте потока и приостановит его выполнение. Далее система просмотрит остальные объекты ядра «поток», подлежащие выполнению, выберет один из них, загрузит его контекст в регистры процессора, и все повторится. Этот цикл операций — выбор потока, загрузка его контекста, выполнение и сохранение контекста — начинается с момента запуска системы и продолжается до ее выключения.

Таков вкратце механизм планирования работы множества потоков. Детали мы обсудим позже, но главное я уже показал. Все очень просто, да? Windows потому и называется системой с вытесняющей многозадачностью, что в любой момент может приостановить любой поток и вместо него запустить другой. Как Вы еще увидите, этим механизмом можно управлять, правда, крайне ограниченно. Всегда помните: Вы не в состоянии гарантировать, что Ваш поток будет выполняться непрерывно, что никакой другой поток не получит доступ к процессору и т. д.



Меня часто спрашивают, как сделать так, чтобы поток гарантированно запускался в течение определенного времени после какого-нибудь события — например, не позднее чем через миллисекунду после приема данных с последовательного порта? Ответ прост: никак. Такие требования можно предъявлять к операционным системам реального времени, но Windows к ним не относится. Лишь операционная система реального времени имеет полное представление о характеристиках аппаратных средств, на которых она работает (об интервалах запаздывания контроллеров жестких дисков, клавиатуры и т. д.). А создавая Windows, Microsoft ставила другую цель: обеспечить поддержку максимально широкого спектра оборудования — различных процессоров, дисковых устройств, сетей и др. Короче говоря, Windows не является операционной системой реального времени.

Хочу особо подчеркнуть, что система планирует выполнение только тех потоков, которые могут получать процессорное время, но большинство потоков в системе к таковым не относится. Так, у некоторых объектов-потоков значение счетчика простоеев (suspend count) больше 0, а значит, соответствующие потоки приостановлены и не получают процессорное время. Вы можете создать приостановленный поток вызовом *CreateProcess* или *CreateThread* с флагом *CREATE_SUSPENDED*. (В следующем разделе я расскажу и о таких функциях, как *SuspendThread* и *ResumeThread*.)

Кроме приостановленных, существуют и другие потоки, не участвующие в распределении процессорного времени, — они ожидают каких-либо событий. Например, если Вы запускаете Notepad и не работаете в нем с текстом, его поток бездействует, а система не выделяет процессорное время тем, кому нечего делать. Но стоит лишь сместить его окно, прокрутить в нем текст или что-то ввести, как система автоматически включит поток Notepad в число планируемых. Это вовсе не означает, что поток Notepad тут же начнет выполняться. Просто система учитывает его при планировании потоков и когда-нибудь выделит ему время — по возможности в ближайшем будущем.

Приостановка и возобновление потоков

В объекте ядра «поток» имеется переменная — счетчик числа простоев данного потока. При вызове *CreateProcess* или *CreateThread* он инициализируется значением, равным 1, которое запрещает системе выделять новому потоку процессорное время. Такая схема весьма разумна: сразу после создания поток не готов к выполнению, ему нужно время для инициализации.

После того как поток полностью инициализирован, *CreateProcess* или *CreateThread* проверяет, не передан ли ей флаг *CREATE_SUSPENDED*, и, если да, возвращает управление, оставив поток в приостановленном состоянии. В ином случае счетчик простоев обнуляется, и поток включается в число планируемых — если только он не ждет какого-то события (например, ввода с клавиатуры).

Создав поток в приостановленном состоянии, Вы можете настроить некоторые его свойства (например, приоритет, о котором мы поговорим позже). Закончив настройку, Вы должны разрешить выполнение потока. Для этого вызовите *ResumeThread* и передайте описатель потока, возвращенный функцией *CreateThread* (описатель можно взять и из структуры, на которую указывает параметр *ppiProcInfo*, передаваемый в *CreateProcess*).

```
DWORD ResumeThread(HANDLE hThread);
```

Если вызов *ResumeThread* прошел успешно, она возвращает предыдущее значение счетчика простоеев данного потока; в ином случае — 0xFFFFFFFF.

Выполнение отдельного потока можно приостанавливать несколько раз. Если поток приостановлен 3 раза, то и возобновлен он должен быть тоже 3 раза — лишь тогда система выделит ему процессорное время. Выполнение потока можно приостановить не только при его создании с флагом *CREATE_SUSPENDED*, но и вызовом *SuspendThread*:

```
DWORD SuspendThread(HANDLE hThread);
```

Любой поток может вызвать эту функцию и приостановить выполнение другого потока (конечно, если его описатель известен). Хоть об этом нигде и не говорится (но я все равно скажу!), приостановить свое выполнение поток способен сам, а возобновить себя без посторонней помощи — нет. Как и *ResumeThread*, функция *SuspendThread* возвращает предыдущее значение счетчика простоеев данного потока. Поток можно приостанавливать не более чем *MAXIMUM_SUSPEND_COUNT* раз (в файле WinNT.h это значение определено как 127). Обратите внимание, что *SuspendThread* в режиме ядра работает асинхронно, но в пользовательском режиме не выполняется, пока поток остается в приостановленном состоянии.

Создавая реальное приложение, будьте осторожны с вызовами *SuspendThread*, так как нельзя заранее сказать, чем будет заниматься его поток в момент приостановки. Например, он пытается выделить память из кучи и поэтому заблокировал к ней доступ. Тогда другим потокам, которым тоже нужна динамическая память, придется ждать его возобновления. *SuspendThread* безопасна только в том случае, когда Вы точно знаете, что делает (или может делать) поток, и предусматриваете все меры для исключения вероятных проблем и взаимной блокировки потоков. (О взаимной блокировке и других проблемах синхронизации потоков я расскажу в главах 8, 9 и 10.)

Приостановка и возобновление процессов

В Windows понятия «приостановка» и «возобновление» неприменимы к процессам, так как они не участвуют в распределении процессорного времени. Однако меня не раз спрашивали, как одним махом приостановить все потоки определенного процесса. Это можно сделать из другого процесса, причем он должен быть отладчиком и, в частности, вызывать функции вроде *WaitForDebugEvent* и *ContinueDebugEvent*.

Других способов приостановки всех потоков процесса в Windows нет: программа, выполняющая такую операцию, может «потерять» новые потоки. Система должна как-то приостанавливать в этот период не только все существующие, но и вновь создаваемые потоки. Microsoft предпочла встроить эту функциональность в системный механизм отладки.

Вам, конечно, не удастся написать идеальную функцию *SuspendProcess*, но вполне по силам добиться ее удовлетворительной работы во многих ситуациях. Вот мой вариант функции *SuspendProcess*.

```
VOID SuspendProcess(DWORD dwProcessID, BOOL fSuspend) {  
  
    // получаем список потоков в системе  
    HANDLE hSnapshot = CreateToolhelp32Snapshot(  
        TH32CS_SNAPTHREAD, dwProcessID);  
  
    if (hSnapshot != INVALID_HANDLE_VALUE) {  
  
        // просматриваем список потоков  
        THREADENTRY32 te = { sizeof(te) };  
        BOOL f0k = Thread32First(hSnapshot, &te);  
        for (; f0k; f0k = Thread32Next(hSnapshot, &te)) {  
  
            // относится ли данный поток к нужному процессу?  
            if (te.th32OwnerProcessID == dwProcessID) {  
  
                // пытаемся получить описатель потока по его идентификатору  
                HANDLE hThread = OpenThread(THREAD_SUSPEND_RESUME,  
                    FALSE, te.th32ThreadID);  
  
                if (hThread != NULL) {  
  
                    // приостанавливаем или возобновляем поток  
                    if (fSuspend)  
                        SuspendThread(hThread);  
                    else  
                        ResumeThread(hThread);  
                }  
                CloseHandle(hThread);  
            }  
        }  
        CloseHandle(hSnapshot);  
    }  
}
```

Для перечисления списка потоков я использую ToolHelp-функции (они рассматривались в главе 4). Определив потоки нужного процесса, я вызываю *OpenThread*:

```
HANDLE OpenThread(  
    DWORD dwDesiredAccess,  
    BOOL bInheritHandle,  
    DWORD dwThreadId);
```

Это новая функция, которая появилась в Windows 2000. Она находит объект ядра «поток» по идентификатору, указанному в *dwThreadId*, увеличивает его счетчик пользователей на 1 и возвращает описатель объекта. Получив описатель, я могу передать его в *SuspendThread* (или *ResumeThread*). *OpenThread* имеется только в Windows 2000, поэтому моя функция *SuspendProcess* не будет работать ни в Windows 95/98, ни в Windows NT 4.0.

Вероятно, Вы уже догадались, почему *SuspendProcess* будет срабатывать не во всех случаях: при перечислении могут создаваться новые и уничтожаться существующие потоки. После вызова *CreateToolhelp32Snapshot* в процессе может появиться новый поток, который моя функция уже не увидит, а значит, и не приостановит. Впоследствии, когда я попытаюсь возобновить потоки, вновь вызвав *SuspendProcess*, она во-

зобновит поток, который собственно и не приостанавливался. Но может быть еще хуже: при перечислении текущий поток уничтожается и создается новый с тем же идентификатором. Тогда моя функция приостановит неизвестно какой поток (и даже непонятно в каком процессе).

Конечно, все эти ситуации крайне маловероятны, и, если Вы точно представляете, что делает интересующий Вас процесс, никаких проблем не будет. В общем, используйте мою функцию на свой страх и риск.

Функция *Sleep*

Поток может сообщить системе не выделять ему процессорное время на определенный период, вызвав:

```
VOID Sleep(DWORD dwMilliseconds);
```

Эта функция приостанавливает поток на *dwMilliseconds* миллисекунд. Отметим несколько важных моментов, связанных с функцией *Sleep*.

- Вызывая *Sleep*, поток добровольно отказывается от остатка выделенного ему кванта времени.
- Система прекращает выделять потоку процессорное время на период, *примерно* равный заданному. Все верно: если Вы укажете остановить поток на 100 мс, приблизительно на столько он и «заснет», хотя не исключено, что его сон продлится на несколько секунд или даже минут больше. Вспомните, Windows не является системой реального времени. Ваш поток может возобновиться в данный момент, но это зависит от того, какая ситуация сложится в системе к тому времени.
- Вы можете вызвать *Sleep* и передать в *dwMilliseconds* значение *INFINITE*, вообще запретив планировать поток. Но это не очень практично — куда лучше корректно завершить поток, освободив его стек и объект ядра.
- Вы можете вызвать *Sleep* и передать в *dwMilliseconds* нулевое значение. Тогда Вы откажетесь от остатка своего кванта времени и заставите систему подключить к процессору другой поток. Однако система может снова запустить Ваш поток, если других планируемых потоков с тем же приоритетом нет.

Переключение потоков

Функция *SwitchToThread* позволяет подключить к процессору другой поток (если он есть):

```
BOOL SwitchToThread();
```

Когда Вы вызываете эту функцию, система проверяет, есть ли поток, которому не хватает процессорного времени. Если нет, *SwitchToThread* немедленно возвращает управление, а если да, планировщик отдает ему дополнительный квант времени (приоритет этого потока может быть ниже, чем у вызывающего). По истечении этого кванта планировщик возвращается в обычный режим работы.

SwitchToThread позволяет потоку, которому не хватает процессорного времени, отнять этот ресурс у потока с более низким приоритетом. Она возвращает FALSE, если на момент ее вызова в системе нет ни одного потока, готового к исполнению; в ином случае — ненулевое значение.

Вызов *SwitchToThread* аналогичен вызову *Sleep* с передачей в *dwMilliseconds* нулевого значения. Разница лишь в том, что *SwitchToThread* дает возможность выполнять потоки с более низким приоритетом, которым не хватает процессорного времени, а *Sleep* действует без оглядки на «голодающие» потоки.

WINDOWS 98 В Windows 98 функция *SwitchToThread* лишь определена, но не реализована.

Определение периодов выполнения потока

Иногда нужно знать, сколько времени затрачивает поток на выполнение той или иной операции. Многие в таких случаях пишут что-то вроде этого:

```
// получаем стартовое время  
DWORD dwStartTime = GetTickCount();  
  
// здесь выполняем какой-нибудь сложный алгоритм  
  
// вычитаем стартовое время из текущего  
DWORD dwElapsed = GetTickCount() - dwStartTime;
```

Этот код основан на простом допущении, что он не будет прерван. Но в операционной системе с вытесняющей многозадачностью никто не знает, когда поток получит процессорное время, и результат будет сильно искажен. Что нам здесь нужно, так это функция, которая сообщает время, затраченное процессором на обработку данного потока. К счастью, в Windows есть такая функция:

```
BOOL GetThreadTimes(  
    HANDLE hThread,  
    PFILETIME pftCreationTime,  
    PFILETIME pftExitTime,  
    PFILETIME pftKernelTime,  
    PFILETIME pftUserTime);
```

GetThreadTimes возвращает четыре временных параметра:

Показатель времени	Описание
Время создания (creation time)	Абсолютная величина, выраженная в интервалах по 100 нс. Отсчитывается с полуночи 1 января 1601 года по Гринвичу до момента создания потока.
Время завершения (exit time)	Абсолютная величина, выраженная в интервалах по 100 нс. Отсчитывается с полуночи 1 января 1601 года по Гринвичу до момента завершения потока. Если поток все еще выполняется, этот показатель имеет неопределенное значение.
Время выполнения ядра (kernel time)	Относительная величина, выраженная в интервалах по 100 нс. Сообщает время, затраченное этим потоком на выполнение кода операционной системы.
Время выполнения User (User time)	Относительная величина, выраженная в интервалах по 100 нс. Сообщает время, затраченное потоком на выполнение кода приложения.

С помощью этой функции можно определить время, необходимое для выполнения сложного алгоритма:

```
_int64 FileTimeToQuadWord(PFILETIME pft) {
    return(Int64ShlMod32(pft->dwHighDateTime, 32) | pft->dwLowDateTime);
}

void PerformLongOperation () {

    FILETIME ftKernelTimeStart, ftKernelTimeEnd;
    FILETIME ftUserTimeStart, ftUserTimeEnd;
    FILETIME ftDummy;
    _int64 qwKernelTimeElapsed, qwUserTimeElapsed, qwTotalTimeElapsed;

    // получаем начальные показатели времени
    GetThreadTimes(GetCurrentThread(), &ftDummy, &ftDummy,
                   &ftKernelTimeStart, &ftUserTimeStart);

    // здесь выполняем сложный алгоритм

    // получаем конечные показатели времени
    GetThreadTimes(GetCurrentThread(), &ftDummy, &ftDummy,
                   &ftKernelTimeEnd, &ftUserTimeEnd);

    // получаем значения времени, затраченного на выполнение ядра и User,
    // преобразуя начальные и конечные показатели времени из FILETIME
    // в четверенные слова, а затем вычитая начальные показатели из конечных
    qwKernelTimeElapsed = FileTimeToQuadWord(&ftKernelTimeEnd) -
                           FileTimeToQuadWord(&ftKernelTimeStart);

    qwUserTimeElapsed = FileTimeToQuadWord(&ftUserTimeEnd) -
                        FileTimeToQuadWord(&ftUserTimeStart);

    // получаем общее время, складывая время выполнения ядра и User
    qwTotalTimeElapsed = qwKernelTimeElapsed + qwUserTimeElapsed;

    // общее время хранится в qwTotalTimeElapsed
}
```

Заметим, что существует еще одна функция, аналогичная *GetThreadTimes* и применимая ко всем потокам в процессе:

```
BOOL GetProcessTimes(
    HANDLE hProcess,
    PFILETIME pftCreationTime,
    PFILETIME pftExitTime,
    PFILETIME pftKernelTime,
    PFILETIME pftUserTime);
```

GetProcessTimes возвращает временные параметры, суммированные по всем потокам (даже уже завершенным) в указанном процессе. Так, время выполнения ядра будет суммой периодов времени, затраченного всеми потоками процесса на выполнение кода операционной системы.

WINDOWS 98 К сожалению, в Windows 98 функции *GetThreadTimes* и *GetProcessTimes* определены, но не реализованы. Так что в Windows 98 нет надежного механизма, с помощью которого можно было бы определить, сколько процессорного времени выделяется потоку или процессу.

GetThreadTimes не годится для высокоточного измерения временных интервалов — для этого в Windows предусмотрено две специальные функции:

```
BOOL QueryPerformanceFrequency(LARGE_INTEGER* pliFrequency);  
BOOL QueryPerformanceCounter(LARGE_INTEGER* pliCount);
```

Они построены на том допущении, что поток не вытесняется, поскольку высокоточные измерения проводятся, как правило, в очень быстро выполняемых блоках кода. Чтобы слегка упростить работу с этими функциями, я создал следующий C++-класс:

```
class CStopwatch {  
public:  
    CStopwatch() { QueryPerformanceFrequency(&m_liPerfFreq); Start(); }  
  
    void Start() { QueryPerformanceCounter(&m_liPerfStart); }  
  
    __int64 Now() const { // возвращает число миллисекунд после вызова Start  
        LARGE_INTEGER liPerfNow;  
        QueryPerformanceCounter(&liPerfNow);  
        return(((liPerfNow.QuadPart - m_liPerfStart.QuadPart) * 1000)  
               / m_liPerfFreq.QuadPart);  
    }  
  
private:  
    LARGE_INTEGER m_liPerfFreq;      // количество отсчетов в секунду  
    LARGE_INTEGER m_liPerfStart;     // начальный отсчет  
};
```

Я применяю этот класс так:

```
// создаю секундомер (начинающий отсчет с текущего момента времени)  
CStopwatch stopwatch;  
  
// здесь я помещаю код, время выполнения которого нужно измерить  
  
// определяю, сколько времени прошло  
__int64 qwElapsedTime = stopwatch.Now();  
  
// qwElapsedTime сообщает длительность выполнения в миллисекундах
```

Структура CONTEXT

К этому моменту Вы должны понимать, какую важную роль играет структура CONTEXT в планировании потоков. Система сохраняет в ней состояние потока перед самым отключением его от процессора, благодаря чему его выполнение возобновляется с того места, где было прервано.

Вы, наверное, удивитесь, но в документации Platform SDK структуре CONTEXT отведен буквально один абзац:

«В структуре CONTEXT хранятся данные о состоянии регистров с учетом специфики конкретного процессора. Она используется системой для выполнения различных внутренних операций. В настоящее время такие структуры определены для процессоров Intel, MIPS, Alpha и PowerPC. Соответствующие определения см. в заголовочном файле WinNT.h.»

В документации нет ни слова об элементах этой структуры, набор которых зависит от типа процессора. Фактически CONTEXT — единственная из всех структур Windows, специфичная для конкретного процессора.

Так из чего же состоит структура CONTEXT? Давайте посмотрим. Ее элементы четко соответствуют регистрам процессора. Например, для процессоров *x86* в число элементов входят *Eax*, *Ebx*, *Ecx*, *Edx* и т. д., а для процессоров Alpha — *IntVO*, *IntT0*, *IntT1*, *IntSO*, *IntRa*, *IntZero* и др. Структура CONTEXT для процессоров *x86* выглядит так:

```
typedef struct _CONTEXT {
    // ...
    // Флаги, управляющие содержимым записи CONTEXT.
    //
    // Если запись контекста используется как входной параметр, тогда раздел,
    // управляемый флагом (когда он установлен), считается содержащим
    // действительные значения. Если запись контекста используется для
    // модификации контекста потока, то изменяются только те разделы, для
    // которых флаг установлен.
    //
    // Если запись контекста используется как входной и выходной параметр
    // для захвата контекста потока, возвращаются только те разделы контекста,
    // для которых установлены соответствующие флаги. Запись контекста никогда
    // не используется только как выходной параметр.
    //
    DWORD ContextFlags;

    //
    // Этот раздел определяется/возвращается, когда в ContextFlags установлен
    // флаг CONTEXT_DEBUG_REGISTERS. Заметьте, что CONTEXT_DEBUG_REGISTERS
    // не включаются в CONTEXT_FULL.
    //

    DWORD Dr0;
    DWORD Dr1;
    DWORD Dr2;
    DWORD Dr3;
    DWORD Dr6;
    DWORD Dr7;

    //
    // Этот раздел определяется/возвращается, когда в ContextFlags
    // установлен флаг CONTEXT_FLOATING_POINT.
    //

    FLOATING_SAVE_AREA FloatSave;
```

см. след. стр.

```
//  
// Этот раздел определяется/возвращается, когда в ContextFlags  
// установлен флаг CONTEXT_SEGMENTS.  
//  
DWORD SegGs;  
DWORD SegFs;  
DWORD SegEs;  
DWORD SegDs;  
  
//  
// Этот раздел определяется/возвращается, когда в ContextFlags  
// установлен флаг CONTEXT_INTEGER.  
//  
DWORD Edi;  
DWORD Esi;  
DWORD Ebx;  
DWORD Edx;  
DWORD ECX;  
DWORD Eax;  
  
//  
// Этот раздел определяется/возвращается, когда в ContextFlags  
// установлен флаг CONTEXT_CONTROL.  
//  
DWORD Ebp;  
DWORD Eip;  
DWORD SegCs;      // следует очистить  
DWORD EFlags;     // следует очистить  
DWORD Esp;  
DWORD SegSs;  
  
//  
// Этот раздел определяется/возвращается, когда в ContextFlags  
// установлен флаг CONTEXT_EXTENDED_REGISTERS.  
// Формат и смысл значений зависят от типа процессора.  
//  
BYTE ExtendedRegisters[MAXIMUM_SUPPORTED_EXTENSION];  
} CONTEXT;
```

Эта структура разбита на несколько разделов. Раздел CONTEXT_CONTROL содержит управляющие регистры процессора: указатель команд, указатель стека, флаги и адрес возврата функции. (В отличие от x86, который при вызове функции помещает адрес возврата в стек, процессор Alpha сохраняет адрес возврата в одном из регистров.) Раздел CONTEXT_INTEGER соответствует целочисленным регистрам процессора, CONTEXT_FLOATING_POINT — регистрам с плавающей точкой, CONTEXT_SEGMENTS — сегментным регистрам (только для x86), CONTEXT_DEBUG_REGISTERS — регистрам, предназначенным для отладки (только для x86), а CONTEXT_EXTENDED_REGISTERS — дополнительным регистрам (только для x86).

Windows фактически позволяет заглянуть внутрь объекта ядра «поток» и получить сведения о текущем состоянии регистров процессора. Для этого предназначена функция:

```
BOOL GetThreadContext(
    HANDLE hThread,
    PCONTEXT pContext);
```

Создайте экземпляр структуры CONTEXT, инициализируйте нужные флаги (в элементе *ContextFlags*) и передайте функции *GetThreadContext* адрес этой структуры. Функция поместит значения в элементы, сведения о которых Вы запросили.

Прежде чем обращаться к *GetThreadContext*, приостановите поток вызовом *SuspendThread*, иначе поток может быть подключен к процессору, и значения регистров существенно изменятся. На самом деле у потока есть два контекста: пользовательского режима и режима ядра. *GetThreadContext* возвращает лишь первый из них. Если Вы вызываете *SuspendThread*, когда поток выполняет код операционной системы, пользовательский контекст можно считать достоверным, даже несмотря на то что поток еще не остановлен (он все равно не выполнит ни одной команды пользовательского кода до последующего возобновления).

Единственный элемент структуры CONTEXT, которому не соответствует какой-либо регистр процессора, — *ContextFlags*. Присутствуя во всех вариантах этой структуры независимо от типа процессора, он подсказывает функции *GetThreadContext*, значения каких регистров Вы хотите узнать. Например, чтобы получить значения управляющих регистров для потока, напишите что-то вроде:

```
// создаем экземпляр структуры CONTEXT
CONTEXT Context;

// сообщаем системе, что нас интересуют сведения
// только об управляющих регистрах
Context.ContextFlags = CONTEXT_CONTROL;

// требуем от системы информацию о состоянии
// регистров процессора для данного потока
GetThreadContext(hThread, &Context);

// действительные значения содержат элементы структуры CONTEXT,
// соответствующие управляющим регистрам, остальные значения
// не определены
```

Перед вызовом *GetThreadContext* надо инициализировать элемент *ContextFlags*. Чтобы получить значения как управляющих, так и целочисленных регистров, инициализируйте его так:

```
// сообщаем системе, что нас интересуют
// управляющие и целочисленные регистры
Context.ContextFlags = CONTEXT_CONTROL | CONTEXT_INTEGER;
```

Есть еще один идентификатор, позволяющий узнать значения важнейших регистров (т. е. используемых, по мнению Microsoft, чаще всего):

```
// сообщаем системе, что нас интересуют
// все значимые регистры
Context.ContextFlags = CONTEXT_FULL;
```

CONTEXT_FULL определен в файле WinNT.h, как показано в таблице.

Тип процессора	Определение CONTEXT_FULL
x86	CONTEXT_CONTROL CONTEXT_INTEGER CONTEXT_SEGMENTS
Alpha	CONTEXT_CONTROL CONTEXT_FLOATING_POINT CONTEXT_INTEGER

После возврата из *GetThreadContext* Вы легко проверите значения любых регистров для потока, но помните, что такой код зависит от типа процессора. В следующей таблице перечислены элементы структуры CONTEXT, соответствующие указателям команд и стека для разных типов процессоров.

Тип процессора	Указатель команд	Указатель стека
x86	CONTEXT.Eip	CONTEXT.Esp
Alpha	CONTEXT.Fir	CONTEXT.IntSp

Даже удивительно, какой мощный инструмент дает Windows в руки разработчику! Но есть вещь, от которой Вы придете в полный восторг: значения элементов CONTEXT можно изменять и передавать объекту ядра «поток» с помощью функции *SetThreadContext*.

```
BOOL SetThreadContext(  
    HANDLE hThread,  
    CONST CONTEXT *pContext);
```

Перед этой операцией поток тоже нужно приостановить, иначе результаты могут быть непредсказуемыми.

Прежде чем обращаться к *SetThreadContext*, инициализируйте элемент *ContextFlags*, как показано ниже.

```
CONTEXT Context;
```

```
// приостанавливаем поток  
SuspendThread(hThread);  
  
// получаем регистры для контекста потока  
Context.ContextFlags = CONTEXT_CONTROL;  
GetThreadContext(hThread, &Context);  
  
// устанавливаем указатель команд по своему выбору;  
// в нашем примере присваиваем значение 0x00010000  
#if defined(_ALPHA_)  
Context.Fir = 0x00010000;  
#elif defined(_X86_)  
Context.Eip = 0x00010000;  
#else  
#error Module contains CPU-specific code; modify and recompile.  
#endif  
  
// вносим изменения в регистры потока; ContextFlags  
// можно и не инициализировать, так как это уже сделано  
Context.ControlFlags = CONTEXT_CONTROL;  
SetThreadContext(hThread, &Context);  
  
// возобновляем выполнение потока; оно начнется с адреса 0x00010000  
ResumeThread(hThread);
```

Этот код, вероятно, приведет к ошибке защиты (нарушению доступа) в удаленном потоке; система сообщит о необработанном исключении, и удаленный процесс будет закрыт. Все верно — не Ваш, а удаленный. Вы благополучно обрушили другой процесс, оставив свой в целости и сохранности!

Функции *GetThreadContext* и *SetThreadContext* наделяют Вас огромной властью над потоками, но пользоваться ею нужно с осторожностью. Вызывают их лишь считанные приложения. Эти функции предназначены для отладчиков и других инструментальных средств, хотя обращаться к ним можно из любых программ.

Подробнее о структуре CONTEXT мы поговорим в главе 24.

Приоритеты потоков

В начале главы я сказал, что поток получает доступ к процессору на 20 мс, после чего планировщик переключает процессор на выполнение другого потока. Так происходит, только если у всех потоков один приоритет, но на самом деле в системе существуют потоки с разными приоритетами, а это меняет порядок распределения процессорного времени.

Каждому потоку присваивается уровень приоритета — от 0 (самый низкий) до 31 (самый высокий). Решая, какому потоку выделить процессорное время, система сначала рассматривает только потоки с приоритетом 31 и подключает их к процессору по принципу карусели. Если поток с приоритетом 31 не исключен из планирования, он немедленно получает квант времени, по истечении которого система проверяет, есть ли еще один такой поток. Если да, он тоже получает свой квант процессорного времени.

Пока в системе имеются планируемые потоки с приоритетом 31, ни один поток с более низким приоритетом процессорного времени не получает. Такая ситуация называется «голоданием» (starvation). Она наблюдается, когда потоки с более высоким приоритетом так интенсивно используют процессорное время, что остальные практически не работают. Вероятность этой ситуации намного ниже в многопроцессорных системах, где потоки с приоритетами 31 и 30 могут выполняться одновременно. Система всегда старается, чтобы процессоры были загружены работой, и они пропстаивают только в отсутствие планируемых потоков.

На первый взгляд, в системе, организованной таким образом, у потоков с низким приоритетом нет ни единого шанса на исполнение. Но, как я уже говорил, зачастую потоки как раз и не нужно выполнять. Например, если первичный поток Вашего процесса вызывает *GetMessage*, а система видит, что никаких сообщений пока нет, она приостанавливает его выполнение, отнимает остаток неиспользованного времени и тут же подключает к процессору другой ожидающий поток. И пока в системе не появятся сообщения для потока Вашего процесса, он будет пропстаивать — система не станет тратить на него процессорное время. Но вот в очереди этого потока появляется сообщение, и система сразу же подключает его к процессору (если только в этот момент не выполняется поток с более высоким приоритетом).

А теперь обратите внимание на еще один момент. Потоки с более высоким приоритетом всегда вытесняют потоки с более низким приоритетом независимо от того, исполняются последние или нет. Допустим, процессор исполняет поток с приоритетом 5, и тут система обнаруживает, что поток с более высоким приоритетом готов к выполнению. Что будет? Система остановит поток с более низким приоритетом — даже если не истек отведенный ему квант процессорного времени — и подключит к процессору поток с более высоким приоритетом (и, между прочим, выдаст ему полный квант времени).

Кстати, при загрузке системы создается особый поток — *поток обнуления страниц* (zero page thread), которому присваивается нулевой уровень приоритета. Ни один поток, кроме этого, не может иметь нулевой уровень приоритета. Он обнуляет свободные страницы в оперативной памяти при отсутствии других потоков, требующих внимания со стороны системы.

Абстрагирование приоритетов

Создавая планировщик потоков, разработчики из Microsoft прекрасно понимали, что он не подойдет на все случаи жизни. Они также осознавали, что со временем «назначение» компьютера может измениться. Например, в момент выпуска Windows NT создание приложений с поддержкой OLE еще только начиналось. Теперь такие приложения — обычное дело. Кроме того, значительно расширилось применение игрового программного обеспечения, ну и, конечно же, Интернета.

Алгоритм планирования потоков существенно влияет на выполнение приложений. С самого начала разработчики Microsoft понимали, что его придется изменять по мере того, как будут расширяться сферы применения компьютеров. Microsoft гарантирует, что наши программы будут работать и в следующих версиях Windows. Как же ей удается изменять внутреннее устройство системы, не нарушая работоспособность наших программ? Ответ в том, что:

- планировщик документируется не полностью;
- Microsoft не разрешает в полной мере использовать все особенности планировщика;
- Microsoft предупреждает, что алгоритм работы планировщика постоянно меняется, и не рекомендует писать программы в расчете на текущий алгоритм.

Windows API предоставляет слой абстрагирования от конкретного алгоритма работы планировщика, запрещая прямое обращение к планировщику. Вместо этого Вы вызываете функции Windows, которые «интерпретируют» Ваши параметры в зависимости от версии системы. Я буду рассказывать именно об этом слое абстрагирования.

Проектируя свое приложение, Вы должны учитывать возможность параллельного выполнения других программ. Следовательно, Вы обязаны выбирать класс приоритета, исходя из того, насколько «отзывчивой» должна быть Ваша программа. Согласен, такая формулировка довольно туманна, но так и задумано: Microsoft не желает обеспечить ничего такого, что могло бы нарушить работу Вашего кода в будущем.

Windows поддерживает шесть классов приоритета: idle (простаивающий), below normal (ниже обычного), normal (обычный), above normal (выше обычного), high (высокий) и realtime (реального времени). Самый распространенный класс приоритета, естественно, — normal; его использует 99% приложений. Классы приоритета показаны в следующей таблице.

Класс приоритета	Описание
Real-time	Потоки в этом процессе обязаны немедленно реагировать на события, обеспечивая выполнение критических по времени задач. Такие потоки вытесняют даже компоненты операционной системы. Будьте крайне осторожны с этим классом.
High	Потоки в этом процессе тоже должны немедленно реагировать на события, обеспечивая выполнение критических по времени задач. Этот класс присвоен, например, Task Manager, что дает возможность пользователю закрывать больше неконтролируемые процессы.

продолжение

Класс приоритета	Описание
Above normal	Класс приоритета, промежуточный между normal и high. Это новый класс, введенный в Windows 2000.
Normal	Потоки в этом процессе не предъявляют особых требований к выделению им процессорного времени.
Below normal	Класс приоритета, промежуточный между normal и idle. Это новый класс, введенный в Windows 2000.
Idle	Потоки в этом процессе выполняются, когда система не занята другой работой. Этот класс приоритета обычно используется для утилит, работающих в фоновом режиме, экранных заставок и приложений, собирающих статистическую информацию.

Приоритет idle идеален для программ, выполняемых, только когда системе больше нечего делать. Примеры таких программ — экранные заставки и средства мониторинга. Компьютер, не используемый в интерактивном режиме, может быть занят другими задачами (действуя, скажем, в качестве файлового сервера), и их потокам незачем конкурировать с экранной заставкой за доступ к процессору. Средства мониторинга, собирающие статистическую информацию о системе, тоже не должны мешать выполнению более важных задач.

Класс приоритета high следует использовать лишь при крайней необходимости. Может, Вы этого и не знаете, но Explorer выполняется с высоким приоритетом. Большинство часть времени его потоки простоявают, готовые пробудиться, как только пользователь нажмет какую-нибудь клавишу или щелкнет кнопку мыши. Пока потоки Explorer простоявают, система не выделяет им процессорное время, что позволяет выполнять потоки с более низким приоритетом. Но вот пользователь нажал, скажем, Ctrl+Esc, и система пробуждает поток Explorer. (Комбинация клавиш Ctrl+Esc попутно открывает меню Start.) Если в данный момент исполняются потоки с более низким приоритетом, они немедленно вытесняются, и начинает работать поток Explorer. Microsoft разработала Explorer именно так потому, что любой пользователь — независимо от текущей ситуации в системе — ожидает мгновенной реакции оболочки на свои команды. В сущности, окна Explorer можно открывать, даже когда все потоки с более низким приоритетом зависают в бесконечных циклах. Обладая более высоким приоритетом, потоки Explorer вытесняют поток, исполняющий бесконечный цикл, и дают возможность закрыть зависший процесс.

Надо отметить высокую степень продуманности Explorer. Основную часть времени он просто «спит», не требуя процессорного времени. Будь это не так, вся система работала бы гораздо медленнее, а многие приложения просто не отзывались бы на действия пользователя.

Классом приоритета real-time почти никогда не стоит пользоваться. На самом деле в ранних бета-версиях Windows NT 3.1 присвоение этого класса приоритета приложениям даже не предусматривалось, хотя операционная система поддерживала эту возможность. Real-time — чрезвычайно высокий приоритет, и, поскольку большинство потоков в системе (включая управляющие самой системой) имеет более низкий приоритет, процесс с таким классом окажет на них сильное влияние. Так, потоки реального времени могут заблокировать необходимые операции дискового и сетевого ввода-вывода и привести к несвоевременной обработке ввода от мыши и клавиатуры — пользователь может подумать, что система зависла. У Вас должна быть очень веская причина для применения класса real-time — например, программе требуется

реагировать на события в аппаратных средствах с минимальной задержкой или выполнять быстротечную операцию, которую нельзя прерывать ни при каких обстоятельствах.



Процесс с классом приоритета real-time нельзя запустить, если пользователь не имеет привилегии Increase Scheduling Priority. По умолчанию такой привилегией обладает администратор и пользователь с расширенными полномочиями.

Конечно, большинство процессов имеет обычный класс приоритета. В Windows 2000 появилось два новых промежуточных класса — below normal и above normal. Microsoft добавила их, поскольку некоторые компании жаловались, что существующий набор классов приоритетов не дает нужной гибкости.

Выбрав класс приоритета, забудьте о том, как Ваша программа будет выполняться совместно с другими приложениями, и сосредоточьтесь на ее потоках. Windows поддерживает семь относительных приоритетов потоков: idle (простаивающий), lowest (нижний), below normal (ниже обычного), normal (обычный), above normal (выше обычного), highest (высший) и time-critical (критичный по времени). Эти приоритеты относительны классу приоритета процесса. Как обычно, большинство потоков использует обычный приоритет. Относительные приоритеты потоков описаны в следующей таблице.

Относительный приоритет потока	Описание
Time-critical	Поток выполняется с приоритетом 31 в классе real-time и с приоритетом 15 в других классах
Highest	Поток выполняется с приоритетом на два уровня выше обычного для данного класса
Above normal	Поток выполняется с приоритетом на один уровень выше обычного для данного класса
Normal	Поток выполняется с обычным приоритетом процесса для данного класса
Below normal	Поток выполняется с приоритетом на один уровень ниже обычного для данного класса
Lowest	Поток выполняется с приоритетом на два уровня ниже обычного для данного класса
Idle	Поток выполняется с приоритетом 16 в классе real-time и с приоритетом 1 в других классах

Итак, Вы присваиваете процессу некий класс приоритета и можете изменять относительные приоритеты потоков в пределах процесса. Заметьте, что я не сказал ни слова об уровнях приоритетов 0–31. Разработчики приложений не имеют с ними дела. Уровень приоритета формируется самой системой, исходя из класса приоритета процесса и относительного приоритета потока. А механизм его формирования — как раз то, чем Microsoft не хочет себя ограничивать. И действительно, этот механизм меняется практически в каждой версии системы.

В следующей таблице показано, как формируется уровень приоритета в Windows 2000, но не забывайте, что в Windows NT и тем более в Windows 95/98 этот механизм действует несколько иначе. Учтите также, что в будущих версиях Windows он вновь изменится.

Например, обычный поток в обычном процессе получает уровень приоритета 8. Поскольку большинство процессов имеет класс normal, а большинство потоков —

относительный приоритет *normal*, у основной части потоков в системе уровень приоритета равен 8.

Обычный поток в процессе с классом приоритета *high* получает уровень приоритета 13. Изменив класс приоритета процесса на *idle*, Вы снизите уровень приоритета того же потока до 4. Вспомните, что приоритет потока всегда относителен классу приоритета его процесса. Изменение класса приоритета процесса не влияет на относительные приоритеты его потоков, но сказывается на уровне их приоритета.

Относительный приоритет потока	Idle	Класс приоритета процесса				
		Below normal	Normal	Above normal	High	Real-time
Time-critical (критичный по времени)	15	15	15	15	15	31
Highest (высший)	6	8	10	12	15	26
Above normal (выше обычного)	5	7	9	11	14	25
Normal (обычный)	4	6	8	10	13	24
Below normal (ниже обычного)	3	5	7	9	12	23
Lowest (низший)	2	4	6	8	11	22
Idle (простаивающий)	1	1	1	1	1	16

Обратите внимание, что в таблице не показано, как задать уровень приоритета 0. Это связано с тем, что нулевой приоритет зарезервирован для потока обнуления страниц, и никакой другой поток не может иметь такой приоритет. Кроме того, уровни 17–21 и 27–30 в обычном приложении тоже недоступны. Вы можете пользоваться ими, только если пишете драйвер устройства, работающий в режиме ядра. И еще одно: уровень приоритета потока в процессе с классом *real-time* не может опускаться ниже 16, а потока в процессе с любым другим классом — подниматься выше 15.



Концепция класса приоритета вводит некоторых в заблуждение. Они делают отсюда вывод, будто процессы участвуют в распределении процессорного времени. Так вот, процессы никогда не получают процессорное время — оно выделяется лишь потокам. Класс приоритета процесса — сугубо абстрактная концепция, введенная Microsoft с единственной целью: скрыть от разработчика внутреннее устройство планировщика.



В общем случае поток с высоким уровнем приоритета должен быть активен как можно меньше времени. При появлении у него какой-либо работы он тут же получает процессорное время. Выполнив минимальное количество команд, он должен снова вернуться в ждущий режим. С другой стороны, поток с низким уровнем приоритета может оставаться активным и занимать процессор довольно долго. Следуя этим правилам, Вы сохранитеющую отзывчивость операционной системы на действия пользователя.

Программирование приоритетов

Так как же процесс получает класс приоритета? Очень просто. Вызывая *CreateProcess*, Вы можете указать в ее параметре *fdwCreate* нужный класс приоритета. Идентификаторы этих классов приведены в следующей таблице.

Класс приоритета	Идентификатор
Real-time	REALTIME_PRIORITY_CLASS
High	HIGH_PRIORITY_CLASS
Above normal	ABOVE_NORMAL_PRIORITY_CLASS
Normal	NORMAL_PRIORITY_CLASS
Below normal	BELOW_NORMAL_PRIORITY_CLASS
Idle	IDLE_PRIORITY_CLASS

Вам может показаться странным, что, создавая дочерний процесс, родительский сам устанавливает ему класс приоритета. За примером далеко ходить не надо — возьмем все тот же Explorer. При запуске из него какого-нибудь приложения новый процесс создается с обычным приоритетом. Но Explorer ведь не знает, что делает этот процесс и как часто его потокам надо выделять процессорное время. Поэтому в системе предусмотрена возможность изменения класса приоритета самим выполняемым процессом — вызовом функции *SetPriorityClass*:

```
BOOL SetPriorityClass(  
    HANDLE hProcess,  
    DWORD fdwPriority);
```

Эта функция меняет класс приоритета процесса, определяемого описателем *hProcess*, в соответствии со значением параметра *fdwPriority*. Последний должен содержать одно из значений, указанных в таблице выше. Поскольку *SetPriorityClass* принимает описатель процесса, Вы можете изменить приоритет любого процесса, выполняемого в системе, — если его описатель известен и у Вас есть соответствующие права доступа.

Обычно процесс пытается изменить свой класс приоритета. Вот как процесс может сам себе установить класс приоритета idle:

```
BOOL SetPriorityClass(GetCurrentProcess(), IDLE_PRIORITY_CLASS);
```

Парная ей функция *GetPriorityClass* позволяет узнать класс приоритета любого процесса:

```
DWORD GetPriorityClass(HANDLE hProcess);
```

Она возвращает, как Вы догадываетесь, один из ранее перечисленных флагов.

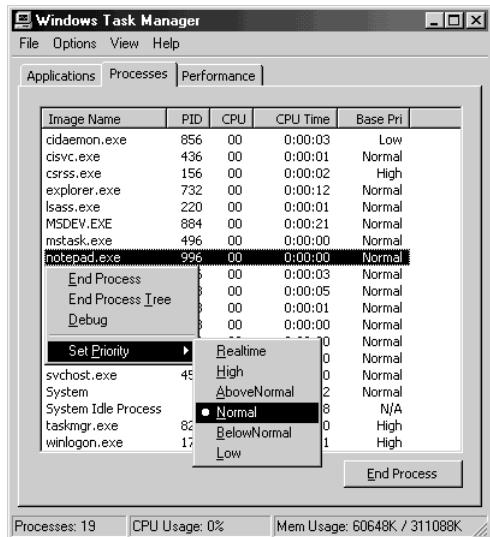
При запуске из оболочки командного процессора начальный приоритет программы тоже обычный. Однако, запуская ее командой Start, можно указать ключ, определяющий начальный приоритет. Так, следующая команда, введенная в оболочке командного процессора, заставит систему запустить приложение Calculator и присвоить ему приоритет idle:

```
C:\>START /LOW CALC.EXE
```

Команда Start допускает также ключи /BELOWNORMAL, /NORMAL, /ABOVENORMAL, /HIGH и /REALTIME, позволяющие начать выполнение программы с соответствующим классом приоритета. Разумеется, после запуска программа может вызвать *SetPriorityClass* и установить себе другой класс приоритета.

WINDOWS 98 В Windows 98 команда Start не поддерживает ни один из этих ключей. Из оболочки командного процессора Windows 98 процессы всегда запускаются с классом приоритета normal.

Task Manager в Windows 2000 дает возможность изменять класс приоритета процесса. На рисунке ниже показана вкладка Processes в окне Task Manager со списком выполняемых на данный момент процессов. В колонке Base Pri сообщается класс приоритета каждого процесса. Вы можете изменить его, выбрав процесс и указав другой класс в подменю Set Priority контекстного меню.



Только что созданный поток получает относительный приоритет `normal`. Почему `CreateThread` не позволяет задать относительный приоритет — для меня так и остается загадкой. Такая операция осуществляется вызовом функции:

```
BOOL SetThreadPriority(  
    HANDLE hThread,  
    int nPriority);
```

Разумеется, параметр *bThread* указывает на поток, чей приоритет Вы хотите изменить, а через *nPriority* передается один из идентификаторов (см. таблицу ниже).

Относительный приоритет потока	Идентификатор
Time-critical	THREAD_PRIORITY_TIME_CRITICAL
Highest	THREAD_PRIORITY_HIGHEST
Above normal	THREAD_PRIORITY_ABOVE_NORMAL
Normal	THREAD_PRIORITY_NORMAL
Below normal	THREAD_PRIORITY_BELOW_NORMAL
Lowest	THREAD_PRIORITY_LOWEST
Idle	THREAD_PRIORITY_IDLE

Функция *GetThreadPriority*, парная *SetThreadPriority*, позволяет узнать относительный приоритет потока:

```
int GetThreadPriority(HANDLE hThread);
```

Она возвращает один из идентификаторов, показанных в таблице выше.

Чтобы создать поток с относительным приоритетом idle, сделайте, например, так:

```
DWORD dwThreadID;
HANDLE hThread = CreateThread(NULL, 0, ThreadFunc, NULL,
    CREATE_SUSPENDED, &dwThreadID);
SetThreadPriority(hThread, THREAD_PRIORITY_IDLE);
ResumeThread(hThread);
CloseHandle(hThread);
```

Заметьте, что *CreateThread* всегда создает поток с относительным приоритетом *normal*. Чтобы присвоить потоку относительный приоритет *idle*, создайте приостановленный поток, передав в *CreateThread* флаг *CREATE_SUSPENDED*, а потом вызовите *SetThreadPriority* и установите нужный приоритет. Далее можно вызвать *ResumeThread*, и поток будет включен в число планируемых. Сказать заранее, когда поток получит процессорное время, нельзя, но планировщик уже учитывает его новый приоритет. Выполнив эти операции, Вы можете закрыть описатель потока, чтобы соответствующий объект ядра был уничтожен по завершении данного потока.



Ни одна Windows-функция не возвращает уровень приоритета потока. Такая ситуация создана преднамеренно. Вспомните, что Microsoft может в любой момент изменить алгоритм распределения процессорного времени. Поэтому при разработке приложений не стоит опираться на какие-то нюансы этого алгоритма. Используйте классы приоритетов процессов и относительные приоритеты потоков, и Ваши приложения будут нормально работать как в нынешних, так и в следующих версиях Windows.

Динамическое изменение уровня приоритета потока

Уровень приоритета, получаемый комбинацией относительного приоритета потока и класса приоритета процесса, которому принадлежит данный поток, называют *базовым уровнем приоритета потока*. Иногда система изменяет уровень приоритета потока. Обычно это происходит в ответ на некоторые события, связанные с вводом-выводом (например, на появление оконных сообщений или чтение с диска).

Так, поток с относительным приоритетом *normal*, выполняемый в процессе с классом приоритета *high*, имеет базовый приоритет 13. Если пользователь нажимает какую-нибудь клавишу, система помещает в очередь потока сообщение *WM_KEYDOWN*. А поскольку в очереди потока появилось сообщение, поток становится планируемым. При этом драйвер клавиатуры может заставить систему временно поднять уровень приоритета потока с 13 до 15 (действительное значение может отличаться в ту или другую сторону).

Процессор исполняет поток в течение отведенного отрезка времени, а по его истечении система снижает приоритет потока на 1, до уровня 14. Далее потоку вновь выделяется квант процессорного времени, по окончании которого система опять снижает уровень приоритета потока на 1. И теперь приоритет потока снова соответствует его базовому уровню.

Текущий уровень приоритета не может быть ниже базового. Кроме того, драйвер устройства, «разбудивший» поток, сам устанавливает величину повышения приоритета. И опять же Microsoft не документирует, насколько повышаются эти значения конкретными драйверами. Таким образом, она получает возможность тонко настраивать динамическое изменение приоритетов потоков в операционной системе, чтобы та максимально быстро реагировала на действия пользователя.

Система повышает приоритет только тех потоков, базовый уровень которых находится в пределах 1–15. Именно поэтому данный диапазон называется «областью динамического приоритета» (dynamic priority range). Система не допускает динамического повышения приоритета потока до уровней реального времени (более 15). Поскольку потоки с такими уровнями обслуживают системные функции, это ограничение не дает приложению нарушить работу операционной системы. И, кстати, система никогда не меняет приоритет потоков с уровнями реального времени (от 16 до 31).

Некоторые разработчики жаловались, что динамическое изменение приоритета системой отрицательно сказывается на производительности их приложений, и поэтому Microsoft добавила две функции, позволяющие отключать этот механизм:

```
BOOL SetProcessPriorityBoost(  
    HANDLE hProcess,  
    BOOL DisablePriorityBoost);  
  
BOOL SetThreadPriorityBoost(  
    HANDLE hThread,  
    BOOL DisablePriorityBoost);
```

SetProcessPriorityBoost заставляет систему включить или отключить изменение приоритетов всех потоков в указанном процессе, а *SetThreadPriorityBoost* действует применительно к отдельным потокам. Эти функции имеют свои аналоги, позволяющие определять, разрешено или запрещено изменение приоритетов:

```
BOOL GetProcessPriorityBoost(  
    HANDLE hProcess,  
    PBOOL pDisablePriorityBoost);  
  
BOOL GetThreadPriorityBoost(  
    HANDLE hThread,  
    PBOOL pDisablePriorityBoost);
```

Каждой из этих двух функций Вы передаете описатель нужного процесса или потока и адрес переменной типа *BOOL*, в которой и возвращается результат.

WINDOWS 98 В Windows 98 эти четыре функции определены, но не реализованы, и при вызове любой из них возвращается FALSE. Последующий вызов *GetLastError* дает *ERROR_CALL_NOT_IMPLEMENTED*.

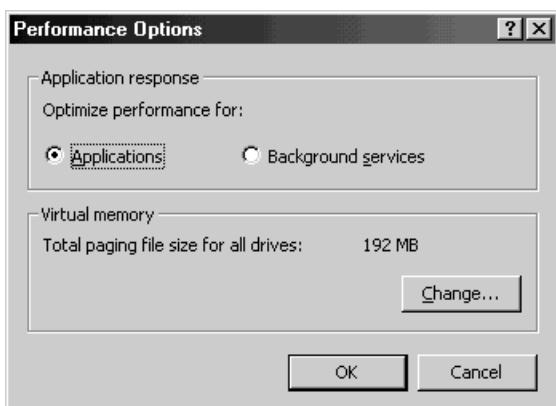
Есть еще одна ситуация, в которой система динамически повышает приоритет потока. Представьте, что поток с приоритетом 4 готов к выполнению, но не может получить доступ к процессору из-за того, что его постоянно занимают потоки с приоритетом 8. Это типичный случай «голодания» потока с более низким приоритетом. Обнаружив такой поток, не выполняемый на протяжении уже трех или четырех секунд, система поднимает его приоритет до 15 и выделяет ему двойную порцию времени. По его истечении потоку немедленно возвращается его базовый приоритет.

Подстройка планировщика для активного процесса

Когда пользователь работает с окнами какого-то процесса, последний считается *активным* (foreground process), а остальные процессы — *фоновыми* (background processes). Естественно, пользователь заинтересован в повышенной отзывчивости активного процесса по сравнению с фоновыми. Для этого Windows подстраивает алгоритм планирования потоков активного процесса. В Windows 2000, когда процесс становит-

ся активным, система выделяет его потокам более длительные кванты времени. Такая регулировка применяется только к процессам с классом приоритета normal.

Windows 2000 позволяет модифицировать работу этого механизма подстройки. Щелкнув кнопку Performance Options на вкладке Advanced диалогового окна System Properties, Вы открываете следующее окно.



Переключатель Applications включает подстройку планировщика для активного процесса, а переключатель Background Services — выключает (в этом случае оптимизируется выполнение фоновых сервисов). В Windows 2000 Professional по умолчанию выбирается переключатель Applications, а в остальных версиях Windows 2000 — переключатель Background Services, так как серверы редко используются в интерактивном режиме.

Windows 98 тоже позволяет подстраивать распределение процессорного времени для потоков активного процесса с классом приоритета normal. Когда процесс этого класса становится активным, система повышает на 1 приоритет его потоков, если их исходные приоритеты были lowest, below normal, normal, above normal или highest; приоритет потоков idle или time-critical не меняется. Поэтому поток с относительным приоритетом normal в активном процессе с классом приоритета normal имеет уровень приоритета 9, а не 8. Когда процесс вновь становится фоновым, приоритеты его потоков автоматически возвращаются к исходным уровням.

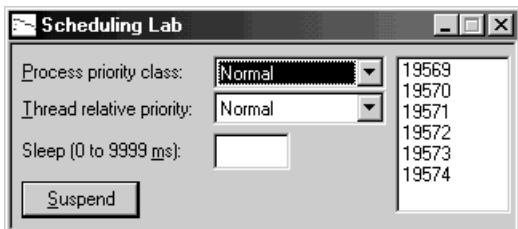
WINDOWS 98 Windows 98 не предусматривает возможности настройки этого механизма, так как не рассчитана на работу в качестве выделенного сервера.

Причина для таких изменений активных процессов очень проста: система дает им возможность быстрее реагировать на пользовательский ввод. Если бы приоритеты их потоков не менялись, то и обычный процесс фоновой печати, и обычный, но активный процесс, принимающий пользовательский ввод, — оба одинаково конкурировали бы за процессорное время. И тогда пользователь, набирая текст в активном приложении, заметил бы, что текст появляется на экране какими-то рывками. Но благодаря тому, что система повышает уровни приоритета потоков активного процесса, они получают преимущество над потоками обычных фоновых процессов.

Программа-пример Scheduling Lab

Эта программа, «07 SchedLab.exe» (см. листинг на рис. 7-1), позволяет экспериментировать с классами приоритетов процессов и относительными приоритетами потоков

и исследовать их влияние на общую производительность системы. Файлы исходного кода и ресурсов этой программы находятся в каталоге 07-SchedLab на компакт-диске, прилагаемом к книге. После запуска SchedLab открывается окно, показанное ниже.



Изначально первичный поток работает очень активно, и степень использования процессора подскакивает до 100%. Все, чем он занимается, — постоянно увеличивает исходное значение на 1 и выводит текущее значение в крайнее справа окно списка. Все эти числа не несут никакой смысловой информации; их появление просто демонстрирует, что поток чем-то занят. Чтобы прочувствовать, как повлияет на него изменение приоритета, запустите по крайней мере два экземпляра программы. Можете также открыть Task Manager и понаблюдать за нагрузкой на процессор, создаваемой каждым экземпляром.

В начале теста процессор будет загружен на 100%, и Вы увидите, что все экземпляры SchedLab получают примерно равные кванты процессорного времени. (Task Manager должен показать практически одинаковые процентные доли для всех ее экземпляров.) Как только Вы поднимете класс приоритета одного из экземпляров до above normal или high, львиную долю процессорного времени начнет получать именно этот экземпляр, а аналогичные показатели для других экземпляров резко упадут. Однако они никогда не опустятся до нуля — это действует механизм динамического повышения приоритета «голодающих» процессов. Теперь Вы можете самостоятельно поиграть с изменением классов приоритетов процессов и относительных приоритетов потоков. Возможность установки класса приоритета real-time я исключил намеренно, чтобы не нарушить работу операционной системы. Если Вы все же хотите поэкспериментировать с этим приоритетом, Вам придется модифицировать исходный текст моей программы.

Используя поле Sleep, можно приостановить первичный поток на заданное число миллисекунд в диапазоне от 0 до 9999. Попробуйте приостанавливать его хотя бы на 1 мс и посмотрите, сколько процессорного времени это позволит сэкономить. На своем ноутбуке с процессором Pentium II 300 МГц, я выиграл аж 99% — впечатляет!

Кнопка Suspend заставляет первичный поток создать дочерний поток, который приостанавливает родительский и выводит следующее окно.



Пока это окно открыто, первичный поток полностью отключается от процессора, а дочерний тоже не требует процессорного времени, так как ждет от пользователя дальнейших действий. Вы можете свободно перемещать это окно в пределах экрана или убрать его в сторону от основного окна программы. Поскольку первичный поток остановлен, основное окно не принимает оконных сообщений (в том числе

WM_PAINT). Это еще раз доказывает, что поток задержан. Закрыв окно с сообщением, Вы возобновите первичный поток, и нагрузка на процессор снова возрастет до 100%.

А теперь проведите еще один эксперимент. Откройте диалоговое окно Performance Options (я говорил о нем в предыдущем разделе) и выберите переключатель Background Services (или, наоборот, Application). Потом запустите несколько экземпляров моей программы с классом приоритета normal и выберите один из них, сделав его активным процессом. Вы сможете наглядно убедиться, как эти переключатели влияют на активные и фоновые процессы.



SchedLab.cpp

```

/*
Модуль: SchedLab.cpp
Автор: Copyright (c) 2000, Джейфри Рихтер (Jeffrey Richter)
*/

#include "..\CmnHdr.h"           /* см. приложение A */
#include <windowsx.h>
#include <tchar.h>
#include <process.h>             // для доступа к _beginthreadex
#include "Resource.h"

//////////



DWORD WINAPI ThreadFunc(PVOID pvParam) {
    HANDLE hThreadPrimary = (HANDLE) pvParam;
    SuspendThread(hThreadPrimary);
    chMB(
        "The Primary thread is suspended.\n"
        "It no longer responds to input and produces no output.\n"
        "Press OK to resume the primary thread & exit this secondary thread.\n");
    ResumeThread(hThreadPrimary);
    CloseHandle(hThreadPrimary);

    // во избежание взаимной блокировки после ResumeThread вызываем EnableWindow
    EnableWindow(
        GetDlgItem(FindWindow(NULL, TEXT("Scheduling Lab")), IDC_SUSPEND), TRUE);
    return(0);
}

//////////


BOOL Dlg_OnInitDialog (HWND hwnd, HWND hwndFocus, LPARAM lParam) {
    chSETDLGICONS(hwnd, IDI_SCHEDLAB);

    // инициализируем классы приоритетов процесса
    HWND hwndCtl = GetDlgItem(hwnd, IDC_PROCESSPRIORITYCLASS);
}

```

Рис. 7-1. Программа-пример *SchedLab*

Рис. 7-1. продолжение

```

int n = ComboBox_AddString(hwndCtl, TEXT("High"));
ComboBox_SetItemData(hwndCtl, n, HIGH_PRIORITY_CLASS);

// запоминаем текущий класс приоритета своего процесса
DWORD dwpc = GetPriorityClass(GetCurrentProcess());

if (SetPriorityClass(GetCurrentProcess(), BELOW_NORMAL_PRIORITY_CLASS)) {

    // эта система поддерживает класс приоритета below normal

    // восстанавливаем исходный класс приоритета
    SetPriorityClass(GetCurrentProcess(), dwpc);

    // добавляем класс above normal
    n = ComboBox_AddString(hwndCtl, TEXT("Above normal"));
    ComboBox_SetItemData(hwndCtl, n, ABOVE_NORMAL_PRIORITY_CLASS);

    dwpc = 0; // данная система поддерживает класс below normal
}

int nNormal = n = ComboBox_AddString(hwndCtl, TEXT("Normal"));
ComboBox_SetItemData(hwndCtl, n, NORMAL_PRIORITY_CLASS);

if (dwpc == 0) {

    // эта система поддерживает класс приоритета below normal

    // добавляем класс below normal
    n = ComboBox_AddString(hwndCtl, TEXT("Below normal"));
    ComboBox_SetItemData(hwndCtl, n, BELOW_NORMAL_PRIORITY_CLASS);
}

n = ComboBox_AddString(hwndCtl, TEXT("Idle"));
ComboBox_SetItemData(hwndCtl, n, IDLE_PRIORITY_CLASS);

ComboBox_SetCurSel(hwndCtl, nNormal);

// инициализируем относительные приоритеты потоков
hwndCtl = GetDlgItem(hwnd, IDC_THREADRELATIVEPRIORITY);

n = ComboBox_AddString(hwndCtl, TEXT("Time critical"));
ComboBox_SetItemData(hwndCtl, n, THREAD_PRIORITY_TIME_CRITICAL);

n = ComboBox_AddString(hwndCtl, TEXT("Highest"));
ComboBox_SetItemData(hwndCtl, n, THREAD_PRIORITY_HIGHEST);

n = ComboBox_AddString(hwndCtl, TEXT("Above normal"));
ComboBox_SetItemData(hwndCtl, n, THREAD_PRIORITY_ABOVE_NORMAL);

```

см. след. стр.

Рис. 7-1. продолжение

```
nNormal = n = ComboBox_AddString(hwndCtl, TEXT("Normal"));
ComboBox_SetItemData(hwndCtl, n, THREAD_PRIORITY_NORMAL);

n = ComboBox_AddString(hwndCtl, TEXT("Below normal"));
ComboBox_SetItemData(hwndCtl, n, THREAD_PRIORITY_BELOW_NORMAL);

n = ComboBox_AddString(hwndCtl, TEXT("Lowest"));
ComboBox_SetItemData(hwndCtl, n, THREAD_PRIORITY_LOWEST);

n = ComboBox_AddString(hwndCtl, TEXT("Idle"));
ComboBox_SetItemData(hwndCtl, n, THREAD_PRIORITY_IDLE);

ComboBox_SetCurSel(hwndCtl, nNormal);

Edit_LimitText(GetDlgItem(hwnd, IDC_SLEEPSIME), 4); // максимум 9999 мс

return(TRUE);
}

///////////////////////////////
void Dlg_OnCommand (HWND hwnd, int id, HWND hwndCtl, UINT codeNotify) {

switch (id) {
    case IDCANCEL:
        PostQuitMessage(0);
        break;

    case IDC_PROCESSPRIORITYCLASS:
        if (codeNotify == CBN_SELCHANGE) {
            SetPriorityClass(GetCurrentProcess(), (DWORD)
                ComboBox_GetItemData(hwndCtl, ComboBox_GetCurSel(hwndCtl)));
        }
        break;

    case IDC_THREADRELATIVEPRIORITY:
        if (codeNotify == CBN_SELCHANGE) {
            SetThreadPriority(GetCurrentThread(), (DWORD)
                ComboBox_GetItemData(hwndCtl, ComboBox_GetCurSel(hwndCtl)));
        }
        break;

    case IDC_SUSPEND:
        // во избежание взаимной блокировки вызываем EnableWindow
        // до создания потока, который вызовет SuspendThread
        EnableWindow(hwndCtl, FALSE);

        HANDLE hThreadPrimary;
        DuplicateHandle(GetCurrentProcess(), GetCurrentThread(),
            GetCurrentProcess(), &hThreadPrimary,
            THREAD_SUSPEND_RESUME, FALSE, DUPLICATE_SAME_ACCESS);
        DWORD dwThreadID;
```

Рис. 7-1. продолжение

```

        CloseHandle(chBEGINTHREADEX(NULL, 0, ThreadFunc,
            hThreadPrimary, 0, &dwThreadID));
        break;
    }
}

///////////////////////////////
INT_PTR WINAPI Dlg_Proc(HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam) {
    switch (uMsg) {
        chHANDLE_DLGMMSG(hwnd, WM_INITDIALOG, Dlg_OnInitDialog);
        chHANDLE_DLGMMSG(hwnd, WM_COMMAND, Dlg_OnCommand);
    }

    return(FALSE);
}

/////////////////////////////
int WINAPI _tWinMain(HINSTANCE hinstExe, HINSTANCE, LPTSTR pszCmdLine, int) {
    HWND hwnd =
        CreateDialog(hinstExe, MAKEINTRESOURCE(IDD_SCHEDLAB), NULL, Dlg_Proc);
    BOOL fQuit = FALSE;

    while (!fQuit) {
        MSG msg;
        if (PeekMessage(&msg, NULL, 0, 0, PM_REMOVE)) {

            // IsDialogMessage позволяет вовремя реагировать на ввод с клавиатуры
            if (!IsDialogMessage(hwnd, &msg)) {

                if (msg.message == WM_QUIT) {
                    fQuit = TRUE; // принято сообщение WM_QUIT, выходим из цикла
                } else {
                    // принято сообщение, отличное от WM_QUIT;
                    // интерпретируем его и передаем адресату
                    TranslateMessage(&msg);
                    DispatchMessage(&msg);
                }
            } // if (!IsDialogMessage())
        }

        // помещаем число в окно списка
        static int s_n = -1;
        TCHAR sz[20];
        wsprintf(sz, TEXT("%u"), ++s_n);
        HWND hwndWork = GetDlgItem(hwnd, IDC_WORK);
        ListBox_SetCurSel(hwndWork, ListBox_AddString(hwndWork, sz));
    }
}

```

см. след. стр.

Рис. 7-1. продолжение

```

// при переполнении списка удаляем лишние строки
while (ListBox_GetCount(hwndWork) > 100)
    ListBox_DeleteString(hwndWork, 0);

// определяем, сколько нужно "спать" потоку
int nSleep = GetDlgItemInt(hwnd, IDC_SLEEPSIZE, NULL, FALSE);
if (chINRANGE(1, nSleep, 9999))
    Sleep(nSleep);
}
}

DestroyWindow(hwnd);
return(0);
}

//////////////////////////// Конец файла /////////////////////

```

Привязка потоков к процессорам

По умолчанию Windows 2000 использует *нежесткую привязку* (soft affinity) потоков к процессорам. Это означает, что при прочих равных условиях, система пытается выполнять поток на том же процессоре, на котором он работал в последний раз. При таком подходе можно повторно использовать данные, все еще хранящиеся в кэше процессора.

В новой компьютерной архитектуре NUMA (Non-Uniform Memory Access) машина состоит из нескольких плат, на каждой из которых находятся четыре процессора и отдельный банк памяти. На следующей иллюстрации показана машина с тремя такими платами, в сумме содержащими 12 процессоров. Отдельный поток может выполняться на любом из этих процессоров.

Машина с архитектурой NUMA

Система NUMA достигает максимальной производительности, если процессоры используют память на своей плате. Если же они обращаются к памяти на другой плате, производительность резко падает. В такой среде желательно, чтобы потоки одного процесса выполнялись на процессорах 0–3, другого — на процессорах 4–7 и т. д. Windows 2000 позволяет подстроиться под эту архитектуру, закрепляя отдельные процессы и потоки за конкретными процессорами. Иначе говоря, Вы можете контролировать, на каких процессорах будут выполняться Ваши потоки. Такая привязка называется *жесткой* (hard affinity).

Количество процессоров система определяет при загрузке, и эта информация становится доступной приложениям через функцию *GetSystemInfo* (о ней — в главе 14). По умолчанию любой поток может выполняться на любом процессоре. Чтобы потоки отдельного процесса работали лишь на некоем подмножестве процессоров, используйте функцию *SetProcessAffinityMask*:

```
BOOL SetProcessAffinityMask(
    HANDLE hProcess,
    DWORD_PTR dwProcessAffinityMask);
```

В первом параметре, *hProcess*, передается описатель процесса. Второй параметр, *dwProcessAffinityMask*, — это битовая маска, указывающая, на каких процессорах могут выполняться потоки данного процесса. Передав, например, значение 0x00000005, мы разрешим процессу использовать только процессоры 0 и 2 (процессоры 1 и 3–31 ему будут недоступны).

Привязка к процессорам наследуется дочерними процессами. Так, если для родительского процесса задана битовая маска 0x00000005, у всех потоков его дочерних процессов будет идентичная маска, и они смогут работать лишь на тех же процессорах. Для привязки целой группы процессов к определенным процессорам используйте объект ядра «задание» (см. главу 5).

Ну и, конечно же, есть функция, позволяющая получить информацию о такой привязке:

```
BOOL GetProcessAffinityMask(
    HANDLE hProcess,
    PDWORD_PTR pdwProcessAffinityMask,
    PDWORD_PTR pdwSystemAffinityMask);
```

Вы передаете ей описатель процесса, а результат возвращается в переменной, на которую указывает *pdwProcessAffinityMask*. Кроме того, функция возвращает системную маску привязки через переменную, на которую ссылается *pdwSystemAffinityMask*. Эта маска указывает, какие процессоры в системе могут выполнять потоки. Таким образом, маска привязки процесса всегда является подмножеством системной маски привязки.

WINDOWS 98 В Windows 98, которая использует только один процессор независимо от того, сколько их на самом деле, *GetProcessAffinityMask* всегда возвращает в обеих переменных значение 1.

До сих пор мы говорили о том, как назначить все потоки процесса определенным процессорам. Но иногда такие ограничения нужно вводить для отдельных потоков. Допустим, в процессе имеется четыре потока, выполняемые на четырехпроцессорной машине. Один из потоков занимается особо важной работой, и Вы, желая повысить вероятность того, что у него всегда будет доступ к вычислительным мощностям, запрещаете остальным потокам использовать процессор 0.

Задать маски привязки для отдельных потоков позволяет функция:

```
DWORD_PTR SetThreadAffinityMask(
    HANDLE hThread,
    DWORD_PTR dwThreadAffinityMask);
```

В параметре *hThread* передается описатель потока, а *dwThreadAffinityMask* определяет процессоры, доступные этому потоку. Параметр *dwThreadAffinityMask* должен

быть корректным подмножеством маски привязки процесса, которому принадлежит данный поток. Функция возвращает предыдущую маску привязки потока. Вот как ограничить три потока из нашего примера процессорами 1, 2 и 3:

```
// поток 0 выполняется только на процессоре 0  
SetThreadAffinityMask(hThread0, 0x00000001);  
  
// потоки 1, 2, 3 выполняются на процессорах 1, 2, 3  
SetThreadAffinityMask(hThread1, 0x0000000E);  
SetThreadAffinityMask(hThread2, 0x0000000E);  
SetThreadAffinityMask(hThread3, 0x0000000E);
```

WINDOWS 98 В Windows 98, которая использует только один процессор независимо от того, сколько их на самом деле, параметр *dwThreadAffinityMask* всегда должен быть равен 1.

При загрузке система тестирует процессоры типа x86 на наличие в них знаменитого «жучка» в операциях деления чисел с плавающей точкой (эта ошибка имеется в некоторых Pentium). Она привязывает поток, выполняющий потенциально сбойную операцию деления, к исследуемому процессору и сравнивает результат с тем, что должно быть на самом деле. Такая последовательность операций выполняется для каждого процессора в машине.



В большинстве сред вмешательство в системную привязку потоков нарушает нормальную работу планировщика, не позволяя ему максимально эффективно распределять вычислительные мощности. Рассмотрим один пример.

Поток	Приоритет	Маска привязки	Результат
A	4	0x00000001	Работает только на процессоре 0
B	8	0x00000003	Работает на процессоре 0 и 1
C	6	0x00000002	Работает только на процессоре 1

Когда поток А пробуждается, планировщик, видя, что тот жестко привязан к процессору 0, подключает его именно к этому процессору. Далее активизируется поток В, который может выполняться на процессорах 0 и 1, и планировщик выделяет ему процессор 1, так как процессор 0 уже занят. Пока все нормально.

Но вот пробуждается поток С, привязанный к процессору 1. Этот процессор уже занят потоком В с приоритетом 8, а значит, поток С, приоритет которого равен 6, не может его вытеснить. Он, конечно, мог бы вытеснить поток А (с приоритетом 4) с процессора 0, но у него нет прав на использование этого процессора.

Ограничение потока одним процессором не всегда является лучшим решением. Ведь может оказаться так, что три потока конкурируют за доступ к процессору 0, тогда как процессоры 1, 2 и 3 простоявают. Гораздо лучше сообщить системе, что поток желательно выполнять на определенном процессоре, но, если он занят, его можно переключать на другой процессор.

Указать предпочтительный (идеальный) процессор позволяет функция:

```
DWORD SetThreadIdealProcessor(
    HANDLE hThread,
    DWORD dwIdealProcessor);
```

В параметре *hThread* передается описатель потока. В отличие от функций, которые мы уже рассматривали, параметр *dwIdealProcessor* содержит не битовую маску, а целое значение в диапазоне 0–31, которое указывает предпочтительный процессор для данного потока. Передав в нем константу MAXIMUM_PROCESSORS (в WinNT.h она определена как 32), Вы сообщите системе, что потоку не требуется предпочтительный процессор. Функция возвращает установленный ранее номер предпочтительного процессора или MAXIMUM_PROCESSORS, если таковой процессор не задан.

Привязку к процессорам можно указать в заголовке исполняемого файла. Как ни странно, но подходящего ключа компоновщика на этот случай, похоже, не предусмотрено. Тем не менее Вы можете воспользоваться, например, таким кодом:

```
// загружаем EXE-файл в память
PLOADED_IMAGE pLoadedImage = ImageLoad(szExeName, NULL);

// получаем информацию о текущей загрузочной конфигурации EXE-файла
IMAGE_LOAD_CONFIG_DIRECTORY ilcd;
GetImageConfigInformation(pLoadedImage, &ilcd);

// изменяем маску привязки процесса
ilcd.ProcessAffinityMask = 0x00000003; // нам нужны процессоры 0 и 1

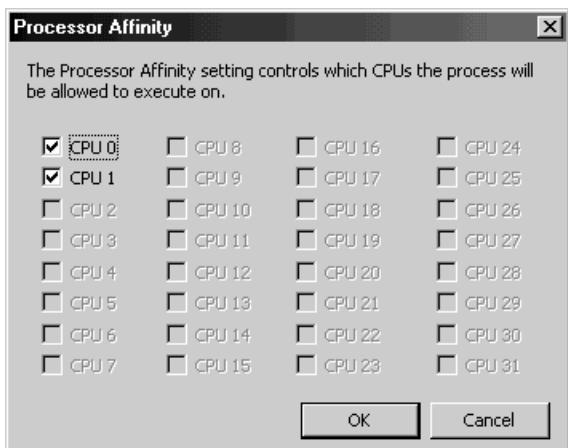
// сохраняем новую информацию о загрузочной конфигурации
SetImageConfigInformation(pLoadedImage, &ilcd);

// выгружаем EXE-файл из памяти
ImageUnload(pLoadedImage);
```

Детально описывать эти функции я не стану — при необходимости Вы найдете их в документации Platform SDK. Кроме того, Вы можете использовать утилиту Image-Cfg.exe, которая позволяет изменять некоторые флаги в заголовке исполняемого модуля. Подсказку по ее применению Вы получите, запустив ImageCfg.exe без ключей.

Указав при запуске ImageCfg ключ –а, Вы сможете изменить маску привязки для приложения. Конечно, все, что делает эта утилита, — вызывает функции, перечисленные в подсказке по ее применению. Обратите внимание на ключ –и, который сообщает системе, что исполняемый файл может выполняться исключительно на однопроцессорной машине.

И, наконец, привязку процесса к процессорам можно изменять с помощью Task Manager в Windows 2000. В многопроцессорных системах в контекстном меню для процесса появляется команда Set Affinity (ее нет на компьютерах с одним процессором). Выбрав эту команду, Вы откроете показанное ниже диалоговое окно и выберете конкретные процессоры для данного процесса.



WINDOWS 2000 При запуске Windows 2000 на машине с процессорами типа x86 можно ограничить число процессоров, используемых системой. В процессе загрузки система считывает файл Boot.ini, который находится в корневом каталоге загрузочного диска. Вот как он выглядит на моем компьютере с двумя процессорами:

```
[boot loader]
timeout=2
default=multi(0)disk(0)rdisk(0)partition(1)\WINNT
[operating systems]
multi(0)disk(0)rdisk(0)partition(1)\WINNT="Windows 2000 Server"
    /fastdetect
multi(0)disk(0)rdisk(0)partition(1)\WINNT="Windows 2000 Server"
    /fastdetect /NumProcs=1
```

Этот файл создается при установке Windows 2000; последнюю запись я добавил сам (с помощью Notepad). Она заставляет систему использовать только один процессор. Ключ /NumProcs=1 — как раз то зелье, которое и вызывает все эти магические превращения. Я пользуюсь им иногда для отладки. (Но обычно работаю со всеми своими процессорами.)

Заметьте, что ключи перенесены на отдельные строки с отступом лишь для удобства чтения. На самом деле ключи и путь от загрузочного раздела жесткого диска должны находиться на одной строке.

Синхронизация потоков в пользовательском режиме

Windows лучше всего работает, когда все потоки могут заниматься своим делом, не взаимодействуя друг с другом. Однако такая ситуация очень редка. Обычно поток создается для выполнения определенной работы, о завершении которой, вероятно, захочет узнать другой поток.

Все потоки в системе должны иметь доступ к системным ресурсам — кучам, последовательным портам, файлам, окнам и т. д. Если один из потоков запросит монопольный доступ к какому-либо ресурсу, другим потокам, которым тоже нужен этот ресурс, не удастся выполнить свои задачи. А с другой стороны, просто недопустимо, чтобы потоки бесконтрольно пользовались ресурсами. Иначе может получиться так, что один поток пишет в блок памяти, из которого другой что-то считывает. Представьте, Вы читаете книгу, а в это время кто-то переписывает текст на открытой Вами странице. Ничего хорошего из этого не выйдет.

Потоки должны взаимодействовать друг с другом в двух основных случаях:

- совместно используя разделяемый ресурс (чтобы не разрушить его);
- когда нужно уведомлять другие потоки о завершении каких-либо операций.

Синхронизации потоков — тематика весьма обширная, и мы рассмотрим ее в этой и следующих главах. Одна новость Вас обрадует: в Windows есть масса средств, упрощающих синхронизацию потоков. Но другая огорчит: точно спрогнозировать, в какой момент потоки будут делать то-то и то-то, крайне сложно. Наш мозг не умеет работать асинхронно; мы обдумываем свои мысли старым добрым способом — одну за другой по очереди. Однако многопоточная среда ведет себя иначе.

С программированием для многопоточной среды я впервые столкнулся в 1992 г. Понапалу я делал уйму ошибок, так что в главах моих книг и журнальных статьях хватало огурцов, связанных с синхронизацией потоков. Сегодня я намного опытнее и действительно считаю, что уж в этой-то книге все безукоризненно (хотя самонадеянности у меня вроде бы поубавилось). Единственный способ освоить синхронизацию потоков — заняться этим на практике. Здесь и в следующих главах я объясню, как работает система и как правильно синхронизировать потоки. Однако Вам придется стойко переносить трудности: приобретая опыт, ошибок не избежать.

Атомарный доступ: семейство *Interlocked*-функций

Большая часть синхронизации потоков связана с *атомарным доступом* (*atomic access*) — монопольным захватом ресурса обращающимся к нему потоком. Возьмем простой пример.

```
// определяем глобальную переменную
long g_x = 0;

DWORD WINAPI ThreadFunc1(PVOID pvParam) {
    g_x++;
    return(0);
}

DWORD WINAPI ThreadFunc2(PVOID pvParam) {
    g_x++;
    return(0);
}
```

Я объявил глобальную переменную *g_x* и инициализировал ее нулевым значением. Теперь представьте, что я создал два потока: один выполняет *ThreadFunc1*, другой — *ThreadFunc2*. Код этих функций идентичен: обе увеличивают значение глобальной переменной *g_x* на 1. Поэтому Вы, наверное, подумали: когда оба потока завершат свою работу, значение *g_x* будет равно 2. Так ли это? Может быть. При таком коде заранее сказать, каким будет конечное значение *g_x*, нельзя. И вот почему. Допустим, компилятор сгенерировал для строки, увеличивающей *g_x* на 1, следующий код:

```
MOV EAX, [g_x]      ; значение из g_x помещается в регистр
INC EAX            ; значение регистра увеличивается на 1
MOV [g_x], EAX     ; значение из регистра помещается обратно в g_x
```

Вряд ли оба потока будут выполнять этот код в одно и то же время. Если они будут делать это по очереди — сначала один, потом другой, тогда мы получим такую картину:

```
MOV EAX, [g_x]      ; поток 1: в регистр помещается 0
INC EAX            ; поток 1: значение регистра увеличивается на 1
MOV [g_x], EAX     ; поток 1: значение 1 помещается в g_x

MOV EAX, [g_x]      ; поток 2: в регистр помещается 1
INC EAX            ; поток 2: значение регистра увеличивается до 2
MOV [g_x], EAX     ; поток 2: значение 2 помещается в g_x
```

После выполнения обоих потоков значение *g_x* будет равно 2. Это просто замечательно и как раз то, что мы ожидали: взял переменную с нулевым значением, дважды увеличили ее на 1 и получили в результате 2. Прекрасно. Но постойте-ка, ведь Windows — это среда, которая поддерживает многопоточность и вытесняющую многозадачность. Значит, процессорное время в любой момент может быть отнято у одного потока и передано другому. Тогда код, приведенный мной выше, может выполниться и таким образом:

```
MOV EAX, [g_x]      ; поток 1: в регистр помещается 0
INC EAX            ; поток 1: значение регистра увеличивается на 1

MOV EAX, [g_x]      ; поток 2: в регистр помещается 0
INC EAX            ; поток 2: значение регистра увеличивается на 1
MOV [g_x], EAX     ; поток 2: значение 1 помещается в g_x

MOV [g_x], EAX     ; поток 1: значение 1 помещается в g_x
```

А если код будет выполняться именно так, конечное значение `g_x` окажется равным 1, а не 2, как мы думали! Довольно пугающе, особенно если учесть, как мало у нас рычагов управления планировщиком. Фактически, даже при сотне потоков, которые выполняют функции, идентичные нашей, в конечном итоге вполне можно получить в `g_x` все ту же единицу! Очевидно, что в таких условиях работать просто нельзя. Мы вправе ожидать, что, дважды увеличив 0 на 1, при любых обстоятельствах получим 2. Кстати, результаты могут зависеть от того, как именно компилятор генерирует машинный код, а также от того, как процессор выполняет этот код и сколько процессоров установлено в машине. Это объективная реальность, в которой мы не в состоянии что-либо изменить. Однако в Windows есть ряд функций, которые (при правильном их использовании) гарантируют корректные результаты выполнения кода.

Решение этой проблемы должно быть простым. Все, что нам нужно, — это способ, гарантирующий приращение значения переменной на уровне атомарного доступа, т. е. без прерывания другими потоками. Семейство *Interlocked*-функций как раз и дает нам ключ к решению подобных проблем. Большинство разработчиков программного обеспечения недооценивает эти функции, а ведь они невероятно полезны и очень просты для понимания. Все функции из этого семейства манипулируют переменными на уровне атомарного доступа. Взгляните на *InterlockedExchangeAdd*:

```
LONG InterlockedExchangeAdd(
    PLONG plAddend,
    LONG lIncrement);
```

Что может быть проще? Вы вызываете эту функцию, передавая адрес переменной типа LONG и указываете добавляемое значение. *InterlockedExchangeAdd* гарантирует, что операция будет выполнена атомарно. Перепишем наш код вот так:

```
// определяем глобальную переменную
long g_x = 0;

DWORD WINAPI ThreadFunc1(PVOID pvParam) {
    InterlockedExchangeAdd(&g_x, 1);
    return(0);
}

DWORD WINAPI ThreadFunc2(PVOID pvParam) {
    InterlockedExchangeAdd(&g_x, 1);
    return(0);
}
```

Теперь Вы можете быть уверены, что конечное значение `g_x` будет равно 2. Ну, Вам уже лучше? Заметьте: в любом потоке, где нужно модифицировать значение разделяемой (общей) переменной типа LONG, следует пользоваться лишь *Interlocked*-функциями и никогда не прибегать к стандартным операторам языка С:

```
// переменная типа LONG, используемая несколькими потоками
LONG g_x;
:
// неправильный способ увеличения переменной типа LONG
g_x++;
:
// правильный способ увеличения переменной типа LONG
InterlockedExchangeAdd(&g_x, 1);
```

Как же работают *Interlocked*-функции? Ответ зависит от того, какую процессорную платформу Вы используете. На компьютерах с процессорами семейства x86 эти функции выдают по шине аппаратный сигнал, не давая другому процессору обратиться по тому же адресу памяти. На платформе Alpha *Interlocked*-функции действуют примерно так:

1. Устанавливают специальный битовый флаг процессора, указывающий, что данный адрес памяти сейчас занят.
2. Считывают значение из памяти в регистр.
3. Изменяют значение в регистре.
4. Если битовый флаг сброшен, повторяют операции, начиная с п. 2. В ином случае значение из регистра помещается обратно в память.

Вас, наверное, удивило, с какой это стати битовый флаг может оказаться сброшенным? Все очень просто. Его может сбросить другой процессор в системе, пытаясь модифицировать тот же адрес памяти, а это заставляет *Interlocked*-функции вернуться в п. 2.

Вовсе не обязательно вникать в детали работы этих функций. Вам нужно знать лишь одно: они гарантируют монопольное изменение значений переменных независимо от того, как именно компилятор генерирует код и сколько процессоров установлено в компьютере. Однако Вы должны позаботиться о выравнивании адресов переменных, передаваемых этим функциям, иначе они могут потерпеть неудачу. (О выравнивании данных я расскажу в главе 13.)

Другой важный аспект, связанный с *Interlocked*-функциями, состоит в том, что они выполняются чрезвычайно быстро. Вызов такой функции обычно требует не более 50 тактов процессора, и при этом не происходит перехода из пользовательского режима в режим ядра (а он отнимает не менее 1000 тактов).

Кстати, *InterlockedExchangeAdd* позволяет не только увеличить, но и уменьшить значение — просто передайте во втором параметре отрицательную величину. *InterlockedExchangeAdd* возвращает исходное значение в **plAddend*.

Вот еще две функции из этого семейства:

```
LONG InterlockedExchange(
    PLONG plTarget,
    LONG lValue);

PVOID InterlockedExchangePointer(
    PVOID* ppvTarget,
    PVOID pvValue);
```

InterlockedExchange и *InterlockedExchangePointer* монопольно заменяют текущее значение переменной типа LONG, адрес которой передается в первом параметре, на значение, передаваемое во втором параметре. В 32-разрядном приложении обе функции работают с 32-разрядными значениями, но в 64-разрядной программе первая оперирует с 32-разрядными значениями, а вторая — с 64-разрядными. Все функции возвращают исходное значение переменной. *InterlockedExchange* чрезвычайно полезна при реализации спин-блокировки (spinlock):

```
// глобальная переменная, используемая как индикатор того, занят ли разделяемый ресурс
BOOL g_fResourceInUse = FALSE;
:
```

```

void Func1() {
    // ожидаем доступа к ресурсу
    while (InterlockedExchange(&g_fResourceInUse, TRUE) == TRUE)
        Sleep(0);

    // получаем ресурс в свое распоряжение
    :
    // доступ к ресурсу больше не нужен
    InterlockedExchange(&g_fResourceInUse, FALSE);
}

```

В этой функции постоянно «крутится» цикл *while*, в котором переменной *g_fResourceInUse* присваивается значение TRUE и проверяется ее предыдущее значение. Если оно было равно FALSE, значит, ресурс не был занят, но вызывающий поток только что занял его; на этом цикл завершается. В ином случае (значение было равно TRUE) ресурс занимал другой поток, и цикл повторяется.

Если бы подобный код выполнялся и другим потоком, его цикл *while* работал бы до тех пор, пока значение переменной *g_fResourceInUse* вновь не изменилось бы на FALSE. Вызов *InterlockedExchange* в конце функции показывает, как вернуть переменной *g_fResourceInUse* значение FALSE.

Применяйте эту методику с крайней осторожностью, потому что процессорное время при спин-блокировке тратится впустую. Процессору приходится постоянно сравнивать два значения, пока одно из них не будет «волшебным образом» изменено другим потоком. Учтите: этот код подразумевает, что все потоки, использующие спин-блокировку, имеют одинаковый уровень приоритета. К тому же, Вам, наверное, придется отключить динамическое повышение приоритета этих потоков (вызовом *SetProcessPriorityBoost* или *SetThreadPriorityBoost*).

Вы должны позаботиться и о том, чтобы переменная — индикатор блокировки и данные, защищаемые такой блокировкой, не попали в одну кэш-линию (о кэш-линиях я расскажу в следующем разделе). Иначе процессор, использующий ресурс, будет конкурировать с любыми другими процессорами, которые пытаются обратиться к тому же ресурсу. А это отрицательно скажется на быстродействии.

Избегайте спин-блокировки на однопроцессорных машинах. «Крутясь» в цикле, поток впустую трахнит драгоценное процессорное время, не давая другому потоку изменить значение переменной. Применение функции *Sleep* в цикле *while* несколько улучшает ситуацию. С ее помощью Вы можете отправлять свой поток в сон на некий случайный отрезок времени и после каждой безуспешной попытки обратиться к ресурсу увеличивать этот отрезок. Тогда потоки не будут зря отнимать процессорное время. В зависимости от ситуации вызов *Sleep* можно убрать или заменить на вызов *SwitchToThread* (эта функция в Windows 98 не доступна). Очень жаль, но, по-видимому, Вам придется действовать здесь методом проб и ошибок.

Спин-блокировка предполагает, что защищенный ресурс не бывает занят надолго. И тогда эффективнее делать так: выполнять цикл, переходить в режим ядра и ждать. Многие разработчики повторяют цикл некоторое число раз (скажем, 4000) и, если ресурс к тому времени не освободился, переводят поток в режим ядра, где он спит, ожидая освобождения ресурса (и не расходуя процессорное время). По такой схеме реализуются критические секции (critical sections).

Спин-блокировка полезна на многопроцессорных машинах, где один поток может «крутиться» в цикле, а второй — работать на другом процессоре. Но даже в таких условиях надо быть осторожным. Вряд ли Вам понравится, если поток надолго вой-

дет в цикл, ведь тогда он будет впустую тратить процессорное время. О спин-блокировке мы еще поговорим в этой главе. Кроме того, в главе 10 я покажу, как использовать спин-блокировку на практике.

Последняя пара *Interlocked*-функций выглядит так:

```
PVOID InterlockedCompareExchange(
    PLONG plDestination,
    LONG lExchange,
    LONG lComparand);

PVOID InterlockedCompareExchangePointer(
    PVOID* ppvDestination,
    PVOID pvExchange,
    PVOID pvComparand);
```

Они выполняют операцию сравнения и присвоения на уровне атомарного доступа. В 32-разрядном приложении обе функции работают с 32-разрядными значениями, но в 64-разрядном приложении *InterlockedCompareExchange* используется для 32-разрядных значений, а *InterlockedCompareExchangePointer* — для 64-разрядных. Вот как они действуют, если представить это в псевдокоде:

```
LONG InterlockedCompareExchange(PLONG plDestination,
    LONG lExchange, LONG lComparand) {

    LONG lRet = *plDestination; // исходное значение

    if (*plDestination == lComparand)
        *plDestination = lExchange;
    return(lRet);
}
```

Функция сравнивает текущее значение переменной типа *LONG* (на которую указывает параметр *plDestination*) со значением, передаваемым в параметре *lComparand*. Если значения совпадают, **plDestination* получает значение параметра *lExchange*; в ином случае **plDestination* остается без изменений. Функция возвращает исходное значение **plDestination*. И не забывайте, что все эти действия выполняются как единая атомарная операция.

Обратите внимание на отсутствие *Interlocked*-функции, позволяющей просто считывать значение какой-то переменной, не меняя его. Она и не нужна. Если один поток модифицирует переменную с помощью какой-либо *Interlocked*-функции в тот момент, когда другой читает содержимое той же переменной, ее значение, прочитанное вторым потоком, всегда будет достоверным. Он получит либо исходное, либо измененное значение переменной. Поток, конечно, не знает, какое именно значение он считал, но главное, что оно корректно и не является некоей произвольной величиной. В большинстве приложений этого вполне достаточно.

Interlocked-функции можно также использовать в потоках различных процессов для синхронизации доступа к переменной, которая находится в разделяемой области памяти, например в проекции файла. (Правильное применение *Interlocked*-функций демонстрирует несколько программ-примеров из главы 9.)

В Windows есть и другие функции из этого семейства, но ничего нового по сравнению с тем, что мы уже рассмотрели, они не делают. Вот еще две из них:

```
LONG InterlockedIncrement(PLONG plAddend);
```

```
LONG InterlockedDecrement(PLONG plAddend);
```

InterlockedExchangeAdd полностью заменяет обе эти устаревшие функции. Новая функция умеет добавлять и вычитать произвольные значения, а функции *InterlockedIncrement* и *InterlockedDecrement* увеличивают и уменьшают значения только на 1.

Кэш-линии

Если Вы хотите создать высокоэффективное приложение, работающее на многопроцессорных машинах, то просто обязаны уметь пользоваться кэш-линиями процессора (CPU cache lines). Когда процессору нужно считать из памяти один байт, он извлекает не только его, но и столько смежных байтов, сколько требуется для заполнения кэш-линии. Такие линии состоят из 32 или 64 байтов (в зависимости от типа процессора) и всегда выравниваются по границам, кратным 32 или 64 байтам. Кэш-линии предназначены для повышения быстродействия процессора. Обычно приложение работает с набором смежных байтов, и, если эти байты уже находятся в кэше, процессору не приходится снова обращаться к шине памяти, что обеспечивает существенную экономию времени.

Однако кэш-линии сильно усложняют обновление памяти в многопроцессорной среде. Вот небольшой пример:

1. Процессор 1 считывает байт, извлекая этот и смежные байты в свою кэш-линию.
2. Процессор 2 считывает тот же байт, а значит, и тот же набор байтов, что и процессор 1; извлеченные байты помещаются в кэш-линию процессора 2.
3. Процессор 1 модифицирует байт памяти, и этот байт записывается в его кэш-линию. Но эти изменения еще не записаны в оперативную память.
4. Процессор 2 повторно считывает тот же байт. Поскольку он уже помещен в кэш-линию этого процессора, последний не обращается к памяти и, следовательно, не «видит» новое значение данного байта.

Такой сценарий был бы настоящей катастрофой. Но разработчики чипов прекрасно осведомлены об этой проблеме и учитывают ее при проектировании своих процессоров. В частности, когда один из процессоров модифицирует байты в своей кэш-линии, об этом оповещаются другие процессоры, и содержимое их кэш-линий объявляется недействительным. Таким образом, в примере, приведенном выше, после изменения байта процессором 1, кэш процессора 2 был бы объявлен недействительным. На этапе 4 процессор 1 должен сбросить содержимое своего кэша в оперативную память, а процессор 2 — повторно обратиться к памяти и вновь заполнить свою кэш-линию. Как видите, кэш-линии, которые, как правило, увеличивают быстродействие процессора, в многопроцессорных машинах могут стать причиной снижения производительности.

Все это означает, что Вы должны группировать данные своего приложения в блоки размером с кэш-линию и выравнивать их по тем же правилам, которые применяются к кэш-линиям. Ваша цель — добиться того, чтобы различные процессоры обращались к разным адресам памяти, отделенным друг от друга по крайней мере границей кэш-линии. Кроме того, Вы должны отделить данные «только для чтения» (или редко используемые данные) от данных «для чтения и записи». И еще Вам придется позаботиться о группировании тех блоков данных, обращение к которым происходит примерно в одно и то же время.

Вот пример плохо продуманной структуры данных:

```
struct CUSTINFO {
    DWORD    dwCustomerID;           // в основном "только для чтения"
    int      nBalanceDue;          // для чтения и записи
    char     szName[100];           // в основном "только для чтения"
    FILETIME ftLastOrderDate;      // для чтения и записи
};
```

А это усовершенствованная версия той же структуры:

```
// определяем размер кэш-линии используемого процессора
#ifndef _X86_
#define CACHE_ALIGN 32
#endif
#ifndef _ALPHA_
#define CACHE_ALIGN 64
#endif
#ifndef _IA64_
#define CACHE_ALIGN ???
#endif

#define CACHE_PAD(Name, BytesSoFar) \
    BYTE Name[CACHE_ALIGN - ((BytesSoFar) % CACHE_ALIGN)]
```



```
struct CUSTINFO {
    DWORD    dwCustomerID;           // в основном "только для чтения"
    char     szName[100];           // в основном "только для чтения"

    // принудительно помещаем следующие элементы в другую кэш-линию
    CACHE_PAD(bPad1, sizeof(DWORD) + 100);

    int      nBalanceDue;          // для чтения и записи
    FILETIME ftLastOrderDate;      // для чтения и записи

    // принудительно помещаем следующую структуру в другую кэш-линию
    CACHE_PAD(bPad2, sizeof(int) + sizeof(FILETIME));
};
```

Макрос CACHE_ALIGN неплох, но не идеален. Проблема в том, что байтовый размер каждого элемента придется вводить в макрос вручную, а при добавлении, перемещении или удалении элемента структуры — еще и модифицировать вызов макроса CACHE_PAD. В следующих версиях компилятор Microsoft C/C++ будет поддерживать новый синтаксис, упрощающий выравнивание элементов структур. Это будет что-то вроде `__declspec(align(32))`.



Лучше всего, когда данные используются единственным потоком (самый простой способ добиться этого — применять параметры функций и локальные переменные) или одним процессором (это реализуется привязкой потока к определенному процессору). Если Вы пойдете по такому пути, можете вообще забыть о проблемах, связанных с кэш-линиями.

Более сложные методы синхронизации потоков

Interlocked-функции хороши, когда требуется монопольно изменить всего одну переменную. С них и надо начинать. Но реальные программы имеют дело со структурами данных, которые гораздо сложнее единственной 32- или 64-битной переменной. Чтобы получить доступ на атомарном уровне к таким структурам данных, забудьте об *Interlocked*-функциях и используйте другие механизмы, предлагаемые Windows.

В предыдущем разделе я подчеркнул неэффективность спин-блокировки на однопроцессорных машинах и обратил Ваше внимание на то, что со спин-блокировкой надо быть осторожным даже в многопроцессорных системах. Хочу еще раз напомнить, что основная причина связана с недопустимостью пустой траты процессорного времени. Так что нам нужен механизм, который позволил бы потоку, ждущему освобождения разделяемого ресурса, не расходовать процессорное время.

Когда поток хочет обратиться к разделяемому ресурсу или получить уведомление о некоем «особом событии», он должен вызвать определенную функцию операционной системы и передать ей параметры, сообщающие, чего именно он ждет. Как только операционная система обнаружит, что ресурс освободился или что «особое событие» произошло, эта функция вернет управление потоку, и тот снова будет включен в число планируемых. (Это не значит, что поток тут же начнет выполняться; система подключит его к процессору по правилам, описанным в предыдущей главе.)

Пока ресурс занят или пока не произошло «особое событие», система переводит поток в ждущий режим, исключая его из числа планируемых, и берет на себя роль агента, действующего в интересах спящего потока. Она выведет его из ждущего режима, когда освободится нужный ресурс или произойдет «особое событие».

Большинство потоков почти постоянно находится в ждущем режиме. И когда система обнаруживает, что все потоки уже несколько минут спят, срабатывает механизм управления электропитанием.

Худшее, что можно сделать

Если бы синхронизирующих объектов не было, а операционная система не умела отслеживать особые события, потоку пришлось бы самостоятельно синхронизировать себя с ними, применяя метод, который я как раз и собираюсь продемонстрировать. Но поскольку в операционную систему встроена поддержка синхронизации объектов, *никогда не применяйте* этот метод.

Суть его в том, что поток синхронизирует себя с завершением какой-либо задачи в другом потоке, постоянно просматривая значение переменной, доступной обоим потокам. Возьмем пример:

```
volatile BOOL g_fFinishedCalculation = FALSE;

int WINAPI WinMain(...) {
    CreateThread(..., RecalcFunc, ...);
    :
    // ждем завершения пересчета
    while (!g_fFinishedCalculation)
        ;
    :
}
```

см. след. стр.

```
DWORD WINAPI RecalcFunc(PVOID pvParam) {
    // выполняем пересчет
    :
    g_fFinishedCalculation = TRUE;
    return(0);
}
```

Как видите, первичный поток (он исполняет функцию *WinMain*) при синхронизации по такому событию, как завершение функции *RecalcFunc*, никогда не впадает в спячку. Поэтому система по-прежнему выделяет ему процессорное время за счет других потоков, занимающихся чем-то более полезным.

Другая проблема, связанная с подобным методом опроса, в том, что булева переменная *g_fFinishedCalculation* может не получить значения TRUE — например, если у первичного потока более высокий приоритет, чем у потока, выполняющего функцию *RecalcFunc*. В этом случае система никогда не предоставит процессорное время потоку *RecalcFunc*, а он никогда не выполнит оператор, присваивающий значение TRUE переменной *g_fFinishedCalculation*. Если бы мы не опрашивали поток, выполняющий функцию *WinMain*, а просто отправили в спячку, это позволило бы системе отдать его долю процессорного времени потокам с более низким приоритетом, в частности потоку *RecalcFunc*.

Вполне допускаю, что опрос иногда удобен. В конце концов, именно это и делается при спин-блокировке. Но есть два способа его реализации: корректный и некорректный. Общее правило таково: избегайте применения спин-блокировки и опроса. Вместо этого пользуйтесь функциями, которые переводят Ваш поток в состояние ожидания до освобождения нужного ему ресурса. Как это правильно сделать, я объясню в следующем разделе.

Прежде всего позвольте обратить Ваше внимание на одну вещь: в начале приведенного выше фрагмента кода я использовал спецификатор *volatile* — без него работа моей программы просто немыслима. Он сообщает компилятору, что переменная может быть изменена извне приложения — операционной системой, аппаратным устройством или другим потоком. Точнее, спецификатор *volatile* заставляет компилятор исключить эту переменную из оптимизации и всегда перезагружать ее значение из памяти. Представьте, что компилятор сгенерировал следующий псевдокод для оператора *while* из предыдущего фрагмента кода:

```
MOV      Reg0, [g_fFinishedCalculation] ; копируем значение в регистр
Label:   TEST Reg0, 0                  ; равно ли оно нулю?
JMP      Reg0 == 0, Label            ; в регистре находится 0, повторяем цикл
...                                ; в регистре находится ненулевое значение
                                      ; (выходим из цикла)
```

Если бы я не определил булеву переменную как *volatile*, компилятор мог бы оптимизировать наш код на С именно так. При этом компилятор загружал бы ее значение в регистр процессора только раз, а потом сравнивал бы искомое значение с содержимым регистра. Конечно, такая оптимизация повышает быстродействие, поскольку позволяет избежать постоянного считывания значения из памяти; оптимизирующий компилятор скорее всего сгенерирует код именно так, как я показал. Но тогда наш поток войдет в бесконечный цикл и никогда не проснется. Кстати, если структура определена как *volatile*, таковыми становятся и все ее элементы, т. е. при каждом обращении оничитываются из памяти.

Вас, наверное, заинтересовало, а не следует ли объявить как *volatile* и мою переменную *g_fResourceInUse* в примере со спин-блокировкой. Отвечаю: нет, потому что она передается *Interlocked*-функции по ссылке, а не по значению. Передача переменной по ссылке всегда заставляет функцию считывать ее значение из памяти, и оптимизатор никак не влияет на это.

Критические секции

Критическая секция (critical section) — это небольшой участок кода, требующий монопольного доступа к каким-то общим данным. Она позволяет сделать так, чтобы единовременно только один поток получал доступ к определенному ресурсу. Естественно, система может в любой момент вытеснить Ваш поток и подключить к процессору другой, но ни один из потоков, которым нужен занятый Вами ресурс, не получит процессорное время до тех пор, пока Ваш поток не выйдет за границы критической секции.

Вот пример кода, который демонстрирует, что может произойти без критической секции:

```
const int MAX_TIMES = 1000;
int g_nIndex = 0;
DWORD g_dwTimes[MAX_TIMES];

DWORD WINAPI FirstThread(PVOID pvParam) {

    while (g_nIndex < MAX_TIMES) {
        g_dwTimes[g_nIndex] = GetTickCount();
        g_nIndex++;
    }
    return(0);
}

DWORD WINAPI SecondThread(PVOID pvParam) {

    while (g_nIndex < MAX_TIMES) {
        g_nIndex++;
        g_dwTimes[g_nIndex - 1] = GetTickCount();
    }
    return(0);
}
```

Здесь предполагается, что функции обоих потоков дают одинаковый результат, хоть они и закодированы с небольшими различиями. Если бы исполнялась только функция *FirstThread*, она заполнила бы массив *g_dwTimes* набором чисел с возрастающими значениями. Это верно и в отношении *SecondThread* — если бы она тоже исполнялась независимо. В идеале обе функции даже при одновременном выполнении должны бы по-прежнему заполнять массив тем же набором чисел. Но в нашем коде возникает проблема: массив *g_dwTimes* не будет заполнен, как надо, потому что функции обоих потоков одновременно обращаются к одним и тем же глобальным переменным. Вот как это может произойти.

Допустим, мы только что начали исполнение обоих потоков в системе с одним процессором. Первым включился в работу второй поток, т. е. функция *SecondThread* (что вполне вероятно), и только она успела увеличить счетчик *g_nIndex* до 1, как си-

стема вытеснила ее поток и перешла к исполнению *FirstThread*. Та заносит в *g_dwTimes[1]* показания системного времени, и процессор вновь переключается на исполнение второго потока. *SecondThread* теперь присваивает элементу *g_dwTimes[1 – 1]* новые показания системного времени. Поскольку эта операция выполняется позже, новые показания, естественно, выше, чем записанные в элемент *g_dwTimes[1]* функцией *FirstThread*. Отметьте также, что сначала заполняется первый элемент массива и только потом нулевой. Таким образом, данные в массиве оказываются ошибочными.

Согласен, пример довольно надуманный, но, чтобы привести реалистичный, нужно минимум несколько страниц кода. Важно другое: теперь Вы легко представите, что может произойти в действительности. Возьмем пример с управлением связанным списком объектов. Если доступ к связанному списку не синхронизирован, один поток может добавить элемент в список в тот момент, когда другой поток пытается найти в нем какой-то элемент. Ситуация станет еще более угрожающей, если оба потока одновременно добавят в список новые элементы. Так что, используя критические секции, можно и нужно координировать доступ потоков к структурам данных.

Теперь, когда Вы видите все «подводные камни», попробуем исправить этот фрагмент кода с помощью критической секции:

```
const int MAX_TIMES = 1000;
int g_nIndex = 0;
DWORD g_dwTimes[MAX_TIMES];
CRITICAL_SECTION g_cs;

DWORD WINAPI FirstThread(PVOID pvParam) {

    for (BOOL fContinue = TRUE; fContinue; ) {
        EnterCriticalSection(&g_cs);
        if (g_nIndex < MAX_TIMES) {
            g_dwTimes[g_nIndex] = GetTickCount();
            g_nIndex++;
        } else fContinue = FALSE;
        LeaveCriticalSection(&g_cs);
    }
    return(0);
}

DWORD WINAPI SecondThread(PVOID pvParam) {

    for (BOOL fContinue = TRUE; fContinue; ) {
        EnterCriticalSection(&g_cs);
        if (g_nIndex < MAX_TIMES) {
            g_nIndex++;
            g_dwTimes[g_nIndex - 1] = GetTickCount();
        } else fContinue = FALSE;
        LeaveCriticalSection(&g_cs);
    }
    return(0);
}
```

Я создал экземпляр структуры данных **CRITICAL_SECTION** — *g_cs*, а потом «обернул» весь код, работающий с разделяемым ресурсом (в нашем примере это строки с *g_nIndex* и *g_dwTimes*), вызовами *EnterCriticalSection* и *LeaveCriticalSection*. Заметьте, что при вызовах этих функций я передаю адрес *g.cs*.

Запомните несколько важных вещей. Если у Вас есть ресурс, разделяемый несколькими потоками, Вы должны создать экземпляр структуры CRITICAL_SECTION. Так как я пишу эти строки в самолете, позвольте провести следующую аналогию. Структура CRITICAL_SECTION похожа на туалетную кабинку в самолете, а данные, которые нужно защитить, — на унитаз. Туалетная кабинка (критическая секция) в самолете очень маленькая, и единовременно в ней может находиться только один человек (поток), пользующийся унитазом (защищенным ресурсом).

Если у Вас есть ресурсы, всегда используемые вместе, Вы можете поместить их в одну кабинку — единственная структура CRITICAL_SECTION будет охранять их всех. Но если ресурсы не всегда используются вместе (например, потоки 1 и 2 работают с одним ресурсом, а потоки 1 и 3 — с другим), Вам придется создать им по отдельной кабинке, или структуре CRITICAL_SECTION.

Теперь в каждом участке кода, где Вы обращаетесь к разделяемому ресурсу, вызывайте *EnterCriticalSection*, передавая ей адрес структуры CRITICAL_SECTION, которая выделена для этого ресурса. Иными словами, поток, желая обратиться к ресурсу, должен сначала убедиться, нет ли на двери кабинки знака «занято». Структура CRITICAL_SECTION идентифицирует кабинку, в которую хочет войти поток, а функция *EnterCriticalSection* — тот инструмент, с помощью которого он узнает, свободна или занята кабинка. *EnterCriticalSection* допустит вызвавший ее поток в кабинку, если определит, что та свободна. В ином случае (кабинка занята) *EnterCriticalSection* заставит его ждать, пока она не освободится.

Поток, покидая участок кода, где он работал с защищенным ресурсом, должен вызвать функцию *LeaveCriticalSection*. Тем самым он уведомляет систему о том, что кабинка с данным ресурсом освободилась. Если Вы забудете это сделать, система будет считать, что ресурс все еще занят, и не позволит обратиться к нему другим ждущим потокам. То есть Вы вышли из кабинки и оставили на двери знак «занято».



Самое сложное — запомнить, что любой участок кода, работающего с разделяемым ресурсом, нужно заключить в вызовы функций *EnterCriticalSection* и *LeaveCriticalSection*. Если Вы забудете сделать это хотя бы в одном месте, ресурс может быть поврежден. Так, если в *FirstThread* убрать вызовы *EnterCriticalSection* и *LeaveCriticalSection*, содержимое переменных *g_nIndex* и *g_dwTimes* станет некорректным — даже несмотря на то что в *SecondThread* функции *EnterCriticalSection* и *LeaveCriticalSection* вызываются правильно.

Забыв вызвать эти функции, Вы уподобитесь человеку, который рвется в туалетную кабинку, не обращая внимания на то, есть в ней кто-нибудь или нет. Поток пробивает себе путь к ресурсу и берется им манипулировать. Как Вы прекрасно понимаете, стоит лишь одному потоку проявить такую «грубость», и Ваш ресурс станет кучкой бесполезных байтов.

Применяйте критические секции, если Вам не удается решить проблему синхронизации за счет *Interlocked*-функций. Преимущество критических секций в том, что они просты в использовании и выполняются очень быстро, так как реализованы на основе *Interlocked*-функций. А главный недостаток — нельзя синхронизировать потоки в разных процессах. Однако в главе 10 я продемонстрирую Вам свой синхронизирующий объект, который я назвал оптексом. На его примере Вы увидите, как реализуются критические секции на уровне операционной системы и как этот объект работает с потоками разных процессов.

Критические секции: важное дополнение

Теперь, когда у Вас появилось общее представление о критических секциях (зачем они нужны и как с их помощью можно монопольно распоряжаться разделяемым ресурсом), давайте повнимательнее приглядимся к тому, как они устроены. Начнем со структуры CRITICAL_SECTION. Вы не найдете ее в Platform SDK — о ней нет даже упоминания. В чем дело?

Хотя CRITICAL_SECTION не относится к недокументированным структурам, Microsoft полагает, что Вам незачем знать, как она устроена. И это правильно. Для нас она является своего рода черным ящиком: сама структура известна, а ее элементы — нет. Конечно, поскольку CRITICAL_SECTION — не более чем одна из структур, мы можем сказать, из чего она состоит, изучив заголовочные файлы. (CRITICAL_SECTION определена в файле WinNT.h как RTL_CRITICAL_SECTION, а тип структуры RTL_CRITICAL_SECTION определен в файле WinBase.h.) Но никогда не пишите код, прямо ссылающийся на ее элементы.

Вы работаете со структурой CRITICAL_SECTION исключительно через функции Windows, передавая им адрес соответствующего экземпляра этой структуры. Функции сами знают, как обращаться с ее элементами, и гарантируют, что она всегда будет в согласованном состоянии. Так что теперь мы перейдем к рассмотрению этих функций.

Обычно структуры CRITICAL_SECTION создаются как глобальные переменные, доступные всем потокам процесса. Но ничто не мешает нам создавать их как локальные переменные или переменные, динамически размещаемые в куче. Есть только два условия, которые надо соблюдать. Во-первых, все потоки, которым может понадобиться ресурс, должны знать адрес структуры CRITICAL_SECTION, которая защищает этот ресурс. Вы можете получить ее адрес, используя любой из существующих механизмов. Во-вторых, элементы структуры CRITICAL_SECTION следует инициализировать до обращения какого-либо потока к защищенному ресурсу. Структура инициализируется вызовом:

```
VOID InitializeCriticalSection(PCRITICAL_SECTION pcs);
```

Эта функция инициализирует элементы структуры CRITICAL_SECTION, на которую указывает параметр *pcs*. Поскольку вся работа данной функции заключается в инициализации нескольких переменных-членов, она не дает сбоев и поэтому ничего не возвращает (*void*). *InitializeCriticalSection* должна быть вызвана до того, как один из потоков обратится к *EnterCriticalSection*. В документации Platform SDK недвусмысленно сказано, что попытка воспользоваться неинициализированной критической секцией даст непредсказуемые результаты.

Если Вы знаете, что структура CRITICAL_SECTION больше не понадобится ни одному потоку, удалите ее, вызвав *DeleteCriticalSection*:

```
VOID DeleteCriticalSection(PCRITICAL_SECTION pcs);
```

Она сбрасывает все переменные-члены внутри этой структуры. Естественно, нельзя удалять критическую секцию в тот момент, когда ею все еще пользуется какой-либо поток. Об этом нас предупреждают и в документации Platform SDK.

Участок кода, работающий с разделяемым ресурсом, предваряется вызовом:

```
VOID EnterCriticalSection(PCRITICAL_SECTION pcs);
```

Первое, что делает *EnterCriticalSection*, — исследует значения элементов структуры CRITICAL_SECTION. Если ресурс занят, в них содержатся сведения о том, какой поток пользуется ресурсом. *EnterCriticalSection* выполняет следующие действия.

- Если ресурс свободен, *EnterCriticalSection* модифицирует элементы структуры, указывая, что вызывающий поток занимает ресурс, после чего немедленно возвращает управление, и поток продолжает свою работу (получив доступ к ресурсу).
- Если значения элементов структуры свидетельствуют, что ресурс уже захвачен вызывающим потоком, *EnterCriticalSection* обновляет их, отмечая тем самым, сколько раз подряд этот поток захватил ресурс, и немедленно возвращает управление. Такая ситуация бывает нечасто — лишь тогда, когда поток два раза подряд вызывает *EnterCriticalSection* без промежуточного вызова *LeaveCriticalSection*.
- Если значения элементов структуры указывают на то, что ресурс занят другим потоком, *EnterCriticalSection* переводит вызывающий поток в режим ожидания. Это потрясающее свойство критических секций: поток, пребывая в ожидании, не тратит ни кванта процессорного времени! Система запоминает, что данный поток хочет получить доступ к ресурсу, и — как только поток, занимавший этот ресурс, вызывает *LeaveCriticalSection* — вновь начинает выделять нашему потоку процессорное время. При этом она передает ему ресурс, автоматически обновляя элементы структуры CRITICAL_SECTION.

Внутреннее устройство *EnterCriticalSection* не слишком сложно; она выполняет лишь несколько простых операций. Чем она действительно ценна, так это способностью выполнять их на уровне атомарного доступа. Даже если два потока на много-процессорной машине одновременно вызовут *EnterCriticalSection*, функция все равно корректно справится со своей задачей: один поток получит ресурс, другой — перейдет в ожидание.

Поток, переведенный *EnterCriticalSection* в ожидание, может надолго лишиться доступа к процессору, а в плохо написанной программе — даже вообще не получить его. Когда именно так и происходит, говорят, что поток «голодает».

WINDOWS 2000 В действительности потоки, ожидающие освобождения критической секции, никогда не блокируются «навечно». *EnterCriticalSection* устроена так, что по истечении определенного времени, генерирует исключение. После этого Вы можете подключить к своей программе отладчик и посмотреть, что в ней случилось. Длительность времени ожидания функцией *EnterCriticalSection* определяется значением параметра *CriticalSectionTimeout*, который хранится в следующем разделе системного реестра:

HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\Session Manager

Длительность времени ожидания измеряется в секундах и по умолчанию равна 2 592 000 секунд (что составляет ровно 30 суток). Не устанавливайте слишком малое значение этого параметра (например, менее 3 секунд), так как иначе Вы нарушите работу других потоков и приложений, которые обычно ждут освобождения критической секции дольше трех секунд.

Вместо *EnterCriticalSection* Вы можете воспользоваться:

```
BOOL TryEnterCriticalSection(PCRITICAL_SECTION pcs);
```

Эта функция никогда не приостанавливает выполнение вызывающего потока. Но возвращаемое ею значение сообщает, получил ли этот поток доступ к ресурсу. Если при ее вызове указанный ресурс занят другим потоком, она возвращает FALSE.

TryEnterCriticalSection позволяет потоку быстро проверить, доступен ли ресурс, и, если нет, заняться чем-нибудь другим. Если функция возвращает TRUE, значит, она обновила элементы структуры CRITICAL_SECTION так, чтобы они сообщали о захвате ресурса вызывающим потоком. Отсюда следует, что для каждого вызова функции *TryEnterCriticalSection*, где она возвращает TRUE, надо предусмотреть парный вызов *LeaveCriticalSection*.

WINDOWS 2000 В Windows 98 функция *TryEnterCriticalSection* определена, но не реализована. При ее вызове всегда возвращается FALSE.

В конце участка кода, использующего разделяемый ресурс, должен присутствовать вызов:

```
VOID LeaveCriticalSection(PCRITICAL_SECTION pcs);
```

Эта функция просматривает элементы структуры CRITICAL_SECTION и уменьшает счетчик числа захватов ресурса вызывающим потоком на 1. Если его значение больше 0, *LeaveCriticalSection* ничего не делает и просто возвращает управление.

Если значение счетчика достигло 0, *LeaveCriticalSection* сначала выясняет, есть ли в системе другие потоки, ждущие данный ресурс в вызове *EnterCriticalSection*. Если есть хотя бы один такой поток, функция настраивает значения элементов структуры, чтобы они сигнализировали о занятости ресурса, и отдает его одному из ждущих потоков (поток выбирается «по справедливости»). Если же ресурс никому не нужен, *LeaveCriticalSection* соответственно сбрасывает элементы структуры.

Как и *EnterCriticalSection*, функция *LeaveCriticalSection* выполняет все действия на уровне атомарного доступа. Однако *LeaveCriticalSection* никогда не приостанавливает поток, а управление возвращается немедленно.

Критические секции и спин-блокировка

Когда поток пытается войти в критическую секцию, занятую другим потоком, он немедленно приостанавливается. А это значит, что поток переходит из пользовательского режима в режим ядра (на что затрачивается около 1000 тактов процессора). Цена такого перехода чрезвычайно высока. На многопроцессорной машине поток, владеющий ресурсом, может выполняться на другом процессоре и очень быстро освободить ресурс. Тогда появляется вероятность, что ресурс будет освобожден еще до того, как вызывающий поток завершит переход в режим ядра. В итоге уйма процессорного времени будет потрачена впустую.

Microsoft повысила быстродействие критических секций, включив в них спин-блокировку. Теперь, когда Вы вызываете *EnterCriticalSection*, она выполняет заданное число циклов спин-блокировки, пытаясь получить доступ к ресурсу. И лишь в том случае, когда все попытки заканчиваются неудачно, функция переводит поток в режим ядра, где он будет находиться в состоянии ожидания.

Для использования спин-блокировки в критической секции нужно инициализировать счетчик циклов, вызывав:

```
BOOL InitializeCriticalSectionAndSpinCount(
    PCRITICAL_SECTION pcs,
    DWORD dwSpinCount);
```

Как и в *InitializeCriticalSection*, первый параметр этой функции — адрес структуры критической секции. Но во втором параметре, *dwSpinCount*, передается число циклов спин-блокировки при попытках получить доступ к ресурсу до перевода потока в со-

стояние ожидания. Этот параметр может принимать значения от 0 до 0x00FFFFFF. Учтите, что на однопроцессорной машине значение параметра *dwSpinCount* игнорируется и считается равным 0. Дело в том, что применение спин-блокировки в такой системе бессмысленно: поток, владеющий ресурсом, не сможет освободить его, пока другой поток «крутится» в циклах спин-блокировки.

Вы можете изменить счетчик циклов спин-блокировки вызовом:

```
DWORD SetCriticalSectionSpinCount(
    PCRITICAL_SECTION pcs,
    DWORD dwSpinCount);
```

И в этой функции значение *dwSpinCount* на однопроцессорной машине игнорируется.

На мой взгляд, используя критические секции, Вы должны всегда применять спин-блокировку — терять Вам просто нечего. Могут возникнуть трудности в подборе значения *dwSpinCount*, но здесь нужно просто поэкспериментировать. Имейте в виду, что для критической секции, стоящей на страже динамической кучи Вашего процесса, этот счетчик равен 4000.

Как реализовать критические секции с применением спин-блокировки, я покажу в главе 10.

Критические секции и обработка ошибок

Вероятность того, что *InitializeCriticalSection* потерпит неудачу, крайне мала, но все же существует. В свое время Microsoft не учла этого при разработке функции и определила ее возвращаемое значение как VOID, т. е. она ничего не возвращает. Однако функция может потерпеть неудачу, так как выделяет блок памяти для внутрисистемной отладочной информации. Если выделить память не удается, генерируется исключение STATUS_NO_MEMORY. Вы можете перехватить его, используя структурную обработку исключений (см. главы 23, 24 и 25).

Есть и другой, более простой способ решить эту проблему — перейти на новую функцию *InitializeCriticalSectionAndSpinCount*. Она, тоже выделяя блок памяти для отладочной информации, возвращает FALSE, если выделить память не удается.

В работе с критическими секциями может возникнуть еще одна проблема. Когда за доступ к критической секции конкурирует два и более потоков, она использует объект ядра «событие». (Я покажу, как работать с этим объектом при описании C++-класса COptex в главе 10.) Поскольку такая конкуренция маловероятна, система не создает объект ядра «событие» до тех пор, пока он действительно не потребуется. Это экономит массу системных ресурсов — в большинстве критических секций конкуренция потоков никогда не возникает.

Но если потоки все же будут конкурировать за критическую секцию в условиях нехватки памяти, система не сможет создать нужный объект ядра. И тогда *EnterCriticalSection* возбудит исключение EXCEPTION_INVALID_HANDLE. Большинство разработчиков просто игнорирует вероятность такой ошибки и не предусматривает для нее никакой обработки, поскольку она случается действительно очень редко. Но если Вы хотите заранее подготовиться к такой ситуации, у Вас есть две возможности.

Первая — использовать структурную обработку исключений и перехватывать ошибку. При этом Вы либо отказываетесь от обращения к ресурсу, защищенному критической секцией, либо дожидаетесь появления свободной памяти, а затем повторяете вызов *EnterCriticalSection*.

Вторая возможность заключается в том, что Вы создаете критическую секцию вызовом *InitializeCriticalSectionAndSpinCount*, передавая параметр *dwSpinCount* с уста-

новленным старшим битом. Тогда функция создает объект «событие» и сопоставляет его с критической секцией. Если создать объект не удается, она возвращает FALSE, и это позволяет корректнее обрабатывать такие ситуации. Но успешно созданный объект ядра «событие» гарантирует Вам, что *EnterCriticalSection* выполнит свою задачу при любых обстоятельствах и никогда не вызовет исключение. (Всегда выделяя память под объекты ядра «событие», Вы неэкономно расходуете системные ресурсы. Поэтому делать так следует лишь в нескольких случаях, а именно: если программа может рухнуть из-за неудачного завершения функции *EnterCriticalSection*, если Вы уверены в конкуренции потоков при обращении к критической секции или если программа будет работать в условиях нехватки памяти.)

Несколько полезных приемов

Используя критические секции, желательно привыкнуть делать одни вещи и избегать других. Вот несколько полезных приемов, которые пригодятся Вам в работе с критическими секциями. (Они применимы и к синхронизации потоков с помощью объектов ядра, о которой я расскажу в следующей главе.)

На каждый разделяемый ресурс используйте отдельную структуру CRITICAL_SECTION

Если в Вашей программе имеется несколько независимых структур данных, создавайте для каждой из них отдельный экземпляр структуры CRITICAL_SECTION. Это лучше, чем защищать все разделяемые ресурсы одной критической секцией. Посмотрите на этот фрагмент кода:

```
int g_nNums[100];           // один разделяемый ресурс
TCHAR g_cChars[100];        // другой разделяемый ресурс
CRITICAL_SECTION g_cs;     // защищает оба ресурса

DWORD WINAPI ThreadFunc(PVOID pvParam) {
    EnterCriticalSection(&g_cs);
    for (int x = 0; x < 100; x++) {
        g_nNums[x] = 0;
        g_cChars[x] = TEXT('X');
    }
    LeaveCriticalSection(&g_cs);
    return(0);
}
```

Здесь создана единственная критическая секция, защищающая оба массива — *g_nNums* и *g_cChars* — в период их инициализации. Но эти массивы совершенно различны. И при выполнении данного цикла ни один из потоков не получит доступ ни к одному массиву. Теперь посмотрим, что будет, если *ThreadFunc* реализовать так:

```
DWORD WINAPI ThreadFunc(PVOID pvParam) {
    EnterCriticalSection(&g_cs);
    for (int x = 0; x < 100; x++)
        g_nNums[x] = 0;
    for (x = 0; x < 100; x++)
        g_cChars[x] = TEXT('X');
    LeaveCriticalSection(&g_cs);
    return(0);
}
```

В этом фрагменте массивы инициализируются по отдельности, и теоретически после инициализации *g_nNums* посторонний поток, которому нужен доступ только к первому массиву, сможет начать исполнение — пока *ThreadFunc* занимается вторым массивом. Увы, это невозможно: обе структуры данных защищены одной критической секцией. Чтобы выйти из затруднения, создадим две критические секции:

```
int g_nNums[100];           // разделяемый ресурс
CRITICAL_SECTION g_csNums; // защищает g_nNums
TCHAR g_cChars[100];        // другой разделяемый ресурс
CRITICAL_SECTION g_csChars; // защищает g_cChars

DWORD WINAPI ThreadFunc(PVOID pvParam) {
    EnterCriticalSection(&g_csNums);
    for (int x = 0; x < 100; x++)
        g_nNums[x] = 0;
    LeaveCriticalSection(&g_csNums);
    EnterCriticalSection(&g_csChars);
    for (x = 0; x < 100; x++)
        g_cChars[x] = TEXT('X');
    LeaveCriticalSection(&g_csChars);
    return(0);
}
```

Теперь другой поток сможет работать с массивом *g_nNums*, как только *ThreadFunc* закончит его инициализацию. Можно сделать и так, чтобы один поток инициализировал массив *g_nNums*, а другой — *g_cChars*.

Одновременный доступ к нескольким ресурсам

Иногда нужен одновременный доступ сразу к двум структурам данных. Тогда *ThreadFunc* следует реализовать так:

```
DWORD WINAPI ThreadFunc(PVOID pvParam) {
    EnterCriticalSection(&g_csNums);
    EnterCriticalSection(&g_csChars);
    // в этом цикле нужен одновременный доступ к обоим ресурсам
    for (int x = 0; x < 100; x++) g_nNums[x] = g_cChars[x];
    LeaveCriticalSection(&g_csChars);
    LeaveCriticalSection(&g_csNums);
    return(0);
}
```

Предположим, доступ к обоим массивам требуется и другому потоку в данном процессе; при этом его функция написана следующим образом:

```
DWORD WINAPI OtherThreadFunc(PVOID pvParam) {
    EnterCriticalSection(&g_csChars);
    EnterCriticalSection(&g_csNums);
    for (int x = 0; x < 100; x++) g_nNums[x] = g_cChars[x];
    LeaveCriticalSection(&g_csNums);
    LeaveCriticalSection(&g_csChars);
    return(0);
}
```

Я лишь поменял порядок вызовов *EnterCriticalSection* и *LeaveCriticalSection*. Но из-за того, что функции *ThreadFunc* и *OtherThreadFunc* написаны именно так, существу-

ет вероятность *взаимной блокировки* (deadlock). Допустим, *ThreadFunc* начинает исполнение и занимает критическую секцию *g_csNums*. Получив от системы процессорное время, поток с функцией *OtherThreadFunc* захватывает критическую секцию *g_csChars*. Тут-то и происходит взаимная блокировка потоков. Какая бы из функций — *ThreadFunc* или *OtherThreadFunc* — ни пыталась продолжить исполнение, она не сумеет занять другую, необходимую ей критическую секцию.

Эту ситуацию легко исправить, написав код обеих функций так, чтобы они вызывали *EnterCriticalSection* в одинаковом порядке. Заметьте, что порядок вызовов *LeaveCriticalSection* несуществен, поскольку эта функция никогда не приостанавливает поток.

Не занимайте критические секции надолго

Надолго занимая критическую секцию, Ваше приложение может блокировать другие потоки, что отрицательно скажется на его общей производительности. Вот прием, позволяющий свести к минимуму время пребывания в критической секции. Следующий код не дает другому потоку изменять значение в *g_s* до тех пор, пока в окно не будет отправлено сообщение WM_SOMEWSMSG:

```
SOMESTRUCT g_s;
CRITICAL_SECTION g_cs;

DWORD WINAPI SomeThread(PVOID pvParam) {
    EnterCriticalSection(&g_cs);
    // посыпаем в окно сообщение
    SendMessage(hwndSomeWnd, WM_SOMEWSMSG, &g_s, 0);
    LeaveCriticalSection(&g_cs);
    return(0);
}
```

Трудно сказать, сколько времени уйдет на обработку WM_SOMEWSMSG оконной процедурой — может, несколько миллисекунд, а может, и несколько лет. В течение этого времени никакой другой поток не получит доступ к структуре *g_s*. Поэтому лучше составить код иначе:

```
SOMESTRUCT g_s;
CRITICAL_SECTION g_cs;

DWORD WINAPI SomeThread(PVOID pvParam) {
    EnterCriticalSection(&g_cs);
    SOMESTRUCT sTemp = g_s;
    LeaveCriticalSection(&g_cs);
    // посыпаем в окно сообщение
    SendMessage(hwndSomeWnd, WM_SOMEWSMSG, &sTemp, 0);
    return(0);
}
```

Этот код сохраняет значение элемента *g_s* во временной переменной *sTemp*. Нетрудно догадаться, что на исполнение этой строки уходит всего несколько тактов процессора. Далее программа сразу вызывает *LeaveCriticalSection* — защищать глобальную структуру больше не нужно. Так что вторая версия программы намного лучше первой, поскольку другие потоки «отлучаются» от структуры *g_s* лишь на несколько тактов процессора, а не на неопределенно долгое время. Такой подход предполагает, что «моментальный снимок» структуры вполне пригоден для чтения оконной процедуры, а также что оконная процедура не будет изменять элементы этой структуры.

Синхронизация потоков с использованием объектов ядра

В предыдущей главе мы обсудили, как синхронизировать потоки с применением механизмов, позволяющих Вашим потокам оставаться в пользовательском режиме. Самое удивительное, что эти механизмы работают очень быстро. Поэтому, если Вы озабочены быстродействием потока, сначала проверьте, нельзя ли обойтись синхронизацией в пользовательском режиме.

Хотя механизмы синхронизации в пользовательском режиме обеспечивают высокое быстродействие, им свойствен ряд ограничений, и во многих приложениях они просто не будут работать. Например, *Interlocked*-функции оперируют только с отдельными переменными и никогда не переводят поток в состояние ожидания. Последнюю задачу можно решить с помощью критических секций, но они подходят лишь в тех случаях, когда требуется синхронизировать потоки в рамках одного процесса. Кроме того, при использовании критических секций легко попасть в ситуацию взаимной блокировки потоков, потому что задать предельное время ожидания входа в критическую секцию нельзя.

В этой главе мы рассмотрим, как синхронизировать потоки с помощью объектов ядра. Вы увидите, что такие объекты предоставляют куда больше возможностей, чем механизмы синхронизации в пользовательском режиме. В сущности, единственный их недостаток — меньшее быстродействие. Дело в том, что при вызове любой из функций, упоминаемых в этой главе, поток должен перейти из пользовательского режима в режим ядра. А такой переход обходится очень дорого — в 1000 процессорных тактов на платформе x86. Прибавьте сюда еще и время, которое необходимо на выполнение кода этих функций в режиме ядра.

К этому моменту я уже рассказал Вам о нескольких объектах ядра, в том числе о процессах, потоках и заданиях. Почти все они годятся и для решения задач синхронизации. В случае синхронизации потоков о каждом из этих объектов говорят, что он находится либо в свободном (*signaled state*), либо в занятом состоянии (*nonsignaled state*). Переход из одного состояния в другое осуществляется по правилам, определенным Microsoft для каждого из объектов ядра. Так, объекты ядра «процесс» сразу после создания всегда находятся в занятом состоянии. В момент завершения процесса операционная система автоматически освобождает его объект ядра «процесс», и он на всегда остается в этом состоянии.

Объект ядра «процесс» пребывает в занятом состоянии, пока выполняется сопоставленный с ним процесс, и переходит в свободное состояние, когда процесс завершается. Внутри этого объекта поддерживается булева переменная, которая при создании объекта инициализируется как FALSE («занято»). По окончании работы процесса операционная система меняет значение этой переменной на TRUE, сообщая тем самым, что объект свободен.

Если Вы пишете код, проверяющий, выполняется ли процесс в данный момент, Вам нужно лишь вызвать функцию, которая просит операционную систему проверить значение булевой переменной, принадлежащей объекту ядра «процесс». Тут нет ничего сложного. Вы можете также сообщить системе, чтобы та перевела Ваш поток в состояние ожидания и автоматически пробудила его при изменении значения булевой переменной с FALSE на TRUE. Тогда появляется возможность заставить поток в родительском процессе, ожидающий завершения дочернего процесса, просто заснуть до освобождения объекта ядра, идентифицирующего дочерний процесс. В дальнейшем Вы увидите, что в Windows есть ряд функций, позволяющих легко решать эту задачу.

Я только что описал правила, определенные Microsoft для объекта ядра «процесс». Точно такие же правила распространяются и на объекты ядра «поток». Они тоже сразу после создания находятся в занятом состоянии. Когда поток завершается, операционная система автоматически переводит объект ядра «поток» в свободное состояние. Таким образом, используя те же приемы, Вы можете определить, выполняется ли в данный момент тот или иной поток. Как и объект ядра «процесс», объект ядра «поток» никогда не возвращается в занятое состояние.

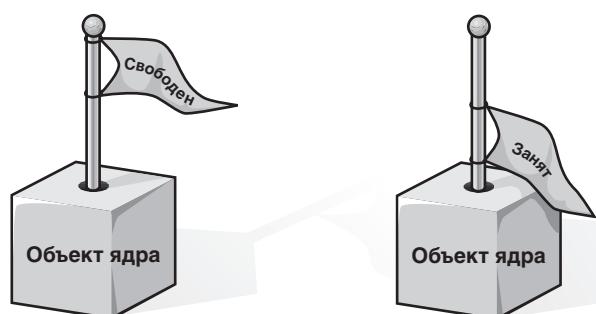
Следующие объекты ядра бывают в свободном или занятом состоянии:

- | | |
|-------------------|-----------------------------------|
| ■ процессы | ■ уведомления об изменении файлов |
| ■ потоки | ■ события |
| ■ задания | ■ ожидаемые таймеры |
| ■ файлы | ■ семафоры |
| ■ консольный ввод | ■ мьютексы |

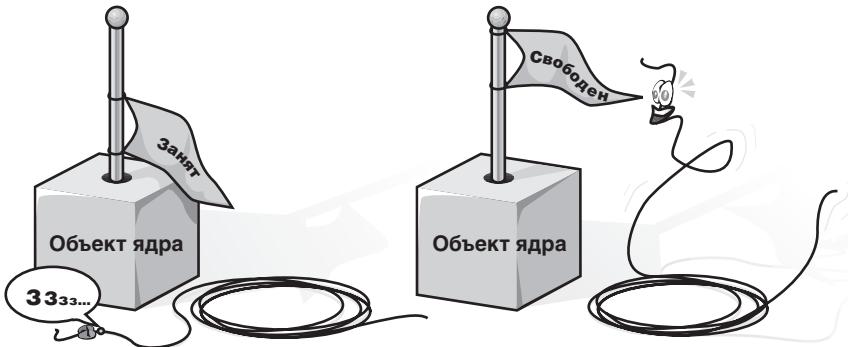
Потоки могут засыпать и в таком состоянии ждать освобождения какого-либо объекта. Правила, по которым объект переходит в свободное или занятое состояние, зависят от типа этого объекта. О правилах для объектов процессов и потоков я упоминал совсем недавно, а правила для заданий были описаны в главе 5.

В этой главе мы обсудим функции, которые позволяют потоку ждать перехода определенного объекта ядра в свободное состояние. Потом мы поговорим об объектах ядра, предоставляемых Windows специально для синхронизации потоков: событиях, ожидаемых таймерах, семафорах и мьютексах.

Когда я только начинал осваивать всю эту тематику, я предпочитал рассматривать понятия «свободен-занят» по аналогии с обычным флагом. Когда объект свободен, флагок поднят, а когда он занят, флагок опущен.



Потоки спят, пока ожидаемые ими объекты заняты (флагок опущен). Как только объект освободился (флагок поднят), спящий поток замечает это, просыпается и возобновляет выполнение.



Wait-функции

Wait-функции позволяют потоку в любой момент приостановиться и ждать освобождения какого-либо объекта ядра. Из всего семейства этих функций чаще всего используется *WaitForSingleObject*:

```
DWORD WaitForSingleObject(
    HANDLE hObject,
    DWORD dwMilliseconds);
```

Когда поток вызывает эту функцию, первый параметр, *hObject*, идентифицирует объект ядра, поддерживающий состояния «свободен-занят». (То есть любой объект, упомянутый в списке из предыдущего раздела.) Второй параметр, *dwMilliseconds*, указывает, сколько времени (в миллисекундах) поток готов ждать освобождения объекта.

Следующий вызов сообщает системе, что поток будет ждать до тех пор, пока не завершится процесс, идентифицируемый описателем *hProcess*:

```
WaitForSingleObject(hProcess, INFINITE);
```

В данном случае константа *INFINITE*, передаваемая во втором параметре, подсказывает системе, что вызывающий поток готов ждать этого события хоть целую вечность. Именно эта константа обычно и передается функции *WaitForSingleObject*, но Вы можете указать любое значение в миллисекундах. Кстати, константа *INFINITE* определена как *0xFFFFFFFF* (или *-1*). Разумеется, передача *INFINITE* не всегда безопасна. Если объект так и не перейдет в свободное состояние, вызывающий поток никогда не проснется; одно утешение: тратить драгоценное процессорное время он при этом не будет.

Вот пример, иллюстрирующий, как вызывать *WaitForSingleObject* со значением таймаута, отличным от *INFINITE*:

```
DWORD dw = WaitForSingleObject(hProcess, 5000);
switch (dw) {

    case WAIT_OBJECT_0:
        // процесс завершается
        break;

    case WAIT_TIMEOUT:
        // процесс не завершился в течение 5000 мс
        break;
}
```

см. след. стр.

```
case WAIT_FAILED:  
    // неправильный вызов функции (неверный описатель?)  
    break;  
}
```

Данный код сообщает системе, чтозывающий поток не должен получать процессорное время, пока не завершится указанный процесс или не пройдет 5000 мс (в зависимости от того, что случится раньше). Поэтому функция вернет управление либо до истечения 5000 мс, если процесс завершится, либо примерно через 5000 мс, если процесс к тому времени не закончит свою работу. Заметьте, что в параметре *dwMilliseconds* можно передать 0, и тогда *WaitForSingleObject* немедленно вернет управление.

Возвращаемое значение функции *WaitForSingleObject* указывает, почемузывающий поток снова стал планируемым. Если функция возвращает WAIT_OBJECT_0, объект свободен, а если WAIT_TIMEOUT — заданное время ожидания (таймаут) истекло. При передаче неверного параметра (например, недопустимого описателя) *WaitForSingleObject* возвращает WAIT_FAILED. Чтобы выяснить конкретную причину ошибки, вызовите функцию *GetLastError*.

Функция *WaitForMultipleObjects* аналогична *WaitForSingleObject* с тем исключением, что позволяет ждать освобождения сразу нескольких объектов или какого-то одного из списка объектов:

```
DWORD WaitForMultipleObjects(  
    DWORD dwCount,  
    CONST HANDLE* phObjects,  
    BOOL fWaitAll,  
    DWORD dwMilliseconds);
```

Параметр *dwCount* определяет количество интересующих Вас объектов ядра. Его значение должно быть в пределах от 1 до MAXIMUM_WAIT_OBJECTS (в заголовочных файлах Windows оно определено как 64). Параметр *phObjects* — это указатель на массив описателей объектов ядра.

WaitForMultipleObjects приостанавливает поток и заставляет его ждать освобождения либо всех заданных объектов ядра, либо одного из них. Параметр *fWaitAll* как раз и определяет, чего именно Вы хотите от функции. Если он равен TRUE, функция не даст потоку возобновить свою работу, пока не освободятся все объекты.

Параметр *dwMilliseconds* идентичен одноименному параметру функции *WaitForSingleObject*. Если Вы указываете конкретное время ожидания, то по его истечении функция в любом случае возвращает управление. И опять же, в этом параметре обычно передают INFINITE (будьте внимательны при написании кода, чтобы не создать ситуацию взаимной блокировки).

Возвращаемое значение функции *WaitForMultipleObjects* сообщает, почему возобновилось выполнение вызвавшего ее потока. Значения WAIT_FAILED и WAIT_TIMEOUT никаких пояснений не требуют. Если Вы передали TRUE в параметре *fWaitAll* и все объекты перешли в свободное состояние, функция возвращает значение WAIT_OBJECT_0. Если же *fWaitAll* приравнен FALSE, она возвращает управление, как только освобождается любой из объектов. Вы, по-видимому, захотите выяснить, какой именно объект освободился. В этом случае возвращается значение от WAIT_OBJECT_0 до WAIT_OBJECT_0 + *dwCount* - 1. Иначе говоря, если возвращаемое значение не равно WAIT_TIMEOUT или WAIT_FAILED, вычтите из него значение WAIT_OBJECT_0, и Вы получите индекс в массиве описателей, на который указывает второй параметр функции *WaitForMultipleObjects*. Индекс подскажет Вам, какой объект перешел в незанятое состояние. Поясню сказанное на примере.

```

HANDLE h[3];
h[0] = hProcess1;
h[1] = hProcess2;
h[2] = hProcess3;
DWORD dw = WaitForMultipleObjects(3, h, FALSE, 5000);
switch (dw) {
    case WAIT_FAILED:
        // неправильный вызов функции (неверный описатель?)
        break;

    case WAIT_TIMEOUT:
        // ни один из объектов не освободился в течение 5000 мс
        break;

    case WAIT_OBJECT_0 + 0:
        // завершился процесс, идентифицируемый h[0], т. е. описателем (hProcess1)
        break;

    case WAIT_OBJECT_0 + 1:
        // завершился процесс, идентифицируемый h[1], т. е. описателем (hProcess2)
        break;

    case WAIT_OBJECT_0 + 2:
        // завершился процесс, идентифицируемый h[2], т. е. описателем (hProcess3)
        break;
}

```

Если Вы передаете FALSE в параметре *fWaitAll*, функция *WaitForMultipleObjects* сканирует массив описателей (начиная с нулевого элемента), и первый же освободившийся объект прерывает ожидание. Это может привести к нежелательным последствиям. Например, Ваш поток ждет завершения трех дочерних процессов; при этом Вы передали функции массив с их описателями. Если завершается процесс, описатель которого находится в нулевом элементе массива, *WaitForMultipleObjects* возвращает управление. Теперь поток может сделать то, что ему нужно, и вновь вызвать эту функцию, ожидая завершения другого процесса. Если поток передаст те же три описателя, функция немедленно вернет управление, и Вы снова получите значение WAIT_OBJECT_0. Таким образом, пока Вы не удалите описатели тех объектов, об освобождении которых функция уже сообщила Вам, код будет работать некорректно.

Побочные эффекты успешного ожидания

Успешный вызов *WaitForSingleObject* или *WaitForMultipleObjects* на самом деле меняет состояние некоторых объектов ядра. Под успешным вызовом я имею в виду тот, при котором функция видит, что объект освободился, и возвращает значение, относительное WAIT_OBJECT_0. Вызов считается неудачным, если возвращается WAIT_TIMEOUT или WAIT_FAILED. В последнем случае состояние каких-либо объектов не меняется.

Изменение состояния объекта в результате вызова я называю *побочным эффектом успешного ожидания* (successful wait side effect). Например, поток ждет объект «событие с автосбросом» (auto-reset event object) (об этих объектах я расскажу чуть позже). Когда объект переходит в свободное состояние, функция обнаруживает это и может вернуть вызывающему потоку значение WAIT_OBJECT_0. Однако перед самым возвратом из функции событие переводится в занятое состояние — здесь оказывается побочный эффект успешного ожидания.

Объекты ядра «событие с автосбросом» ведут себя подобным образом, потому что таково одно из правил, определенных Microsoft для объектов этого типа. Другие объекты дают иные побочные эффекты, а некоторые — вообще никаких. К последним относятся объекты ядра «процесс» и «поток», так что поток, ожидающий один из этих объектов, никогда не изменит его состояние. Подробнее о том, как ведут себя объекты ядра, я буду рассказывать при рассмотрении соответствующих объектов.

Чем ценна функция *WaitForMultipleObjects*, так это тем, что она выполняет все действия на уровне атомарного доступа. Когда поток обращается к этой функции, она ждет освобождения всех объектов и в случае успеха вызывает в них требуемые побочные эффекты; причем все действия выполняются как одна операция.

Возьмем такой пример. Два потока вызывают *WaitForMultipleObjects* совершенно одинаково:

```
HANDLE h[2];
h[0] = hAutoResetEvent1; // изначально занят
h[1] = hAutoResetEvent2; // изначально занят
WaitForMultipleObjects(2, h, TRUE, INFINITE);
```

На момент вызова *WaitForMultipleObjects* эти объекты-события заняты, и оба потока переходят в режим ожидания. Но вот освобождается объект *hAutoResetEvent1*. Это становится известным обоим потокам, однако ни один из них не пробуждается, так как объект *hAutoResetEvent2* по-прежнему занят. Поскольку потоки все еще ждут, никакого побочного эффекта для объекта *hAutoResetEvent1* не возникает.

Наконец освобождается и объект *hAutoResetEvent2*. В этот момент один из потоков обнаруживает, что освободились оба объекта, которых он ждал. Его ожидание успешно завершается, оба объекта снова переводятся в занятое состояние, и выполнение потока возобновляется. А что же происходит со вторым потоком? Он продолжает ждать и будет делать это, пока вновь не освободятся оба объекта-события.

Как я уже упоминал, *WaitForMultipleObjects* работает на уровне атомарного доступа, и это очень важно. Когда она проверяет состояние объектов ядра, никто не может «у нее за спиной» изменить состояние одного из этих объектов. Благодаря этому исключаются ситуации со взаимной блокировкой. Только представьте, что получится, если один из потоков, обнаружив освобождение *hAutoResetEvent1*, сбросит его в занятое состояние, а другой поток, узнав об освобождении *hAutoResetEvent2*, тоже переведет его в занятое состояние. Оба потока просто зависнут: первый будет ждать освобождения объекта, захваченного вторым потоком, а второй — освобождения объекта, захваченного первым. *WaitForMultipleObjects* гарантирует, что такого не случится никогда.

Тут возникает интересный вопрос. Если несколько потоков ждет один объект ядра, какой из них пробудится при освобождении этого объекта? Официально Microsoft отвечает на этот вопрос так: «Алгоритм действует честно.» Что это за алгоритм, Microsoft не говорит, потому что не хочет связывать себя обязательствами всегда придерживаться именно этого алгоритма. Она утверждает лишь одно: если объект ожидается несколькими потоками, то всякий раз, когда этот объект переходит в свободное состояние, каждый из них получает шанс на пробуждение.

Таким образом, приоритет потока не имеет значения: поток с самым высоким приоритетом не обязательно первым захватит объект. Не получает преимущества и поток, который ждал дольше всех. Есть даже вероятность, что какой-то поток сумеет повторно захватить объект. Конечно, это было бы нечестно по отношению к другим потокам, и алгоритм пытается не допустить этого. Но никаких гарантий нет.

На самом деле этот алгоритм просто использует популярную схему «первым вошел — первым вышел» (FIFO). В принципе, объект захватывается потоком, ждавшим дольше всех. Но в системе могут произойти какие-то события, которые повлияют на окончательное решение, и из-за этого алгоритм становится менее предсказуемым. Вот почему Microsoft и не хочет говорить, как именно он работает. Одно из таких событий — приостановка какого-либо потока. Если поток ждет объект и вдруг приостанавливается, система просто забывает, что он ждал этот объект. А причина в том, что нет смысла планировать приостановленный поток. Когда он в конце концов возобновляется, система считает, что он только что начал ждать данный объект.

Учитывайте это при отладке, поскольку в точках прерывания (breakpoints) все потоки внутри отлаживаемого процесса приостанавливаются. Отладка делает алгоритм FIFO в высшей степени непредсказуемым из-за частых приостановки и возобновления потоков процесса.

События

События — самая примитивная разновидность объектов ядра. Они содержат счетчик числа пользователей (как и все объекты ядра) и две булевые переменные: одна сообщает тип данного объекта-события, другая — его состояние (свободен или занят).

События просто уведомляют об окончании какой-либо операции. Объекты-события бывают двух типов: со сбросом вручную (manual-reset events) и с автосбросом (auto-reset events). Первые позволяют возобновлять выполнение сразу нескольких ждущих потоков, вторые — только одного.

Объекты-события обычно используют в том случае, когда какой-то поток выполняет инициализацию, а затем сигнализирует другому потоку, что тот может продолжить работу. Инициализирующий поток переводит объект «событие» в занятое состояние и приступает к своим операциям. Закончив, он сбрасывает событие в свободное состояние. Тогда другой поток, который ждал перехода события в свободное состояние, пробуждается и вновь становится планируемым.

Объект ядра «событие» создается функцией *CreateEvent*:

```
HANDLE CreateEvent(
    PSECURITY_ATTRIBUTES psa,
    BOOL fManualReset,
    BOOL fInitialState,
    PCTSTR pszName);
```

В главе 3 мы обсуждали общие концепции, связанные с объектами ядра, — защиту, учет числа пользователей объектов, наследование их описателей и совместное использование объектов за счет присвоения им одинаковых имен. Поскольку все это Вы теперь знаете, я не буду рассматривать первый и последний параметры данной функции.

Параметр *fManualReset* (булева переменная) сообщает системе, хотите Вы создать событие со сбросом вручную (TRUE) или с автосбросом (FALSE). Параметр *fInitialState* определяет начальное состояние события — свободное (TRUE) или занятое (FALSE). После того как система создает объект-событие, *CreateEvent* возвращает описатель события, специфичный для конкретного процесса. Потоки из других процессов могут получить доступ к этому объекту: 1) вызовом *CreateEvent* с тем же параметром *pszName*; 2) наследованием описателя; 3) применением функции *DuplicateHandle*; и 4) вызовом *OpenEvent* с передачей в параметре *pszName* имени, совпадающего с указанным в аналогичном параметре функции *CreateEvent*. Вот что представляет собой функция *OpenEvent*.

```
HANDLE OpenEvent(  
    DWORD fdwAccess,  
    BOOL fInherit,  
    PCTSTR pszName);
```

Ненужный объект ядра «событие» следует, как всегда, закрыть вызовом *CloseHandle*. Создав событие, Вы можете напрямую управлять его состоянием. Чтобы перевесить его в свободное состояние, Вы вызываете:

```
BOOL SetEvent(HANDLE hEvent);
```

А чтобы поменять его на занятое:

```
BOOL ResetEvent(HANDLE hEvent);
```

Вот так все просто.

Для событий с автосбросом действует следующее правило. Когда его ожидание потоком успешно завершается, этот объект автоматически сбрасывается в занятое состояние. Отсюда и произошло название таких объектов-событий. Для этого объекта обычно не требуется вызывать *ResetEvent*, поскольку система сама восстанавливает его состояние. А для событий со сбросом вручную никаких побочных эффектов успешного ожидания не предусмотрено.

Рассмотрим небольшой пример тому, как на практике использовать объекты ядра «событие» для синхронизации потоков. Начнем с такого кода:

```
// глобальный описатель события со сбросом вручную (в занятом состоянии)  
HANDLE g_hEvent;  
  
int WINAPI WinMain(...) {  
  
    // создаем объект "событие со сбросом вручную" (в занятом состоянии)  
    g_hEvent = CreateEvent(NULL, TRUE, FALSE, NULL);  
  
    // порождаем три новых потока  
    HANDLE hThread[3];  
    DWORD dwThreadID;  
    hThread[0] = _beginthreadex(NULL, 0, WordCount, NULL, 0, &dwThreadID);  
    hThread[1] = _beginthreadex(NULL, 0, SpellCheck, NULL, 0, &dwThreadID);  
    hThread[2] = _beginthreadex(NULL, 0, GrammarCheck, NULL, 0, &dwThreadID);  
  
    OpenFileAndReadContentsIntoMemory(...);  
    // разрешаем всем трем потокам обращаться к памяти  
    SetEvent(g_hEvent);  
    :  
}  
  
DWORD WINAPI WordCount(PVOID pvParam) {  
    // ждем, когда в память будут загружены данные из файла  
    WaitForSingleObject(g_hEvent, INFINITE);  
    // обращаемся к блоку памяти  
    :  
    return(0);  
}  
  
DWORD WINAPI SpellCheck(PVOID pvParam) {
```

```

// ждем, когда в память будут загружены данные из файла
WaitForSingleObject(g_hEvent, INFINITE);

// обращаемся к блоку памяти
:
return(0);
}

DWORD WINAPI GrammarCheck(PVOID pvParam) {
    // ждем, когда в память будут загружены данные из файла
    WaitForSingleObject(g_hEvent, INFINITE);

    // обращаемся к блоку памяти
    :
    return(0);
}

```

При запуске этот процесс создает занятое событие со сбросом вручную и записывает его описатель в глобальную переменную. Это упрощает другим потокам процесса доступ к тому же объекту-событию. Затем порождаются три потока. Они ждут, когда в память будут загружены данные (текст) из некоего файла, и потом обращаются к этим данным: один поток подсчитывает количество слов, другой проверяет орфографические ошибки, третий — грамматические. Все три функции потоков начинают работать одинаково: каждый поток вызывает *WaitForSingleObject*, которая приостанавливает его до тех пор, пока первичный поток не считает в память содержимое файла.

Загрузив нужные данные, первичный поток вызывает *SetEvent*, которая переводит событие в свободное состояние. В этот момент система пробуждает три вторичных потока, и они, вновь получив процессорное время, обращаются к блоку памяти. Заметьте, что они получают доступ к памяти в режиме только для чтения. Это единственная причина, по которой все три потока могут выполняться одновременно.

Если событие со сбросом вручную заменить на событие с автосбросом, программа будет вести себя совершенно иначе. После вызова первичным потоком функции *SetEvent* система возобновит выполнение только одного из вторичных потоков. Кого именно — сказать заранее нельзя. Остальные два потока продолжат ждать.

Поток, вновь ставший планируемым, получает монопольный доступ к блоку памяти, где хранятся данные, считанные из файла. Давайте перепишем функции потоков так, чтобы перед самым возвратом управления они (подобно функции *WinMain*) вызывали *SetEvent*. Теперь функции потоков выглядят следующим образом:

```

DWORD WINAPI WordCount(PVOID pvParam) {
    // ждем, когда в память будут загружены данные из файла
    WaitForSingleObject(g_hEvent, INFINITE);

    // обращаемся к блоку памяти
    :
    SetEvent(g_hEvent);
    return(0);
}

DWORD WINAPI SpellCheck(PVOID pvParam) {
    // ждем, когда в память будут загружены данные из файла

```

см. след. стр.

```
WaitForSingleObject(g_hEvent, INFINITE);

// обращаемся к блоку памяти
⋮
SetEvent(g_hEvent);
return(0);
}

DWORD WINAPI GrammarCheck(PVOID pvParam) {
    // ждем, когда в память будут загружены данные из файла
    WaitForSingleObject(g_hEvent, INFINITE);

    // обращаемся к блоку памяти
    ⋮
    SetEvent(g_hEvent);
    return(0);
}
```

Закончив свою работу с данными, поток вызывает *SetEvent*, которая разрешает системе возобновить выполнение следующего из двух ждущих потоков. И опять мы не знаем, какой поток выберет система, но так или иначе кто-то из них получит монопольный доступ к тому же блоку памяти. Когда и этот поток закончит свою работу, он тоже вызовет *SetEvent*, после чего с блоком памяти сможет монопольно оперировать третий, последний поток. Обратите внимание, что использование события с автосбросом снимает проблему с доступом вторичных потоков к памяти как для чтения, так и для записи; Вам больше не нужно ограничивать их доступ только чтением. Этот пример четко иллюстрирует различия в применении событий со сбросом вручную и с автосбросом.

Для полноты картины упомяну о еще одной функции, которую можно использовать с объектами-событиями:

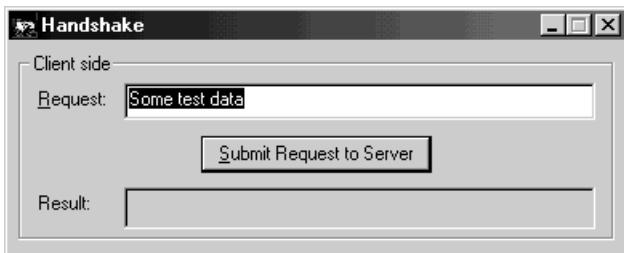
```
BOOL PulseEvent(HANDLE hEvent);
```

PulseEvent освобождает событие и тут же переводит его обратно в занятое состояние; ее вызов равнозначен последовательному вызову *SetEvent* и *ResetEvent*. Если Вы вызываете *PulseEvent* для события со сбросом вручную, любые потоки, ждущие этот объект, становятся планируемыми. При вызове этой функции применительно к событию с автосбросом пробуждается только один из ждущих потоков. А если ни один из потоков не ждет объект-событие, вызов функции не дает никакого эффекта.

Особой пользы от *PulseEvent* я не вижу. В сущности, я никогда не пользовался ею на практике, потому что абсолютно неясно, какой из потоков заметит этот импульс и станет планируемым. Наверное, в каких-то сценариях *PulseEvent* может пригодиться, но ничего такого мне в голову не приходит. Когда мы перейдем к рассмотрению функции *SignalObjectAndWait*, я расскажу о *PulseEvent* чуть подробнее.

Программа-пример Handshake

Эта программа, «09 Handshake.exe» (см. листинг на рис. 9-1), демонстрирует применение событий с автосбросом. Файлы исходного кода и ресурсов этой программы находятся в каталоге 09-Handshake на компакт-диске, прилагаемом к книге. После запуска Handshake открывается окно, показанное ниже.



Handshake принимает строку запроса, меняет в ней порядок всех символов и показывает результат в поле Result. Самое интересное в программе Handshake — то, как она выполняет эту героическую задачу.

Программа решает типичную проблему программирования. У Вас есть клиент и сервер, которые должны как-то общаться друг с другом. Изначально серверу делать нечего, и он переходит в состояние ожидания. Когда клиент готов передать ему запрос, он помещает этот запрос в разделяемый блок памяти и переводит объект-событие в свободное состояние, чтобы поток сервера считал этот блок памяти и обработал клиентский запрос. Пока серверный поток занят обработкой запроса, клиентский должен ждать, когда будет готов результат. Поэтому клиент переходит в состояние ожидания и остается в нем до тех пор, пока сервер не освободит другой объект-событие, указав тем самым, что результат готов. Вновь пробудившись, клиент узнает, что результат находится в разделяемом блоке памяти, и выводит готовые данные пользователю.

При запуске программа немедленно создает два объекта-события с автосбросом в занятом состоянии. Один из них, *g_hevtRequestSubmitted*, используется как индикатор готовности запроса к серверу. Это событие ожидается серверным потоком и освобождается клиентским. Второй объект-событие, *g_hevtResultReturned*, служит индикатором готовности данных для клиента. Это событие ожидается клиентским потоком, а освобождается серверным.

После создания событий программа порождает серверный поток и выполняет функцию *ServerThread*. Эта функция немедленно заставляет серверный поток ждать запроса от клиента. Тем временем первичный поток, который одновременно является и клиентским, вызывает функцию *DialogBox*, отвечающую за отображение пользовательского интерфейса программы. Вы вводите какой-нибудь текст в поле Request и, щелкнув кнопку Submit Request To Server, заставляете программу поместить строку запроса в буфер памяти, разделяемый между клиентским и серверным потоками, а также перевести событие *g_hevtRequestSubmitted* в свободное состояние. Далее клиентский поток ждет результат от сервера, используя объект-событие *g_hevtResultReturned*.

Теперь пробуждается серверный поток, обращает строку в блоке разделяемой памяти, освобождает событие *g_hevtResultReturned* и вновь засыпает, ожидая очередного запроса от клиента. Заметьте, что программа никогда не вызывает *ResetEvent*, так как в этом нет необходимости: события с автосбросом автоматически восстанавливают свое исходное (занятое) состояние в результате успешного ожидания. Клиентский поток обнаруживает, что событие *g_hevtResultReturned* освободилось, пробуждается и копирует строку из общего буфера памяти в поле Result.

Последнее, что заслуживает внимания в этой программе, — то, как она завершается. Вы закрываете ее окно, и это приводит к тому, что *DialogBox* в функции *_tWinMain* возвращает управление. Тогда первичный поток копирует в общий буфер специальную строку и пробуждает серверный поток, чтобы тот ее обработал. Далее первичный поток ждет от сервера подтверждения о приеме этого специального запроса и

завершения его потока. Серверный поток, получив от клиента специальный запрос, выходит из своего цикла и сразу же завершается.

Я предпочел сделать так, чтобы первичный поток ждал завершения серверного вызовом *WaitForMultipleObjects*, — просто из желания продемонстрировать, как используется эта функция. На самом деле я мог бы вызвать и *WaitForSingleObject*, передав ей описатель серверного потока, и все работало бы точно так же.

Как только первичный поток узнает о завершении серверного, он трижды вызывает *CloseHandle* для корректного закрытия всех объектов ядра, которые использовались программой. Конечно, система могла бы закрыть их за меня, но как-то спокойнее, когда делаешь это сам. Я предпочитаю полностью контролировать все, что происходит в моих программах.



Handshake.cpp

```
/*
 * Модуль: Handshake.cpp
 * Автор: Copyright (c) 2000, Джейфри Рихтер (Jeffrey Richter)
 */

#include "..\CmnHdr.h"      /* см. приложение A */
#include <windowsx.h>
#include <tchar.h>
#include <process.h>        // для доступа к beginthreadex
#include "Resource.h"

// это событие освобождается, когда клиент передает запрос серверу
HANDLE g_hevtRequestSubmitted;

// это событие освобождается, когда сервер готов сообщить результат клиенту
HANDLE g_hevtResultReturned;

// это буфер, разделяемый между клиентским и серверным потоками
TCHAR g_szSharedRequestAndResultBuffer[1024];

// специальное значение, посыпаемое клиентом;
// оно заставляет серверный поток корректно завершиться
TCHAR g_szServerShutdown[] = TEXT("Server Shutdown");

// это код, выполняемый серверным потоком
DWORD WINAPI ServerThread(PVOID pvParam) {

    // предполагаем, что серверный поток будет выполняться вечно
    BOOL fShutdown = FALSE;

    while (!fShutdown) {
```

Рис. 9-1. Программа-пример *Handshake*

Рис. 9-1. продолжение

```

// ждем от клиента передачи запроса
WaitForSingleObject(g_hevtRequestSubmitted, INFINITE);

// проверяем, не хочет ли клиент, чтобы сервер завершился
fShutdown =
    (lstrcmpi(g_szSharedRequestAndResultBuffer, g_szServerShutdown) == 0);

if (!fShutdown) {
    // обрабатываем клиентский запрос (инвертируем строку)
    _tcsrev(g_szSharedRequestAndResultBuffer);
}

// разрешаем клиенту обработать результат запроса
SetEvent(g_hevtResultReturned);
}

// клиент хочет нас завершить - выходим
return(0);
}

///////////////////////////////
BOOL Dlg_OnInitDialog(HWND hwnd, HWND hwndFocus, LPARAM lParam) {

chSETDLGICONS(hwnd, IDI_HANDSHAKE);

// инициализируем поле ввода текстом запроса по умолчанию
Edit_SetText(GetDlgItem(hwnd, IDC_REQUEST), TEXT("Some test data"));

return(TRUE);
}

/////////////////////////////
void Dlg_OnCommand(HWND hwnd, int id, HWND hwndCtl, UINT codeNotify) {

switch (id) {

case IDCANCEL:
    EndDialog(hwnd, id);
    break;

case IDC_SUBMIT: // передаем запрос серверному потоку

    // копируем строку запроса в разделяемый блок памяти
    Edit_GetText(GetDlgItem(hwnd, IDC_REQUEST),
        g_szSharedRequestAndResultBuffer,
        chDIMOF(g_szSharedRequestAndResultBuffer));

    // даем знать серверному потоку, что в буфере появился запрос
    SetEvent(g_hevtRequestSubmitted);
}
}

```

см. след. стр.

Рис. 9-1. продолжение

```
// ждем, когда сервер обработает запрос и сообщит нам результат
WaitForSingleObject(g_hevtResultReturned, INFINITE);

// показываем результат пользователю
Edit_SetText(GetDlgItem(hwnd, IDC_RESULT),
    g_szSharedRequestAndResultBuffer);

break;
}

////////// INT_PTR WINAPI Dlg_Proc(HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam) {

switch (uMsg) {
    chHANDLE_DLGMMSG(hwnd, WM_INITDIALOG, Dlg_OnInitDialog);
    chHANDLE_DLGMMSG(hwnd, WM_COMMAND, Dlg_OnCommand);
}

return(FALSE);
}

////////// int WINAPI _tWinMain(HINSTANCE hinstExe, HINSTANCE, PTSTR pszCmdLine, int) {

// создаем и инициализируем два события с автосбросом в занятом состоянии
g_hevtRequestSubmitted = CreateEvent(NULL, FALSE, FALSE, NULL);
g_hevtResultReturned = CreateEvent(NULL, FALSE, FALSE, NULL);

// порождаем серверный поток
DWORD dwThreadID;
HANDLE hThreadServer = chBEGINTHREADEX(NULL, 0, ServerThread, NULL,
    0, &dwThreadID);

// отображаем пользовательский интерфейс клиентского потока
DialogBox(hinstExe, MAKEINTRESOURCE(IDD_HANDSHAKE), NULL, Dlg_Proc);

// пользовательский интерфейс клиента закрывается – надо завершить серверный поток
lstrcpy(g_szSharedRequestAndResultBuffer, g_szServerShutdown);
SetEvent(g_hevtRequestSubmitted);

// ждем от серверного потока подтверждения и его завершения
HANDLE h[2];
h[0] = g_hevtResultReturned;
h[1] = hThreadServer;
WaitForMultipleObjects(2, h, TRUE, INFINITE);

// проводим должную очистку
CloseHandle(hThreadServer);
```

Рис. 9-1. продолжение

```

CloseHandle(g_hevtRequestSubmitted);
CloseHandle(g_hevtResultReturned);

// клиентский поток завершается вместе со всем процессом
return(0);
}

/////////////////////////////// Конец файла ///////////////////////////////

```

Ожидаемые таймеры

Ожидаемые таймеры (waitable timers) — это объекты ядра, которые самостоятельно переходят в свободное состояние в определенное время или через регулярные промежутки времени. Чтобы создать ожидаемый таймер, достаточно вызвать функцию *CreateWaitableTimer*:

```

HANDLE CreateWaitableTimer(
    PSECURITY_ATTRIBUTES psa,
    BOOL fManualReset,
    PCTSTR pszName);

```

О параметрах *psa* и *pszName* я уже рассказывал в главе 3. Разумеется, любой процесс может получить свой («процессо-зависимый») описатель существующего объекта «ожидаемый таймер», вызвав *OpenWaitableTimer*:

```

HANDLE OpenWaitableTimer(
    DWORD dwDesiredAccess,
    BOOL bInheritHandle,
    PCTSTR pszName);

```

По аналогии с событиями параметр *fManualReset* определяет тип ожидаемого таймера: со сбросом вручную или с автосбросом. Когда освобождается таймер со сбросом вручную, возобновляется выполнение всех потоков, ожидавших этот объект, а когда в свободное состояние переходит таймер с автосбросом — лишь одного из потоков.

Объекты «ожидаемый таймер» всегда создаются в занятом состоянии. Чтобы сообщить таймеру, в какой момент он должен перейти в свободное состояние, вызовите функцию *SetWaitableTimer*:

```

BOOL SetWaitableTimer(
    HANDLE hTimer,
    const LARGE_INTEGER *pDueTime,
    LONG lPeriod,
    PTIMERAPCROUTINE pfnCompletionRoutine,
    PVOID pvArgToCompletionRoutine,
    BOOL fResume);

```

Эта функция принимает несколько параметров, в которых легко запутаться. Очевидно, что *hTimer* определяет нужный таймер. Следующие два параметра (*pDueTime* и *lPeriod*) используются совместно: первый из них задает, когда таймер должен сработать в первый раз, второй определяет, насколько часто это должно происходить в дальнейшем. Попробуем для примера установить таймер так, чтобы в первый раз он сработал 1 января 2002 года в 1:00 PM, а потом срабатывал каждые 6 часов.

```
// объявляем свои локальные переменные
HANDLE hTimer;
SYSTEMTIME st;
FILETIME ftLocal, ftUTC;
LARGE_INTEGER liUTC;

// создаем таймер с автосбросом
hTimer = CreateWaitableTimer(NULL, FALSE, NULL);

// таймер должен сработать в первый раз 1 января 2002 года в 1:00 PM
// по местному времени
st.wYear      = 2002;    // год
st.wMonth     = 1;       // январь
st.wDayOfWeek = 0;      // игнорируется
st.wDay        = 1;       // первое число месяца
st.wHour       = 13;     // 1 PM
st.wMinute     = 0;       // 0 минут
st.wSecond     = 0;       // 0 секунд
st.wMilliseconds = 0;   // 0 миллисекунд

SystemTimeToFileTime(&st, &ftLocal);

// преобразуем местное время в UTC-время
LocalFileTimeToFileTime(&ftLocal, &ftUTC);
// преобразуем FILETIME в LARGE_INTEGER из-за различий в выравнивании данных
liUTC.LowPart = ftUTC.dwLowDateTime;
liUTC.HighPart = ftUTC.dwHighDateTime;
// устанавливаем таймер
SetWaitableTimer(hTimer, &liUTC, 6 * 60 * 60 * 1000, NULL, NULL, FALSE);

:
```

Этот фрагмент кода сначала инициализирует структуру SYSTEMTIME, определяя время первого срабатывания таймера (его перехода в свободное состояние). Я установил это время как местное. Второй параметр представляется как *const LARGE_INTEGER ** и поэтому не позволяет напрямую использовать структуру SYSTEMTIME. Однако двоичные форматы структур FILETIME и LARGE_INTEGER идентичны: обе содержат по два 32-битных значения. Таким образом, мы можем преобразовать структуру SYSTEMTIME в FILETIME. Другая проблема заключается в том, что функция *SetWaitableTimer* ждет передачи времени в формате UTC (Coordinated Universal Time). Нужное преобразование легко осуществляется вызовом *LocalFileTimeToFileTime*.

Поскольку двоичные форматы структур FILETIME и LARGE_INTEGER идентичны, у Вас может появиться искушение передать в *SetWaitableTimer* адрес структуры FILETIME напрямую:

```
// устанавливаем таймер
SetWaitableTimer(hTimer, (PLARGE_INTEGER) &ftUTC,
 6 * 60 * 60 * 1000, NULL, NULL, FALSE);
```

В сущности, разбираясь с этой функцией, я так и поступил. Но это большая ошибка! Хотя двоичные форматы структур FILETIME и LARGE_INTEGER совпадают, выравнивание этих структур осуществляется по-разному. Адрес любой структуры FILETIME должен начинаться на 32-битной границе, а адрес любой структуры LARGE_INTEGER — на 64-битной. Вызов *SetWaitableTimer* с передачей ей структуры FILETIME может сра-

ботать корректно, но может и не сработать — все зависит от того, попадет ли начало структуры FILETIME на 64-битную границу. В то же время компилятор гарантирует, что структура LARGE_INTEGER всегда будет начинаться на 64-битной границе, и поэтому правильнее скопировать элементы FILETIME в элементы LARGE_INTEGER, а затем передать в *SetWaitableTimer* адрес именно структуры LARGE_INTEGER.



Процессоры x86 всегда «молча» обрабатывают ссылки на невыровненные данные. Поэтому передача в *SetWaitableTimer* адреса структуры FILETIME будет срабатывать, если приложение выполняется на машине с процессором x86. Однако другие процессоры (например, Alpha) в таких случаях, как правило, генерируют исключение EXCEPTION_DATATYPE_MISALIGNMENT, которое приводит к завершению Вашего процесса. Ошибки, связанные с выравниванием данных, — самый серьезный источник проблем при переносе на другие процессорные платформы программного кода, корректно работавшего на процессорах x86. Так что, обратив внимание на проблемы выравнивания данных сейчас, Вы сэкономите себе месяцы труда при переносе программы на другие платформы в будущем! Подробнее о выравнивании данных см. главу 13.

Чтобы разобраться в том, как заставить таймер срабатывать каждые 6 часов (начиная с 1:00 PM 1 января 2002 года), рассмотрим параметр *lPeriod* функции *SetWaitableTimer*. Этот параметр определяет последующую частоту срабатывания таймера (в мс). Чтобы установить 6 часов, я передаю значение, равное 21 600 000 мс (т. е. 6 часов • 60 минут • 60 секунд • 1000 миллисекунд).

О последних трех параметрах функции *SetWaitableTimer* мы поговорим ближе к концу этого раздела, а сейчас продолжим обсуждение второго и третьего параметров. Вместо того чтобы устанавливать время первого срабатывания таймера в абсолютных единицах, Вы можете задать его в относительных единицах (в интервалах по 100 нс); при этом число должно быть отрицательным. (Одна секунда равна десяти миллионам интервалов по 100 нс.)

Следующий код демонстрирует, как установить таймер на первое срабатывание через 5 секунд после вызова *SetWaitableTimer*:

```
// объявляем свои локальные переменные
HANDLE hTimer;
LARGE_INTEGER li;

// создаем таймер с автосбросом
hTimer = CreateWaitableTimer(NULL, FALSE, NULL);

// таймер должен сработать через 5 секунд после вызова SetWaitableTimer;
// задаем время в интервалах по 100 нс
const int nTimerUnitsPerSecond = 10000000;

// делаем полученное значение отрицательным, чтобы SetWaitableTimer
// знала: нам нужно относительное, а не абсолютное время
li.QuadPart = -(5 * nTimerUnitsPerSecond);

// устанавливаем таймер (он срабатывает сначала через 5 секунд,
// а потом через каждые 6 часов)
SetWaitableTimer(hTimer, &li, 6 * 60 * 60 * 1000, NULL, NULL, FALSE);

:
```

Обычно нужно, чтобы таймер сработал только раз — через определенное (абсолютное или относительное) время перешел в свободное состояние и уже больше никогда не срабатывал. Для этого достаточно передать 0 в параметре *lPeriod*. Затем можно либо вызвать *CloseHandle*, чтобы закрыть таймер, либо перенастроить таймер повторным вызовом *SetWaitableTimer* с другими параметрами.

И о последнем параметре функции *SetWaitableTimer* — *fResume*. Он полезен на компьютерах с поддержкой режима сна. Обычно в нем передают FALSE, и в приведенных ранее фрагментах кода я тоже делал так. Но если Вы, скажем, пишете программу-планировщик, которая позволяет устанавливать таймеры для напоминания о запланированных встречах, то должны передавать в этом параметре TRUE. Когда таймер сработает, машина выйдет из режима сна (если она находилась в нем), и пробудятся потоки, ожидающие этот таймер. Далее программа сможет проиграть какой-нибудь WAV-файл и вывести окно с напоминанием о предстоящей встрече. Если же Вы передадите FALSE в параметре *fResume*, объект-таймер перейдет в свободное состояние, но ожидающие его потоки не получат процессорное время, пока компьютер не выйдет из режима сна.

Рассмотрение ожидаемых таймеров было бы неполным, пропустив мы функцию *CancelWaitableTimer*:

```
BOOL CancelWaitableTimer(HANDLE hTimer);
```

Эта очень простая функция принимает описатель таймера и отменяет его (таймер), после чего тот уже никогда не сработает, — если только Вы не переустановите его повторным вызовом *SetWaitableTimer*. Кстати, если Вам понадобится перенастроить таймер, то вызывать *CancelWaitableTimer* перед повторным обращением к *SetWaitableTimer* не требуется; каждый вызов *SetWaitableTimer* автоматически отменяет предыдущие настройки перед установкой новых.

Ожидаемые таймеры и APC-очередь

Теперь Вы знаете, как создавать и настраивать таймер. Вы также научились приостанавливать потоки на таймере, передавая его описатель в *WaitForSingleObjects* или *WaitForMultipleObjects*. Однако у Вас есть возможность создать очередь асинхронных вызовов процедур (asynchronous procedure call, APC) для потока,зывающего *SetWaitableTimer* в момент, когда таймер свободен.

Обычно при обращении к функции *SetWaitableTimer* Вы передаете NULL в параметрах *pfnCompletionRoutine* и *pvArgToCompletionRoutine*. В этом случае объект-таймер переходит в свободное состояние в заданное время. Чтобы таймер в этот момент поместил в очередь вызовов APC-функции, нужно реализовать данную функцию и передать ее адрес в *SetWaitableTimer*. APC-функция должна выглядеть примерно так:

```
VOID APIENTRY TimerAPCRoutine(PVOID pvArgToCompletionRoutine,
    DWORD dwTimerLowValue, DWORD dwTimerHighValue) {
    // здесь делаем то, что нужно
}
```

Я назвал эту функцию *TimerAPCRoutine*, но Вы можете назвать ее как угодно. Она вызывается из того потока, который обратился к *SetWaitableTimer* в момент срабатывания таймера, — но только если вызывающий поток находится в «тревожном» (alertable) состоянии, т. е. ожидает этого в вызове одной из функций: *SleepEx*, *WaitForSingleObjectEx*, *WaitForMultipleObjectsEx*, *MsgWaitForMultipleObjectsEx* или *SignalObjectAndWait*. Если же поток этого не ожидает в любой из перечисленных функций, система не

поставит в очередь APC-функцию таймера. Тем самым система не даст APC-очереди потока переполниться уведомлениями от таймера, которые могли бы впустую израсходовать колоссальный объем памяти.

Если в момент срабатывания таймера Ваш поток находится в одной из перечисленных ранее функций, система заставляет его вызвать процедуру обратного вызова. Первый ее параметр совпадает с параметром *pvArgToCompletionRoutine*, передаваемым в функцию *SetWaitableTimer*. Это позволяет передавать в *TimerAPCRoutine* какие-либо данные (обычно указатель на определенную Вами структуру). Остальные два параметра, *dwTimerLowValue* и *dwTimerHighValue*, задают время срабатывания таймера. Код, приведенный ниже, демонстрирует, как принять эту информацию и показать ее пользователю.

```
VOID APIENTRY TimerAPCRoutine(PVOID pvArgToCompletionRoutine,
    DWORD dwTimerLowValue, DWORD dwTimerHighValue) {

    FILETIME ftUTC, ftLocal;
    SYSTEMTIME st;
    TCHAR szBuf[256];

    // записываем время в структуру FILETIME
    ftUTC.dwLowDateTime = dwTimerLowValue;
    ftUTC.dwHighDateTime = dwTimerHighValue;

    // преобразуем UTC-время в местное
    FileTimeToLocalFileTime(&ftUTC, &ftLocal);

    // преобразуем структуру FILETIME в структуру SYSTEMTIME,
    // как того требуют функции GetDateFormat и GetTimeFormat
    FileTimeToSystemTime(&ftLocal, &st);

    // формируем строку с датой и временем, в которое
    // сработал таймер
    GetDateFormat(LOCALE_USER_DEFAULT, DATE_LONGDATE,
        &st, NULL, szBuf, sizeof(szBuf) / sizeof(TCHAR));
    _tcscat(szBuf, _TEXT(" "));
    GetTimeFormat(LOCALE_USER_DEFAULT, 0, &st, NULL, _tcschr(szBuf, 0),
        sizeof(szBuf) / sizeof(TCHAR) - _tcslen(szBuf));

    // показываем время пользователю
    MessageBox(NULL, szBuf, "Timer went off at...", MB_OK);
}
```

Функция «тревожного ожидания» возвращает управление только после обработки всех элементов APC-очереди. Поэтому Вы должны позаботиться о том, чтобы Ваша функция *TimerAPCRoutine* заканчивала свою работу до того, как таймер вновь подаст сигнал (перейдет в свободное состояние). Иначе говоря, элементы не должны ставиться в APC-очередь быстрее, чем они могут быть обработаны.

Следующий фрагмент кода показывает, как правильно пользоваться таймерами и APC:

```
void SomeFunc() {
    // создаем таймер (его тип не имеет значения)
    HANDLE hTimer = CreateWaitableTimer(NULL, TRUE, NULL);
```

см. след. стр.

```
// настраиваем таймер на срабатывание через 5 секунд
LARGE_INTEGER li = { 0 };
SetWaitableTimer(hTimer, &li, 5000, TimerAPCRoutine, NULL, FALSE);
// ждем срабатывания таймера в "тревожном" состоянии
SleepEx(INFINITE, TRUE);

CloseHandle(hTimer);
}
```

И последнее. Взгляните на этот фрагмент кода:

```
HANDLE hTimer = CreateWaitableTimer(NULL, FALSE, NULL);
SetWaitableTimer(hTimer, ..., TimerAPCRoutine, ...);
WaitForSingleObjectEx(hTimer, INFINITE, TRUE);
```

Никогда не пишите такой код, потому что вызов *WaitForSingleObjectEx* на деле заставляет дважды ожидать таймер — по описателю *hTimer* и в «тревожном» состоянии. Когда таймер перейдет в свободное состояние, поток пробудится, что выведет его из «тревожного» состояния, и вызова APC-функции не последует. Правда, APC-функции редко используются совместно с ожидаемыми таймерами, так как всегда можно дождаться перехода таймера в свободное состояние, а затем сделать то, что нужно.

И еще кое-что о таймерах

Таймеры часто применяются в коммуникационных протоколах. Например, если клиент делает запрос серверу и тот не отвечает в течение определенного времени, клиент считает, что сервер не доступен. Сегодня клиентские машины взаимодействуют, как правило, со множеством серверов одновременно. Если бы объект ядра «таймер» создавался для каждого запроса, производительность системы снизилась бы весьма заметно. В большинстве приложений можно создавать единственный объект-таймер и по мере необходимости просто изменять время его срабатывания.

Постоянное отслеживание параметров таймера и его перенастройка довольно утомительны, из-за чего реализованы лишь в немногих приложениях. Однако в числе новых функций для операций с пулами потоков (о них — в главе 11) появилась *CreateTimerQueueTimer* — она как раз и берет на себя всю эту рутинную работу. Приступитесь к ней, если в Вашей программе приходится создавать несколько объектов-таймеров и управлять ими.

Конечно, очень мило, что таймеры поддерживают APC-очереди, но большинство современных приложений использует не APC, а порты завершения ввода-вывода. Как-то раз мне понадобилось, чтобы один из потоков в пуле (управляя его через порт завершения ввода-вывода) пробуждался по таймеру через определенные интервалы времени. К сожалению, такую функциональность ожидаемые таймеры не поддерживают. Для решения этой задачи мне пришлось создать отдельный поток, который все-го-то и делал, что настраивал ожидаемый таймер и ждал его освобождения. Когда таймер переходил в свободное состояние, этот поток вызывал *PostQueuedCompletionStatus*, передавая соответствующее уведомление потоку в пуле.

Любой, мало-мальски опытный Windows-программист непременно поинтересуется различиями ожидаемых таймеров и таймеров User (настраиваемых через функцию *SetTimer*). Так вот, главное отличие в том, что ожидаемые таймеры реализованы в ядре, а значит, не столь тяжеловесны, как таймеры User. Кроме того, это означает, что ожидаемые таймеры — объекты защищенные.

Таймеры User генерируют сообщения WM_TIMER, посыпаемые тому потоку, который вызвал *SetTimer* (в случае таймеров с обратной связью) или создал определенное

окно (в случае оконных таймеров). Таким образом, о срабатывании таймера User уведомляется только один поток. А ожидаемый таймер позволяет ждать любому числу потоков, и, если это таймер со сбросом вручную, при его освобождении может пробуждаться сразу несколько потоков.

Если в ответ на срабатывание таймера Вы собираетесь выполнять какие-то операции, связанные с пользовательским интерфейсом, то, по-видимому, будет легче структурировать код под таймеры User, поскольку применение ожидаемых таймеров требует от потоков ожидания не только сообщений, но и объектов ядра. (Если у Вас есть желание переделать свой код, используйте функцию *MsgWaitForMultipleObjects*, которая как раз и рассчитана на такие ситуации.) Наконец, в случае ожидаемых таймеров Вы с большей вероятностью будете получать уведомления именно по истечении заданного интервала. Как поясняется в главе 26, сообщения WM_TIMER всегда имеют наименьший приоритет и принимаются, только когда в очереди потока нет других сообщений. Но ожидаемый таймер обрабатывается так же, как и любой другой объект ядра: если он сработал, ждущий поток немедленно пробуждается.

Семафоры

Объекты ядра «семафор» используются для учета ресурсов. Как и все объекты ядра, они содержат счетчик числа пользователей, но, кроме того, поддерживают два 32-битных значения со знаком: одно определяет максимальное число ресурсов (контролируемое семафором), другое используется как счетчик текущего числа ресурсов.

Попробуем разобраться, зачем нужны все эти счетчики, и для примера рассмотрим программу, которая могла бы использовать семафоры. Допустим, я разрабатываю серверный процесс, в адресном пространстве которого выделяется буфер для хранения клиентских запросов. Размер этого буфера «зашит» в код программы и рассчитан на хранение максимум пяти клиентских запросов. Если новый клиент пытается связаться с сервером, когда эти пять запросов еще не обработаны, генерируется ошибка, которая сообщает клиенту, что сервер занят и нужно повторить попытку позже. При инициализации мой серверный процесс создает пул из пяти потоков, каждый из которых готов обрабатывать клиентские запросы по мере их поступления.

Изначально, когда запросов от клиентов еще нет, сервер не разрешает выделять процессорное время каким-либо потокам в пуле. Но как только серверу поступает, скажем, три клиентских запроса одновременно, три потока в пуле становятся планируемыми, и система начинает выделять им процессорное время. Для слежения за ресурсами и планированием потоков семафор очень удобен. Максимальное число ресурсов задается равным 5, что соответствует размеру буфера. Счетчик текущего числа ресурсов первоначально получает нулевое значение, так как клиенты еще не выдали ни одного запроса. Этот счетчик увеличивается на 1 в момент приема очередного клиентского запроса и на столько же уменьшается, когда запрос передается на обработку одному из серверных потоков в пуле.

Для семафоров определены следующие правила:

- когда счетчик текущего числа ресурсов становится больше 0, семафор переходит в свободное состояние;
- если этот счетчик равен 0, семафор занят;
- система не допускает присвоения отрицательных значений счетчику текущего числа ресурсов;
- счетчик текущего числа ресурсов не может быть больше максимального числа ресурсов.

Не путайте счетчик текущего числа ресурсов со счетчиком числа пользователей объекта-семафора.

Объект ядра «семафор» создается вызовом *CreateSemaphore*:

```
HANDLE CreateSemaphore(
    PSECURITY_ATTRIBUTES psa,
    LONG lInitialCount,
    LONG lMaximumCount,
    PCTSTR pszName);
```

О параметрах *psa* и *pszName* я рассказывал в главе 3. Разумеется, любой процесс может получить свой («процессо-зависимый») описатель существующего объекта «семафор», вызвав *OpenSemaphore*:

```
HANDLE OpenSemaphore(
    DWORD fdwAccess,
    BOOL bInheritHandle,
    PCTSTR pszName);
```

Параметр *lMaximumCount* сообщает системе максимальное число ресурсов, обрабатываемое Вашим приложением. Поскольку это 32-битное значение со знаком, предельное число ресурсов может достигать 2 147 483 647. Параметр *lInitialCount* указывает, сколько из этих ресурсов доступно изначально (на данный момент). При инициализации моего серверного процесса клиентских запросов нет, поэтому я вызываю *CreateSemaphore* так:

```
HANDLE hSem = CreateSemaphore(NULL, 0, 5, NULL);
```

Это приводит к созданию семафора со счетчиком максимального числа ресурсов, равным 5, при этом изначально ни один ресурс не доступен. (Кстати, счетчик числа пользователей данного объекта ядра равен 1, так как я только что создал этот объект; не запутайтесь в счетчиках.) Поскольку счетчику текущего числа ресурсов присвоен 0, семафор находится в занятом состоянии. А это значит, что любой поток, ждущий семафор, просто засыпает.

Поток получает доступ к ресурсу, вызывая одну из *Wait*-функций и передавая ей описатель семафора, который охраняет этот ресурс. *Wait*-функция проверяет у семафора счетчик текущего числа ресурсов: если его значение больше 0 (семафор свободен), уменьшает значение этого счетчика на 1, и вызывающий поток остается планируемым. Очень важно, что семафоры выполняют эту операцию проверки и присвоения на уровне атомарного доступа; иначе говоря, когда Вы запрашиваете у семафора какой-либо ресурс, операционная система проверяет, доступен ли этот ресурс, и, если да, уменьшает счетчик текущего числа ресурсов, не позволяя вмешиваться в эту операцию другому потоку. Только после того как счетчик ресурсов будет уменьшен на 1, доступ к ресурсу сможет запросить другой поток.

Если *Wait*-функция определяет, что счетчик текущего числа ресурсов равен 0 (семафор занят), система переводит вызывающий поток в состояние ожидания. Когда другой поток увеличит значение этого счетчика, система вспомнит о ждущем потоке и снова начнет выделять ему процессорное время (а он, захватив ресурс, уменьшил значение счетчика на 1).

Поток увеличивает значение счетчика текущего числа ресурсов, вызывая функцию *ReleaseSemaphore*:

```
BOOL ReleaseSemaphore(
    HANDLE hSem,
```

```
LONG lReleaseCount,
PLONG plPreviousCount);
```

Она просто складывает величину *lReleaseCount* со значением счетчика текущего числа ресурсов. Обычно в параметре *lReleaseCount* передают 1, но это вовсе не обязательно: я часто передаю в нем значения, равные или большие 2. Функция возвращает исходное значение счетчика ресурсов в **plPreviousCount*. Если Вас не интересует это значение (а в большинстве программ так оно и есть), передайте в параметре *plPreviousCount* значение NULL.

Было бы удобнее определять состояние счетчика текущего числа ресурсов, не меняя его значение, но такой функции в Windows нет. Поначалу я думал, что вызовом *ReleaseSemaphore* с передачей ей во втором параметре нуля можно узнать истинное значение счетчика в переменной типа LONG, на которую указывает параметр *plPreviousCount*. Но не вышло: функция занесла туда нуль. Я передал во втором параметре заведомо большее число, и — тот же результат. Тогда мне стало ясно: получить значение этого счетчика, не изменив его, невозможно.

Мьютексы

Объекты ядра «мьютексы» гарантируют потокам взаимоисключающий доступ к единственному ресурсу. Отсюда и произошло название этих объектов (mutual exclusion, mutex). Они содержат счетчик числа пользователей, счетчик рекурсии и переменную, в которой запоминается идентификатор потока. Мьютексы ведут себя точно так же, как и критические секции. Однако, если последние являются объектами пользовательского режима, то мьютексы — объектами ядра. Кроме того, единственный объект-мьютекс позволяет синхронизировать доступ к ресурсу нескольких потоков из разных процессов; при этом можно задать максимальное время ожидания доступа к ресурсу.

Идентификатор потока определяет, какой поток захватил мьютекс, а счетчик рекурсий — сколько раз. У мьютексов много применений, и это наиболее часто используемые объекты ядра. Как правило, с их помощью защищают блок памяти, к которому обращается множество потоков. Если бы потоки одновременно использовали какой-то блок памяти, данные в нем были бы повреждены. Мьютексы гарантируют, что любой поток получает монопольный доступ к блоку памяти, и тем самым обеспечивают целостность данных.

Для мьютексов определены следующие правила:

- если его идентификатор потока равен 0 (у самого потока не может быть такой идентификатор), мьютекс не захвачен ни одним из потоков и находится в свободном состоянии;
- если его идентификатор потока не равен 0, мьютекс захвачен одним из потоков и находится в занятом состоянии;
- в отличие от других объектов ядра мьютексы могут нарушать обычные правила, действующие в операционной системе (об этом — чуть позже).

Для использования объекта-мьютекса один из процессов должен сначала создать его вызовом *CreateMutex*:

```
HANDLE CreateMutex(
    PSECURITY_ATTRIBUTES psa,
    BOOL fInitialOwner,
    PCTSTR pszName);
```

О параметрах *psa* и *pszName* я рассказывал в главе 3. Разумеется, любой процесс может получить свой («процессо-зависимый») описатель существующего объекта «мьютекс», вызвав *OpenMutex*:

```
HANDLE OpenMutex(  
    DWORD fdwAccess,  
    BOOL bInheritHandle,  
    PCTSTR pszName);
```

Параметр *fInitialOwner* определяет начальное состояние мьютекса. Если в нем передается FALSE (что обычно и бывает), объект-мьютекс не принадлежит ни одному из потоков и поэтому находится в свободном состоянии. При этом его идентификатор потока и счетчик рекурсии равны 0. Если же в нем передается TRUE, идентификатор потока, принадлежащий мьютексу, приравнивается идентификатору вызывающего потока, а счетчик рекурсии получает значение 1. Поскольку теперь идентификатор потока отличен от 0, мьютекс изначально находится в занятом состоянии.

Поток получает доступ к разделяемому ресурсу, вызывая одну из *Wait*-функций и передавая ей описатель мьютекса, который охраняет этот ресурс. *Wait*-функция проверяет у мьютекса идентификатор потока: если его значение не равно 0, мьютекс свободен; в ином случае оно принимает значение идентификатора вызывающего потока, и этот поток остается планируемым.

Если *Wait*-функция определяет, что у мьютекса идентификатор потока не равен 0 (мьютекс занят), вызывающий поток переходит в состояние ожидания. Система запоминает это и, когда идентификатор обнуляется, записывает в него идентификатор ждущего потока, а счетчику рекурсии присваивает значение 1, после чего ждущий поток вновь становится планируемым. Все проверки и изменения состояния объекта-мьютекса выполняются на уровне атомарного доступа.

Для мьютексов сделано одно исключение в правилах перехода объектов ядра из одного состояния в другое. Допустим, поток ждет освобождения занятого объекта-мьютекса. В этом случае поток обычно засыпает (переходит в состояние ожидания). Однако система проверяет, не совпадает ли идентификатор потока, пытающегося захватить мьютекс, с аналогичным идентификатором у мьютекса. Если они совпадают, система по-прежнему выделяет потоку процессорное время, хотя мьютекс все еще занят. Подобных особенностей в поведении нет ни у каких других объектов ядра в системе. Всякий раз, когда поток захватывает объект-мьютекс, счетчик рекурсии в этом объекте увеличивается на 1. Единственная ситуация, в которой значение счетчика рекурсии может быть больше 1, — поток захватывает один и тот же мьютекс несколько раз, пользуясь упомянутым исключением из общих правил.

Когда ожидание мьютекса потоком успешно завершается, последний получает монопольный доступ к защищенному ресурсу. Все остальные потоки, пытающиеся обратиться к этому ресурсу, переходят в состояние ожидания. Когда поток, занимающий ресурс, заканчивает с ним работать, он должен освободить мьютекс вызовом функции *ReleaseMutex*:

```
BOOL ReleaseMutex(HANDLE hMutex);
```

Эта функция уменьшает счетчик рекурсии в объекте-мьютексе на 1. Если данный объект передавался во владение потоку неоднократно, поток обязан вызвать *ReleaseMutex* столько раз, сколько необходимо для обнуления счетчика рекурсии. Как только счетчик станет равен 0, переменная, хранящая идентификатор потока, тоже обнуляется, и объект-мьютекс освободится. После этого система проверит, ожидают ли

освобождения мьютекса какие-нибудь другие потоки. Если да, система «по-честному» выберет один из ждущих потоков и передаст ему во владение объект-мьютекс.

Отказ от объекта-мьютекса

Объект-мьютекс отличается от остальных объектов ядра тем, что занявшему его потоку передаются права на владение им. Прочие объекты могут быть либо свободны, либо заняты — вот, собственно, и все. А объекты-мьютексы способны еще и запоминать, какому потоку они принадлежат. Если какой-то посторонний поток попытается освободить мьютекс вызовом функции *ReleaseMutex*, то она, проверив идентификаторы потоков и обнаружив их несовпадение, ничего делать не станет, а просто вернет FALSE. Тут же вызвав *GetLastError*, Вы получите значение ERROR_NOT_OWNER.

Отсюда возникает вопрос: а что будет, если поток, которому принадлежит мьютекс, завершится, не успев его освободить? В таком случае система считает, что произошел отказ от мьютекса, и автоматически переводит его в свободное состояние (сбрасывая при этом все его счетчики в исходное состояние). Если этот мьютекс ждут другие потоки, система, как обычно, «по-честному» выбирает один из потоков и позволяет ему захватить мьютекс. Тогда *Wait*-функция возвращает потоку WAIT_ABANDONED вместо WAIT_OBJECT_0, и тот узнает, что мьютекс освобожден некорректно. Данная ситуация, конечно, не самая лучшая. Выяснить, что сделал с защищенными данными завершенный поток — бывший владелец объекта-мьютекса, увы, невозможно.

В реальности программы никогда специально не проверяют возвращаемое значение на WAIT_ABANDONED, потому что такое завершение потоков происходит очень редко. (Вот, кстати, еще один яркий пример, доказывающий, что Вы не должны пользоваться функцией *TerminateThread*.)

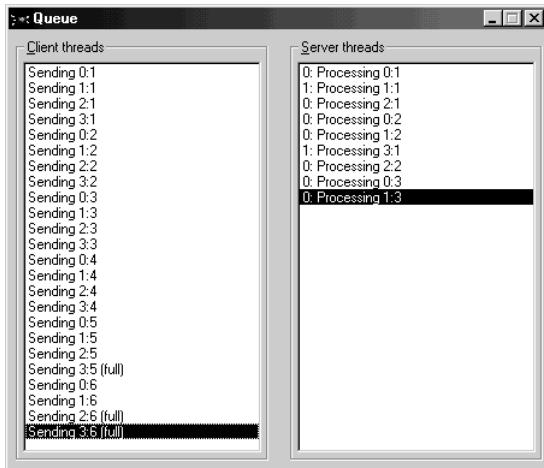
Мьютексы и критические секции

Мьютексы и критические секции одинаковы в том, как они влияют на планирование ждущих потоков, но различны по некоторым другим характеристикам. Эти объекты сравниваются в следующей таблице.

Характеристики	Объект-мьютекс	Объект — критическая секция
Быстродействие	Малое	Высокое
Возможность использования за границами процесса	Да	Нет
Объявление	<i>HANDLE bmtx;</i>	<i>CRITICAL_SECTION cs;</i>
Инициализация	<i>bmtx = CreateMutex(</i> <i>(NULL, FALSE, NULL);</i>	<i>InitializeCriticalSection(&cs);</i>
Очистка	<i>CloseHandle(bmtx);</i>	<i>DeleteCriticalSection(&cs);</i>
Бесконечное ожидание	<i>WaitForSingleObject(</i> <i>(bmtx, INFINITE);</i>	<i>EnterCriticalSection(&cs);</i>
Ожидание в течение 0 мс	<i>WaitForSingleObject(</i> <i>(bmtx, 0);</i>	<i>TryEnterCriticalSection(&cs);</i>
Ожидание в течение произвольного периода времени	<i>WaitForSingleObject(</i> <i>(bmtx, dwMilliseconds);</i>	Невозможно
Освобождение	<i>ReleaseMutex(bmtx);</i>	<i>LeaveCriticalSection(&cs);</i>
Возможность параллельного ожидания других объектов ядра	Да (с помощью <i>WaitForMultipleObjects</i> или аналогичной функции)	Нет

Программа-пример Queue

Эта программа, «09 Queue.exe» (см. листинг на рис. 9-2), управляет очередью обрабатываемых элементов данных, используя мьютекс и семафор. Файлы исходного кода и ресурсов этой программы находятся в каталоге 09-Queue на компакт-диске, прилагаемом к книге. После запуска Queue открывается окно, показанное ниже.



При инициализации Queue создает четыре клиентских и два серверных потока. Каждый клиентский поток засыпает на определенный период времени, а затем помещает в очередь элемент данных. Когда в очередь ставится новый элемент, содержимое списка Client Threads обновляется. Каждый элемент данных состоит из номера клиентского потока и порядкового номера запроса, выданного этим потоком. Например, первая запись в списке сообщает, что клиентский поток 0 поставил в очередь свой первый запрос. Следующие записи свидетельствуют, что далее свои первые запросы выдают потоки 1–3, потом поток 0 помещает второй запрос, то же самое делают остальные потоки, и все повторяется.

Серверные потоки ничего не делают, пока в очереди не появится хотя бы один элемент данных. Как только он появляется, для его обработки пробуждается один из серверных потоков. Состояние серверных потоков отражается в списке Server Threads. Первая запись говорит о том, что первый запрос от клиентского потока 0 обрабатывается серверным потоком 0, вторая запись — что первый запрос от клиентского потока 1 обрабатывается серверным потоком 1, и т. д.

В этом примере серверные потоки не успевают обрабатывать клиентские запросы и очередь в конечном счете заполняется до максимума. Я установил максимальную длину очереди равной 10 элементам, что приводит к быстрому заполнению этой очереди. Кроме того, на четыре клиентских потока приходится лишь два серверных. В итоге очередь полностью заполняется к тому моменту, когда клиентский поток 3 пытается выдать свой пятый запрос.

О'кэй, что делает программа, Вы поняли; теперь посмотрим — как она это делает (что гораздо интереснее). Очередь управляет C++-класс CQueue:

```
class CQueue {
public:
    struct ELEMENT {
        int m_nThreadNum, m_nRequestNum;
```

```

    // другие элементы данных должны быть определены здесь
};

typedef ELEMENT* PELEMENT;

private:
    PELEMENT m_pElements;           // массив элементов, подлежащих обработке
    int      m_nMaxElements;        // количество элементов в массиве
    HANDLE   m_h[2];               // описатели мьютекса и семафора
    HANDLE   &m_hmtxQ;              // ссылка на m_h[0]
    HANDLE   &m_hsemNumElements;    // ссылка на m_h[1]

public:
    CQueue(int nMaxElements);
    ~CQueue();

    BOOL Append(PELEMENT pElement, DWORD dwMilliseconds);
    BOOL Remove(PELEMENT pElement, DWORD dwMilliseconds);
};

```

Открытая структура ELEMENT внутри этого класса определяет, что представляет собой элемент данных, помещаемый в очередь. Его реальное содержимое в данном случае не имеет значения. В этой программе-примере клиентские потоки записывают в элемент данных собственный номер и порядковый номер своего очередного запроса, а серверные потоки, обрабатывая запросы, показывают эту информацию в списке. В реальном приложении такая информация вряд ли бы понадобилась.

Что касается закрытых элементов класса, мы имеем *m_pElements*, который указывает на массив (фиксированного размера) структур ELEMENT. Эти данные как раз и нужно защищать от одновременного доступа к ним со стороны клиентских и серверных потоков. Элемент *m_nMaxElements* определяет размер массива при создании объекта CQueue. Следующий элемент, *m_h*, — это массив из двух описателей объектов ядра. Для корректной защиты элементов данных в очереди нам нужно два объекта ядра: мьютекс и семафор. Эти два объекта создаются в конструкторе CQueue; в нем же их описатели помещаются в массив *m_h*.

Как Вы вскоре увидите, программа периодически вызывает *WaitForMultipleObjects*, передавая этой функции адрес массива описателей. Вы также убедитесь, что программа время от времени приходится ссылаться только на один из этих описателей. Чтобы облегчить чтение кода и его модификацию, я объявил два элемента, каждый из которых содержит ссылку на один из описателей, — *m_hmtxQ* и *m_hsemNumElements*. Конструктор CQueue инициализирует эти элементы содержимым *m_h[0]* и *m_h[1]* соответственно.

Теперь Вы и сами без труда разберетесь в методах конструктора и деструктора CQueue, поэтому я перейду сразу к методу *Append*. Этот метод пытается добавить ELEMENT в очередь. Но сначала он должен убедиться, что вызывающему потоку разрешен монопольный доступ к очереди. Для этого метод *Append* вызывает *WaitForSingleObject*, передавая ей описатель объекта-мьютекса, *m_hmtxQ*. Если функция возвращает WAIT_OBJECT_0, значит, поток получил монопольный доступ к очереди.

Далее метод *Append* должен попытаться увеличить число элементов в очереди, вызвав функцию *ReleaseSemaphore* и передав ей счетчик числа освобождений (release count), равный 1. Если вызов *ReleaseSemaphore* проходит успешно, в очереди еще есть место, и в нее можно поместить новый элемент. К счастью, *ReleaseSemaphore* возвращает в переменной *lPreviousCount* предыдущее количество элементов в очереди. Благодаря этому Вы точно знаете, в какой элемент массива следует записать новый эле-

мент данных. Скопировав элемент в массив очереди, функция возвращает управление. По окончании этой операции *Append* вызывает *ReleaseMutex*, чтобы и другие потоки могли получить доступ к очереди. Остальной код в методе *Append* отвечает за обработку ошибок и неудачных вызовов.

Теперь посмотрим, как серверный поток вызывает метод *Remove* для выборки элемента из очереди. Сначала этот метод должен убедиться, что вызывающий поток получил монопольный доступ к очереди и что в ней есть хотя бы один элемент. Разумеется, серверному потоку нет смысла пробуждаться, если очередь пуста. Поэтому метод *Remove* предварительно обращается к *WaitForMultipleObjects*, передавая ей описатели мьютекса и семафора. И только после освобождения обоих объектов серверный поток может пробудиться.

Если возвращается *WAIT_OBJECT_0*, значит, поток получил монопольный доступ к очереди и в ней есть хотя бы один элемент. В этот момент программа извлекает из массива элемент с индексом 0, а остальные элементы сдвигают вниз на одну позицию. Это, конечно, не самый эффективный способ реализации очереди, так как требует слишком большого количества операций копирования в памяти, но наша цель заключается лишь в том, чтобы продемонстрировать синхронизацию потоков. По окончании этих операций вызывается *ReleaseMutex*, и очередь становится доступной другим потокам.

Заметьте, что объект-семафор отслеживает, сколько элементов находится в очереди. Вы, наверное, сразу же поняли, что это значение увеличивается, когда метод *Append* вызывает *ReleaseSemaphore* после добавления нового элемента к очереди. Но как оно уменьшается после удаления элемента из очереди, уже не столь очевидно. Эта операция выполняется вызовом *WaitForMultipleObjects* из метода *Remove*. Тут надо вспомнить, что побочный эффект успешного ожидания семафора заключается в уменьшении его счетчика на 1. Очень удобно для нас.

Теперь, когда Вы понимаете, как работает класс *CQueue*, Вы легко разберетесь в остальном коде этой программы.



Queue.cpp

```
*****
Modуль: Queue.cpp
Автор: Copyright (c) 2000, Джейфри Рихтер (Jeffrey Richter)
*****
```

```
#include "..\CmnHdr.h"      /* см. приложение A */
#include <windowsx.h>
#include <tchar.h>
#include <process.h>        // для доступа к _beginthreadex
#include "Resource.h"

//////////
```

```
class CQueue {
public:
    struct ELEMENT {
        int m_nThreadNum, m_nRequestNum;
        // другие элементы данных должны быть определены здесь
    };
};
```

Рис. 9-2. Программа-пример *Queue*

Рис. 9-2. продолжение

```

typedef ELEMENT* PELEMENT;

private:
    PELEMENT m_pElements;           // массив элементов, подлежащих обработке
    int      m_nMaxElements;        // количество элементов в массиве
    HANDLE   m_h[2];               // описатели мьютекса и семафора
    HANDLE   &m_hmtxQ;              // ссылка на m_h[0]
    HANDLE   &m_hsemNumElements;    // ссылка на m_h[1]

public:
    CQueue(int nMaxElements);
    ~CQueue();

    BOOL Append(PELEMENT pElement, DWORD dwMilliseconds);
    BOOL Remove(PELEMENT pElement, DWORD dwMilliseconds);
};

///////////////////////////////
CQueue::CQueue(int nMaxElements)
: m_hmtxQ(m_h[0]), m_hsemNumElements(m_h[1]) {

    m_pElements = (PELEMENT)
        HeapAlloc(GetProcessHeap(), 0, sizeof(ELEMENT) * nMaxElements);
    m_nMaxElements = nMaxElements;
    m_hmtxQ = CreateMutex(NULL, FALSE, NULL);
    m_hsemNumElements = CreateSemaphore(NULL, 0, nMaxElements, NULL);
}

///////////////////////////////
CQueue::~CQueue() {
    CloseHandle(m_hsemNumElements);
    CloseHandle(m_hmtxQ);
    HeapFree(GetProcessHeap(), 0, m_pElements);
}

///////////////////////////////
BOOL CQueue::Append(PELEMENT pElement, DWORD dwTimeout) {

    BOOL fOk = FALSE;
    DWORD dw = WaitForSingleObject(m_hmtxQ, dwTimeout);

    if (dw == WAIT_OBJECT_0) {
        // этот поток получил монопольный доступ к очереди

        // увеличиваем число элементов в очереди
        LONG lPrevCount;
        fOk = ReleaseSemaphore(m_hsemNumElements, 1, &lPrevCount);
        if (!fOk) {
}

```

см. след. стр.

Рис. 9-2. продолжение

```
// в очереди еще есть место; добавляем новый элемент
m_pElements[1PrevCount] = *pElement;
} else {

    // очередь полностью заполнена; устанавливаем код ошибки
    // и сообщаем о неудачном завершении вызова
    SetLastError(ERROR_DATABASE_FULL);
}

// разрешаем другим потокам обращаться к очереди
ReleaseMutex(m_hmtxQ);

} else {
    // время ожидания истекло; устанавливаем код ошибки
    // и сообщаем о неудачном завершении вызова
    SetLastError(ERROR_TIMEOUT);
}

return(f0k); // GetLastError сообщает дополнительную информацию
}

///////////////////////////////



BOOL CQueue::Remove(PELEMENT pElement, DWORD dwTimeout) {

    // ждем монопольного доступа к очереди
    // и появления в ней хотя бы одного элемента
    BOOL f0k = (WaitForMultipleObjects(chDIMOF(m_h), m_h, TRUE, dwTimeout)
        == WAIT_OBJECT_0);

    if (f0k) {
        // в очереди есть элемент; извлекаем его
        *pElement = m_pElements[0];

        // сдвигаем остальные элементы вниз на одну позицию
        MoveMemory(&m_pElements[0], &m_pElements[1],
            sizeof(ELEMENT) * (m_nMaxElements - 1));

        // разрешаем другим потокам обращаться к очереди
        ReleaseMutex(m_hmtxQ);

    } else {
        // время ожидания истекло; устанавливаем код ошибки
        // и сообщаем о неудачном завершении вызова
        SetLastError(ERROR_TIMEOUT);
    }

    return(f0k); // GetLastError сообщает дополнительную информацию
}

///////////////////////////////
```

Рис. 9-2. продолжение

```

CQueue g_q(10);           // совместно используемая очередь;
volatile BOOL g_fShutdown = FALSE; // сигнализирует клиентским и серверным потокам,
                                  // когда им нужно завершаться;
HWND g_hwnd;              // позволяет выяснить состояние
                           // клиентских и серверных потоков

// описатели и количество всех потоков (клиентских и серверных)
HANDLE g_hThreads[MAXIMUM_WAIT_OBJECTS];
int g_nNumThreads = 0;

///////////////////////////////



DWORD WINAPI ClientThread(PVOID pvParam) {

    int nThreadNum = PtrToUlong(pvParam);
    HWND hwndLB = GetDlgItem(g_hwnd, IDC_CLIENTS);

    for (int nRequestNum = 1; !g_fShutdown; nRequestNum++) {

        TCHAR sz[1024];
        CQueue::ELEMENT e = { nThreadNum, nRequestNum };

        // пытаемся поместить элемент в очередь
        if (g_q.Append(&e, 200)) {

            // указываем номера потока и запроса
            wsprintf(sz, TEXT("Sending %d:%d"), nThreadNum, nRequestNum);
        } else {

            // поставить элемент в очередь не удалось
            wsprintf(sz, TEXT("Sending %d:%d (%s)"),
                     nThreadNum, nRequestNum,
                     (GetLastError() == ERROR_TIMEOUT)
                     ? TEXT("timeout") : TEXT("full"));
        }

        // показываем результат добавления элемента
        ListBox_SetCurSel(hwndLB, ListBox_AddString(hwndLB, sz));
        Sleep(2500); // интервал ожидания до добавления следующего элемента
    }

    return(0);
}

///////////////////////////////



DWORD WINAPI ServerThread(PVOID pvParam) {

    int nThreadNum = PtrToUlong(pvParam);
    HWND hwndLB = GetDlgItem(g_hwnd, IDC_SERVERS);

    while (!g_fShutdown) {

```

см. след. стр.

Рис. 9-2. продолжение

```
TCHAR sz[1024];
CQueue::ELEMENT e;

// пытаемся получить элемент из очереди
if (g_q.Remove(&e, 5000)) {

    // сообщаем, какой поток обрабатывает этот элемент,
    // какой поток поместил его в очередь и какой он по счету
    wsprintf(sz, TEXT("%d: Processing %d:%d"),
        nThreadNum, e.m_nThreadNum, e.m_nRequestNum);

    // на обработку запроса серверу нужно какое-то время
    Sleep(2000 * e.m_nThreadNum);

} else {
    // получить элемент из очереди не удалось
    wsprintf(sz, TEXT("%d: (timeout)"), nThreadNum);
}

// показываем результат обработки элемента
ListBox_SetCurSel(hwndLB, ListBox_AddString(hwndLB, sz));
}

return(0);
}

///////////
BOOL Dlg_OnInitDialog(HWND hwnd, HWND hwndFocus, LPARAM lParam) {

    chSETDLGICONS(hwnd, IDI_QUEUE);

    g_hwnd = hwnd; // используется клиентскими и серверными потоками
                    // для уведомления о своем состоянии

    DWORD dwThreadID;

    // создаем клиентские потоки
    for (int x = 0; x < 4; x++)
        g_hThreads[g_nNumThreads++] =
            chBEGINTHREADEX(NULL, 0, ClientThread, (PVOID) (INT_PTR) x,
                0, &dwThreadID);

    // создаем серверные потоки
    for (x = 0; x < 2; x++)
        g_hThreads[g_nNumThreads++] =
            chBEGINTHREADEX(NULL, 0, ServerThread, (PVOID) (INT_PTR) x,
                0, &dwThreadID);

    return(TRUE);
}
```

Рис. 9-2. продолжение

```
//////////  

void Dlg_OnCommand(HWND hwnd, int id, HWND hwndCtl, UINT codeNotify) {  

    switch (id) {  

        case IDCANCEL:  

            EndDialog(hwnd, id);  

            break;  

    }  

}  

//////////  

INT_PTR WINAPI Dlg_Proc(HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam) {  

    switch (uMsg) {  

        chHANDLE_DLMSG(hwnd, WM_INITDIALOG, Dlg_OnInitDialog);  

        chHANDLE_DLMSG(hwnd, WM_COMMAND, Dlg_OnCommand);  

    }  

    return(FALSE);  

}  

//////////  

int WINAPI _tWinMain(HINSTANCE hinstExe, HINSTANCE, PTSTR pszCmdLine, int) {  

    DialogBox(hinstExe, MAKEINTRESOURCE(IDD_QUEUE), NULL, Dlg_Proc);  

    InterlockedExchangePointer((PVOID*) &g_fShutdown, (PVOID) TRUE);  

    // ждем завершения всех потоков, а затем проводим очистку  

    WaitForMultipleObjects(g_nNumThreads, g_hThreads, TRUE, INFINITE);  

    while (g_nNumThreads--)  

        CloseHandle(g_hThreads[g_nNumThreads]);  

    return(0);  

}  

////////// Конец файла //////////
```

Сводная таблица объектов, используемых для синхронизации потоков

В следующей таблице суммируются сведения о различных объектах ядра применительно к синхронизации потоков.

Объект	Находится в занятом состоянии, когда:	Переходит в свободное состояние, когда:	Побочный эффект успешного ожидания
Процесс	процесс еще активен	процесс завершается (<i>ExitProcess, TerminateProcess</i>)	Нет
Поток	поток еще активен	поток завершается (<i>ExitThread, TerminateThread</i>)	Нет

см. след. стр.

продолжение

Объект	Находится в занятом состоянии, когда:	Переходит в свободное состояние, когда:	Побочный эффект успешного ожидания
Задание	время, выделенное заданию, еще не истекло	время, выделенное заданию, истекло	Нет
Файл	выдан запрос на ввод-вывод	завершено выполнение запроса на ввод-вывод	Нет
Консольный ввод	ввода нет	ввод есть	Нет
Уведомление об изменении файла	в файловой системе нет изменений	файловая система обнаруживает изменения	Сбрасывается в исходное состояние
Событие с автосбросом	вызывается <i>ResetEvent</i> , <i>PulseEvent</i> или ожидание успешно завершилось	вызывается <i>SetEvent</i> или <i>PulseEvent</i>	Сбрасывается в исходное состояние
Событие со сбросом вручную	вызывается <i>ResetEvent</i> или <i>PulseEvent</i>	вызывается <i>SetEvent</i> или <i>PulseEvent</i>	Нет
Ожидаемый таймер с автосбросом	вызывается <i>CancelWaitableTimer</i> или ожидание успешно завершилось	наступает время срабатывания (<i>SetWaitableTimer</i>)	Сбрасывается в исходное состояние
Ожидаемый таймер со сбросом вручную	вызывается <i>CancelWaitableTimer</i>	наступает время срабатывания (<i>SetWaitableTimer</i>)	Нет
Семафор	ожидание успешно завершилось	счетчик > 0 (<i>ReleaseSemaphore</i>)	Счетчик уменьшается на 1
Мьютекс	ожидание успешно завершилось	поток освобождает мьютекс (<i>ReleaseMutex</i>)	Передается потоку во владение
Критическая секция (пользовательского режима)	ожидание успешно завершилось (<i>(Try)EnterCriticalSection</i>)	поток освобождает критическую секцию (<i>LeaveCriticalSection</i>)	Передается потоку во владение

Interlocked-функции (пользовательского режима) никогда не приводят к исключению потока из числа планируемых; они лишь изменяют какое-то значение и тут же возвращают управление.

Другие функции, применяемые в синхронизации потоков

При синхронизации потоков чаще всего используются функции *WaitForSingleObject* и *WaitForMultipleObjects*. Однако в Windows есть и другие, несколько отличающиеся функции, которые можно применять с той же целью. Если Вы понимаете, как работают *WaitForSingleObject* и *WaitForMultipleObjects*, Вы без труда разберетесь и в этих функциях.

Асинхронный ввод-вывод на устройствах

При асинхронном вводе-выводе поток начинает операцию чтения или записи и не ждет ее окончания. Например, если потоку нужно загрузить в память большой файл, он может сообщить системе сделать это за него. И пока система грузит файл в память, поток спокойно занимается другими задачами — создает окна, инициализирует внутренние структуры данных и т. д. Закончив, поток приостанавливает себя и ждет уведомления от системы о том, что загрузка файла завершена.

Объекты устройств являются синхронизируемыми объектами ядра, а это означает, что Вы можете вызывать *WaitForSingleObject* и передавать ей описатель какого-либо файла, сокета, коммуникационного порта и т. д. Пока система выполняет асинхронный ввод-вывод, объект устройства пребывает в занятом состоянии. Как только операция заканчивается, система переводит объект в свободное состояние, и поток узнает о завершении операции. С этого момента поток возобновляет выполнение.

Функция *WaitForInputIdle*

Поток может приостановить себя и вызовом *WaitForInputIdle*:

```
DWORD WaitForInputIdle(
    HANDLE hProcess,
    DWORD dwMilliseconds);
```

Эта функция ждет, пока у процесса, идентифицируемого описателем *hProcess*, не опустеет очередь ввода в потоке, создавшем первое окно приложения. *WaitForInputIdle* полезна для применения, например, в родительском процессе, который порождает дочерний для выполнения какой-либо нужной ему работы. Когда один из потоков родительского процесса вызывает *CreateProcess*, он продолжает выполнение и в то время, пока дочерний процесс инициализируется. Этому потоку может понадобиться описатель окна, созданного дочерним процессом. Единственная возможность узнать о моменте окончания инициализации дочернего процесса — дождаться, когда тот прекратит обработку любого ввода. Поэтому после вызова *CreateProcess* поток родительского процесса должен вызвать *WaitForInputIdle*.

Эту функцию можно применить и в том случае, когда Вы хотите имитировать в программе нажатие каких-либо клавиш. Допустим, Вы асинхронно отправили в главное окно приложения следующие сообщения:

WM_KEYDOWN	с виртуальной клавишей VK_MENU
WM_KEYDOWN	с виртуальной клавишей VK_F
WM_KEYUP	с виртуальной клавишей VK_F
WM_KEYUP	с виртуальной клавишей VK_MENU
WM_KEYDOWN	с виртуальной клавишей VK_O
WM_KEYUP	с виртуальной клавишей VK_O

Эта последовательность дает тот же эффект, что и нажатие клавиш Alt+F, O, — в большинстве англоязычных приложений это вызывает команду Open из меню File. Выбор данной команды открывает диалоговое окно; но, прежде чем оно появится на экране, Windows должна загрузить шаблон диалогового окна из файла и «пройтись» по всем элементам управления в шаблоне, вызывая для каждого из них функцию *CreateWindow*. Разумеется, на это уходит какое-то время. Поэтому приложение, асинхронно отправившее сообщения типа WM_KEY*, теперь может вызвать *WaitForInputIdle* и таким образом перейти в режим ожидания до того момента, как Windows закончит создание диалогового окна и оно будет готово к приему данных от пользователя. Далее программа может передать диалоговому окну и его элементам управления сообщения о еще каких-то клавишиах, что заставит диалоговое окно проделать те или иные операции.

С этой проблемой, кстати, сталкивались многие разработчики приложений для 16-разрядной Windows. Программам нужно было асинхронно передавать сообщения в окно, но получить точной информации о том, создано ли это окно и готово ли к работе, они не могли. Функция *WaitForInputIdle* решает эту проблему.

Функция *MsgWaitForMultipleObjects(Ex)*

При вызове *MsgWaitForMultipleObjects* или *MsgWaitForMultipleObjectsEx* поток переходит в ожидание своих (предназначенных этому потоку) сообщений:

```
DWORD MsgWaitForMultipleObjects(
    DWORD dwCount,
    PHANDLE phObjects,
    BOOL fWaitAll,
    DWORD dwMilliseconds,
    DWORD dwWakeMask);

DWORD MsgWaitForMultipleObjectsEx(
    DWORD dwCount,
    PHANDLE phObjects,
    DWORD dwMilliseconds,
    DWORD dwWakeMask
    DWORD dwFlags);
```

Эти функции аналогичны *WaitForMultipleObjects*. Единственное различие заключается в том, что они пробуждают поток, когда освобождается некий объект ядра или когда определенное оконное сообщение требует перенаправления в окно, созданное вызывающим потоком.

Поток, который создает окна и выполняет другие операции, относящиеся к пользовательскому интерфейсу, должен работать с функцией *MsgWaitForMultipleObjectsEx*, а не с *WaitForMultipleObjects*, так как последняя не дает возможности реагировать на действия пользователя. Подробнее эти функции рассматриваются в главе 26.

Функция *WaitForDebugEvent*

В Windows встроены богатейшие отладочные средства. Начиная исполнение, отладчик подключает себя к отлаживаемой программе, а потом просто ждет, когда операционная система уведомит его о каком-нибудь событии отладки, связанном с этой программой. Ожидание таких событий осуществляется через вызов:

```
BOOL WaitForDebugEvent(
    PDEBUG_EVENT pde,
    DWORD dwMilliseconds);
```

Когда отладчик вызывает *WaitForDebugEvent*, его поток приостанавливается. Система уведомит поток о событии отладки, разрешив функции *WaitForDebugEvent* вернуть управление. Структура, на которую указывает параметр *pde*, заполняется системой перед пробуждением потока отладчика. В ней содержится информация, касающаяся только что произошедшего события отладки.

Функция *SignalObjectAndWait*

SignalObjectAndWait переводит в свободное состояние один объект ядра и ждет другой объект ядра, выполняя все это как одну операцию на уровне атомарного доступа:

```
DWORD SignalObjectAndWait(
    HANDLE hObjectToSignal,
    HANDLE hObjectToWaitOn,
    DWORD dwMilliseconds,
    BOOL fAlertable);
```

Параметр *bObjectToSignal* должен идентифицировать мьютекс, семафор или событие; объекты любого другого типа заставят *SignalObjectAndWait* вернуть WAIT_FAILED, а функцию *GetLastError* — ERROR_INVALID_HANDLE. Функция *SignalObjectAndWait* проверяет тип объекта и выполняет действия, аналогичные тем, которые предпринимают функции *ReleaseMutex*, *ReleaseSemaphore* (со счетчиком, равным 1) или *ResetEvent*.

Параметр *bObjectToWaitOn* идентифицирует любой из следующих объектов ядра: мьютекс, семафор, событие, таймер, процесс, поток, задание, уведомление об изменении файла или консольный ввод. Параметр *dwMilliseconds*, как обычно, определяет, сколько времени функция будет ждать освобождения объекта, а флаг *fAlertable* указывает, сможет ли поток в процессе ожидания обрабатывать посылаемые ему APC-вызовы.

Функция возвращает одно из следующих значений: WAIT_OBJECT_0, WAIT_TIMEOUT, WAIT_FAILED, WAIT_ABANDONED (см. раздел о мьютексах) или WAIT_IO_COMPLETION.

SignalObjectAndWait — удачное добавление к Windows API по двум причинам. Во-первых, освобождение одного объекта и ожидание другого — задача весьма распространенная, а значит, объединение двух операций в одной функции экономит процессорное время. Каждый вызов функции, заставляющей поток переходить из кода, который работает в пользовательском режиме, в код, работающий в режиме ядра, требует примерно 1000 процессорных тактов (на платформах x86), и поэтому для выполнения, например, такого кода:

```
ReleaseMutex(hMutex);
WaitForSingleObject(hEvent, INFINITE);
```

понадобится около 2000 тактов. В высокопроизводительных серверных приложениях *SignalObjectAndWait* дает заметную экономию процессорного времени.

Во-вторых, без функции *SignalObjectAndWait* ни у одного потока не было бы возможности узнать, что другой поток перешел в состояние ожидания. Знание таких вещей очень полезно для функций типа *PulseEvent*. Как я уже говорил в этой главе, *PulseEvent* переводит событие в свободное состояние и тут же сбрасывает его. Если ни один из потоков не ждет данный объект, событие не зафиксирует этот импульс (pulse). Я встречал программистов, которые пишут вот такой код:

```
// выполняем какие-то операции
:
SetEvent(hEventWorkerThreadDone);
WaitForSingleObject(hEventMoreWorkToDo, INFINITE);
// выполняем еще какие-то операции
:
```

Этот фрагмент кода выполняется рабочим потоком, который проделывает какие-то операции, а затем вызывает *SetEvent*, чтобы сообщить (другому потоку) об окончании своих операций. В то же время в другом потоке имеется код:

```
WaitForSingleObject(hEventWorkerThreadDone);
PulseEvent(hEventMoreWorkToDo);
```

Приведенный ранее фрагмент кода рабочего потока порочен по самой своей сути, так как будет работать ненадежно. Ведь вполне вероятно, что после того, как рабочий поток обратится к *SetEvent*, немедленно пробудится другой поток и вызовет *PulseEvent*. Проблема здесь в том, что рабочий поток уже вытеснен и пока еще не получил шанса на возврат из вызова *SetEvent*, не говоря уж о вызове *WaitForSingleObject*. В итоге

ге рабочий поток не сможет своевременно освободить событие *hEventMoreWorkToBeDone*.

Но если Вы перепишете код рабочего потока с использованием функции *SignalObjectAndWait*:

```
// выполняем какие-то операции  
:  
SignalObjectAndWait(hEventWorkerThreadDone,  
    hEventMoreWorkToBeDone, INFINITE, FALSE);  
// выполняем еще какие-то операции  
:
```

то код будет работать надежно, поскольку освобождение и ожидание реализуются на уровне атомарного доступа. И когда пробудится другой поток, Вы сможете быть абсолютно уверены, что рабочий поток ждет события *hEventMoreWorkToBeDone*, а значит, он обязательно заметит импульс, «приложенный» к событию.

WINDOWS 98 В Windows 98 функция *SignalObjectAndWait* определена, но не реализована.

Полезные средства для синхронизации потоков

За годы своей практики я часто сталкивался с проблемами синхронизации потоков и поэтому написал ряд C++-классов и компонентов, которыми я поделюсь с Вами в этой главе. Надеюсь, этот код Вам пригодится и сэкономит массу времени при разработке приложений — или по крайней мере чему-нибудь научит.

Я начну главу с того, что расскажу о реализации критической секции и расширении ее функциональности. В частности, Вы узнаете, как пользоваться одной критической секцией в нескольких процессах. Далее Вы увидите, как сделать объекты безопасными для применения в многопоточной среде, создав для собственных типов данных оболочку из C++-класса. Используя такие классы, я попутно представлю объект, ведущий себя прямо противоположно семафору.

Потом мы рассмотрим одну из типичных задач программирования: что делать, когда считывает какой-то ресурс несколько потоков, а записывает в него — только один. В Windows нет подходящего на этот случай синхронизирующего объекта, и я написал специальный C++-класс.

Наконец, я продемонстрирую свою функцию *WaitForMultipleExpressions*. Работая по аналогии с *WaitForMultipleObjects*, заставляющей ждать освобождения одного или всех объектов, она позволяет указывать более сложные условия пробуждения потока.

Реализация критической секции: объект-оптекс

Критические секции всегда интересовали меня. В конце концов, если это всего лишь объекты пользовательского режима, то почему бы мне не реализовать их самому? Разве нельзя заставить их работать без поддержки операционной системы? Кроме того, написав собственную критическую секцию, я мог бы расширить ее функциональность и в чем-то даже усовершенствовать. По крайней мере я сделал бы так, чтобы она отслеживала, какой поток захватывает защищаемый ею ресурс. Такая реализация критической секции помогла бы мне устраниТЬ проблемы с взаимной блокировкой потоков: с помощью отладчика я узнавал бы, какой из них не освободил тот или иной ресурс.

Так что давайте без лишних разговоров перейдем к тому, как реализуются критические секции. Я все время утверждаю, что они являются объектами пользовательского режима. На самом деле это не совсем так. Любой поток, который пытается войти в критическую секцию, уже захваченную другим потоком, переводится в состояние ожидания. А для этого он должен перейти из пользовательского режима в режим ядра. Поток пользовательского режима может остановиться, просто войдя в цикл ожидания, но это вряд ли можно назвать эффективной реализацией ждущего режима, и поэтому Вы должны всячески избегать ее.

Значит, в критических секциях есть какой-то объект ядра, умеющий переводить поток в эффективный ждущий режим. Критическая секция обладает высоким быстродействием, потому что этот объект ядра используется только при конкуренции потоков за вход в критическую секцию. И он не задействован, пока потоку удается немедленно захватывать защищаемый ресурс, работать с ним и освобождать его без конкуренции со стороны других потоков, так как выходить из пользовательского режима потоку в этом случае не требуется. В большинстве приложений конкуренция двух (или более) потоков за одновременный вход в критическую секцию наблюдается нечасто.

Мой вариант критической секции содержится в файлах Optex.h и Optex.cpp (см. листинг на рис. 10-1). Я назвал ее *оптимизированным мьютексом* — оптексом и реализовал в виде C++-класса. Разобравшись в этом коде, Вы поймете, почему критические секции работают быстрее объектов ядра «мьютекс».

Поскольку я создавал собственную критическую секцию, у меня была возможность расширить ее функциональность. Например, мой класс COptex позволяет синхронизировать потоки из разных процессов. Это фантастически полезная особенность моей реализации: Вы получаете высокоэффективный механизм взаимодействия между потоками из разных процессов.

Чтобы использовать мой оптекс, Вы просто объявляете объект класса COptex. Для этого объекта предусмотрено три конструктора:

```
COptex::(DWORD dwSpinCount = 4000);
COptex::(PCSTR pszName, DWORD dwSpinCount = 4000);
COptex::(PCWSTR pszName, DWORD dwSpinCount = 4000);
```

Первый создает объект COptex, применимый для синхронизации потоков лишь одного процесса. Оптекс этого типа работает быстрее, чем межпроцессный. Остальные два конструктора создают оптекс, которым могут пользоваться потоки из разных процессов. В параметре *pszName* Вы должны передавать ANSI- или Unicode-строку, уникально идентифицирующую каждый разделяемый оптекс. Чтобы процессы разделяли один оптекс, они должны создать по экземпляру объекта COptex с одинаковым именем.

Поток входит в объект COptex и покидает его, вызывая методы *Enter* и *Leave*:

```
void COptex::Enter();
void COptex::Leave();
```

Я даже включил методы, эквивалентные функциям *TryEnterCriticalSection* и *SetCriticalSectionSpinCount* критических секций:

```
BOOL COptex::TryEnter();
void COptex::SetSpinCount(DWORD dwSpinCount);
```

Тип оптекса (одно- или межпроцессный) позволяет выяснить последний метод класса COptex, показанный ниже. (Необходимость в его вызове возникает очень редко, но внутренние функции класса время от времени к нему обращаются.)

```
BOOL COptex::IsSingleProcessOptex() const;
```

Вот и все (открытые) функции, о которых Вам нужно знать, чтобы пользоваться оптексом. Теперь я объясню, как работает оптекс. Он — как, в сущности, и критическая секция — содержит несколько переменных-членов. Значения этих переменных отражают состояние оптекса. Просмотрев файл Optex.h, Вы увидите, что в основном они являются элементами структуры SHAREDINFO, а остальные — членами самого класса. Назначение каждой переменной описывается в следующей таблице.

Переменная	Описание
<i>m_LockCount</i>	Сообщает, сколько раз потоки пытались занять оптекс. Ее значение равно 0, если оптекс не занят ни одним потоком.
<i>m_dwThreadId</i>	Сообщает уникальный идентификатор потока — владельца оптекса. Ее значение равно 0, если оптекс не занят ни одним потоком.
<i>m_lRecurseCount</i>	Указывает, сколько раз оптекс был занят потоком-владельцем. Ее значение равно 0, если оптекс не занят ни одним потоком.
<i>m_bevt</i>	Содержит описатель объекта ядра «событие», используемого, только если поток пытается войти в оптекс в то время, как им владеет другой поток. Описатели объектов ядра специфичны для конкретных процессов, и именно поэтому данная переменная не включена в структуру SHAREDINFO.
<i>m_dwSpinCount</i>	Определяет, сколько попыток входа в оптекс должен предпринять поток до перехода в состояние ожидания на объекте ядра «событие». На однопроцессорной машине значение этой переменной всегда равно 0.
<i>m_bfm</i>	Содержит описатель объекта ядра «проекция файла», используемого при разделении оптекса несколькими процессами. Описатели объектов ядра специфичны для конкретных процессов, и именно поэтому данная переменная не включена в структуру SHAREDINFO. В однопроцессорном оптексе значение этой переменной всегда равно NULL.
<i>m_psi</i>	Содержит указатель на элементы данных оптекса, которые могут использоваться несколькими процессами. Адреса памяти специфичны для конкретных процессов, и именно поэтому данная переменная не включена в структуру SHAREDINFO. В однопроцессорном оптексе эта переменная указывает на блок памяти, выделенный из кучи, а в межпроцессорном — на файл, спроектированный в память.

Комментариев в исходном коде вполне достаточно, и у Вас не должно возникнуть трудностей в понимании того, как работает оптекс. Важно лишь отметить, что высокое быстродействие оптекса достигается за счет интенсивного использования *Interlocked*-функций. Благодаря им код выполняется в пользовательском режиме и переходит в режим ядра только в том случае, когда это действительно необходимо.

Программа-пример Optex

Эта программа, «10 Optex.exe» (см. листинг на рис. 10-1), предназначена для проверки того, что класс COptex работает корректно. Файлы исходного кода и ресурсов этой программы находятся в каталоге 10-Optex на компакт-диске, прилагаемом к книге. Я всегда запускаю такие приложения под управлением отладчика, чтобы наблюдать за всеми функциями и переменными — членами классов.

При запуске программа сначала определяет, является ли она первым экземпляром. Для этого я создаю именованный объект ядра «событие». Реально я им не пользуюсь, а просто смотрю, вернет ли *GetLastError* значение *ERROR_ALREADY_EXISTS*. Если да, значит, это второй экземпляр программы. Зачем мне два экземпляра этой программы, я объясню позже.

Если же это первый экземпляр, я создаю однопроцессный объект COptex и вызываю свою функцию *FirstFunc*. Она выполняет серию операций с объектом-оптексом и создает второй поток, который манипулирует тем же оптексом. На этом этапе с оптексом работают два потока из одного процесса. Что именно они делают, Вы узнаете, просмотрев исходный код. Я пытался охватить все мыслимые сценарии, чтобы дать шанс на выполнение каждому блоку кода в классе COptex.

После тестирования однопроцессного оптекса я начинаю проверку межпроцессного оптекса. В функции *_tWinMain* по завершении первого вызова *FirstFunc* я создаю

другой объект-оптекс COptex. Но на этот раз я присваиваю ему имя — *CrossOptexTest*. Простое присвоение оптексу имени в момент создания превращает этот объект в межпроцессный. Далее я снова вызываю *FirstFunc*, передавая ей адрес межпроцессного оптекса. При этом *FirstFunc* выполняет в основном тот же код, что и раньше. Но теперь она порождает не второй поток, а дочерний процесс.

Этот дочерний процесс представляет собой всего лишь второй экземпляр той же программы. Однако, создав при запуске объект ядра «событие», она обнаруживает, что такой объект уже существует. Тем самым она узнает, что является вторым экземпляром, и выполняет другой код (отличный от того, который выполняется первым экземпляром). Первое, что делает второй экземпляр, — вызывает *DebugBreak*:

```
VOID DebugBreak();
```

Эта удобная функция инициирует запуск отладчика и его подключение к данному процессу. Это здорово упрощает мне отладку обоих экземпляров данной программы. Далее второй экземпляр создает межпроцессный оптекс, передавая конструктору строку с тем же именем. Поскольку имена идентичны, оптекс становится разделяемым между обоими процессами. Кстати, один оптекс могут разделять более двух процессов.

Наконец, второй экземпляр программы вызывает функцию *SecondFunc*, передавая ей адрес межпроцессного оптекса, и с этого момента выполняется тот же набор тестов. Единственное, что в них меняется, — два потока, манипулирующие оптексом, принадлежат разным процессам.



Optex.cpp

```
*****  
Модуль: Optex.cpp  
Автор: Copyright (c) 2000, Джейфри Рихтер (Jeffrey Richter)  
*****  
  
#include "..\CmnHdr.h"      /* см. приложение A */  
#include "Optex.h"  
  
//////////  
  
// 0=многопроцессорная машина, 1=однопроцессорная, -1=тип пока не определен  
BOOL COptex::sm_fUniprocessorHost = -1;  
  
//////////  
  
PSTR COptex::ConstructObjectName(PSTR pszResult,  
    PCSTR pszPrefix, BOOL fUnicode, PVOID pszName) {  
  
    pszResult[0] = 0;  
    if (pszName == NULL)  
        return(NULL);  
  
    wsprintf(pszResult, fUnicode ? "%s%S" : "%s%s", pszPrefix, pszName);  
    return(pszResult);  
}
```

Рис. 10-1. Программа-пример Optex

Рис. 10-1. продолжение

```
//////////  

void COptex::CommonConstructor(DWORD dwSpinCount,
    BOOL fUnicode, PVOID pszName) {  

    if (sm_fUniprocessorHost == -1) {  

        // конструируется первый объект; выясняем количество процессоров  

        SYSTEM_INFO sinf;  

        GetSystemInfo(&sinf);  

        sm_fUniprocessorHost = (sinf.dwNumberOfProcessors == 1);  

    }  

    m_hevt = m_hfm = NULL;  

    m_psi = NULL;  

    if (pszName == NULL) { // создание однопроцессного оптекса  

        m_hevt = CreateEventA(NULL, FALSE, FALSE, NULL);  

        chASSERT(m_hevt != NULL);  

        m_psi = new SHAREDINFO;  

        chASSERT(m_psi != NULL);  

        ZeroMemory(m_psi, sizeof(*m_psi));  

    } else { // создание межпроцессного оптекса  

        // всегда используем ANSI, чтобы программа работала в Win9x и Windows 2000  

        char szResult[100];  

        ConstructObjectName(szResult, "Optex_Event_", fUnicode, pszName);  

        m_hevt = CreateEventA(NULL, FALSE, FALSE, szResult);  

        chASSERT(m_hevt != NULL);  

        ConstructObjectName(szResult, "Optex_MMF_", fUnicode, pszName);  

        m_hfm = CreateFileMappingA(INVALID_HANDLE_VALUE, NULL,  

            PAGE_READWRITE, 0, sizeof(*m_psi), szResult);  

        chASSERT(m_hfm != NULL);  

        m_psi = (PSHAREDINFO) MapViewOfFile(m_hfm,  

            FILE_MAP_WRITE, 0, 0, 0);  

        chASSERT(m_psi != NULL);  

        // Примечание: элементы m_lLockCount, m_dwThreadId и m_lRecurseCount  

        // структуры SHAREDINFO надо инициализировать нулевым значением. К счастью,  

        // механизм проецирования файлов в память избавляет нас от части работы,  

        // связанной с синхронизацией потоков.  

    }  

    SetSpinCount(dwSpinCount);  

}  

//////////
```

см. след. стр.

Рис. 10-1. продолжение

```
COptex::~COptex() {

#ifndef _DEBUG
    if (IsSingleProcessOptex() && (m_psi->m_dwThreadId != 0)) {
        // однопроцессный оптекс нельзя разрушать,
        // если им владеет какой-нибудь поток
        DebugBreak();
    }

    if (!IsSingleProcessOptex() &&
        (m_psi->m_dwThreadId == GetCurrentThreadId())) {

        // межпроцессный оптекс нельзя разрушать, если им владеет наш поток
        DebugBreak();
    }
#endif

    CloseHandle(m_hevt);

    if (IsSingleProcessOptex()) {
        delete m_psi;
    } else {
        UnmapViewOfFile(m_psi);
        CloseHandle(m_hfm);
    }
}

//////////void COptex::SetSpinCount(DWORD dwSpinCount) {

    // спин-блокировка на однопроцессорных машинах не применяется
    if (!sm_fUniprocessorHost)
        InterlockedExchangePointer((PVOID*) &m_psi->m_dwSpinCount,
                                    (VOID) (DWORD_PTR) dwSpinCount);
}

//////////void COptex::Enter() {

    // "крутимся", пытаясь захватить оптекс
    if (TryEnter())
        return; // получилось, возвращаем управление

    // захватить оптекс не удалось, переходим в состояние ожидания
    DWORD dwThreadId = GetCurrentThreadId();

    if (InterlockedIncrement(&m_psi->m_lLockCount) == 1) {
        // оптекс не занят, пусть этот поток захватит его разок
        m_psi->m_dwThreadId = dwThreadId;
```

Рис. 10-1. продолжение

```

m_psi->m_lRecurseCount = 1;

} else {

    if (m_psi->m_dwThreadId == dwThreadId) {

        // если оптекс принадлежит данному потоку, захватываем его еще раз
        m_psi->m_lRecurseCount++;

    } else {

        // оптекс принадлежит другому потоку, ждем
        WaitForSingleObject(m_hevt, INFINITE);

        // оптекс не занят, пусть этот поток захватит его разок
        m_psi->m_dwThreadId = dwThreadId;
        m_psi->m_lRecurseCount = 1;
    }
}

///////////////////////////////
BOOL COptex::TryEnter() {

    DWORD dwThreadId = GetCurrentThreadId();

    BOOL fThisThreadOwnsTheOptex = FALSE;      // считаем, что поток владеет оптексом
    DWORD dwSpinCount = m_psi->m_dwSpinCount; // задаем число циклов

    do {
        // если счетчик числа блокировок = 0, оптекс не занят,
        // и мы можем захватить его
        fThisThreadOwnsTheOptex = (0 ==
            InterlockedCompareExchange(&m_psi->m_lLockCount, 1, 0));

        if (fThisThreadOwnsTheOptex) {

            // оптекс не занят, пусть этот поток захватит его разок
            m_psi->m_dwThreadId = dwThreadId;
            m_psi->m_lRecurseCount = 1;

        } else {

            if (m_psi->m_dwThreadId == dwThreadId) {

                // если оптекс принадлежит данному потоку, захватываем его еще раз
                InterlockedIncrement(&m_psi->m_lLockCount);
                m_psi->m_lRecurseCount++;
                fThisThreadOwnsTheOptex = TRUE;
            }
        }
    }
}

```

см. след. стр.

Рис. 10-1. продолжение

```
        }

    }

} while (!fThisThreadOwnsTheOptex && (dwSpinCount-- > 0));

// возвращаем управление независимо от того,
// владеет данный поток оптексом или нет
return(fThisThreadOwnsTheOptex);
}

/////////// Конец файла ///////////



void COptex::Leave() {

#ifndef _DEBUG
// покинуть оптекс может лишь тот поток, который им владеет
if (m_psi->m_dwThreadId != GetCurrentThreadId())
    DebugBreak();
#endif

// уменьшаем счетчик числа захватов оптекса данным потоком
if (--m_psi->m_lRecurseCount > 0) {

    // оптекс все еще принадлежит нам
    InterlockedDecrement(&m_psi->m_lLockCount);

} else {

    // оптекс нам больше не принадлежит
    m_psi->m_dwThreadId = 0;

    if (InterlockedDecrement(&m_psi->m_lLockCount) > 0) {

        // если оптекс ждут другие потоки,
        // событие с автосбросом пробудит один из них
        SetEvent(m_hevt);
    }
}
}

/////////// Конец файла ///////////
```

Optex.h

```
*****
Имя модуля: Optex.h
Написан: Джейфри Рихтером
*****/
#pragma once
```

Рис. 10-1. продолжение

```
//////////  

class COptex {  

public:  

    COptex(DWORD dwSpinCount = 4000);  

    COptex(PCSTR pszName, DWORD dwSpinCount = 4000);  

    COptex(PCWSTR pszName, DWORD dwSpinCount = 4000);  

    ~COptex();  

    void SetSpinCount(DWORD dwSpinCount);  

    void Enter();  

    BOOL TryEnter();  

    void Leave();  

    BOOL IsSingleProcessOptex() const;  

private:  

    typedef struct {  

        DWORD m_dwSpinCount;  

        long m_lLockCount;  

        DWORD m_dwThreadId;  

        long m_lRecurseCount;  

    } SHAREDINFO, *PSHAREDINFO;  

    HANDLE m_hevt;  

    HANDLE m_hfm;  

    PSHAREDINFO m_psi;  

private:  

    static BOOL sm_fUniprocessorHost;  

private:  

    void CommonConstructor(DWORD dwSpinCount, BOOL fUnicode, PVOID pszName);  

    PSTR ConstructObjectName(PSTR pszResult,  

                           PCSTR pszPrefix, BOOL fUnicode, PVOID pszName);  

};  

//////////  

inline COptex::COptex(DWORD dwSpinCount) {  

    CommonConstructor(dwSpinCount, FALSE, NULL);  

}  

//////////  

inline COptex::COptex(PCSTR pszName, DWORD dwSpinCount) {  

    CommonConstructor(dwSpinCount, FALSE, (PVOID) pszName);  

}  

//////////
```

см. след. стр.

Рис. 10-1. продолжение

```
inline COptex::COptex(PCWSTR pszName, DWORD dwSpinCount) {  
    CommonConstructor(dwSpinCount, TRUE, (PVOID) pszName);  
}  
/////////////////////////////////////////////////////////////////////////  
inline COptex::IsSingleProcessOptex() const {  
    return(m_hfm == NULL);  
}  
//////////////////////////////////////////////////////////////////////// Конец файла //////////////////////////////////////////////////////////////////
```

OptexTest.cpp

```
*****  
Имя модуля: OptexTest.cpp  
Написан: Джейфри Рихтером  
*****/  
  
#include "..\CmnHdr.h"      /* см. приложение A */  
#include <tchar.h>  
#include <process.h>  
#include "Optex.h"  
  
/////////////////////////////////////////////////////////////////////////  
  
DWORD WINAPI SecondFunc(PVOID pvParam) {  
  
    COptex& optex = * (COptex*) pvParam;  
  
    // здесь оптексом должен владеть первичный поток,  
    // и выполнение этого оператора должно закончиться неудачно  
    chVERIFY(optex.TryEnter() == FALSE);  
  
    // ждем, когда первичный поток откажется от оптекса  
    optex.Enter();  
  
    optex.Enter(); // проверяем "рекурсию захватов"  
    chMB("Secondary: Entered the optex\n(Dissmiss me 2nd)");  
  
    // покидаем оптекс, но он все еще принадлежит нам  
    optex.Leave();  
    chMB("Secondary: The primary thread should not display a box yet");  
    optex.Leave(); // теперь первичный поток может работать  
  
    return(0);  
}  
/////////////////////////////////////////////////////////////////////////
```

Рис. 10-1. продолжение

```

VOID FirstFunc(BOOL fLocal, COptex& optex) {

    optex.Enter(); // захватываем оптекс

    // поскольку этот поток уже владеет оптексом, мы сможем заполучить его еще раз
    chVERIFY(optex.TryEnter());

    HANDLE hOtherThread = NULL;
    if (fLocal) {
        // порождаем вторичный поток просто для тестирования (передаем ему оптекс)

        DWORD dwThreadId;
        hOtherThread = chBEGINTHREADEX(NULL, 0,
            SecondFunc, (PVOID) &optex, 0, &dwThreadId);

    } else {
        // порождаем вторичный процесс просто для тестирования
        STARTUPINFO si = { sizeof(si) };
        PROCESS_INFORMATION pi;
        TCHAR szPath[MAX_PATH];
        GetModuleFileName(NULL, szPath, chDIMOF(szPath));
        CreateProcess(NULL, szPath, NULL, NULL,
            FALSE, 0, NULL, NULL, &si, &pi);
        hOtherThread = pi.hProcess;
        CloseHandle(pi.hThread);
    }

    // ждем, когда второй поток захватит оптекс
    chMB("Primary: Hit OK to give optex to secondary");

    // разрешаем второму потоку захватить оптекс
    optex.Leave();
    optex.Leave();

    // ждем, когда второй поток захватит оптекс
    chMB("Primary: Hit OK to wait for the optex\n(Dissmiss me 1st)");

    optex.Enter(); // пытаемся вновь захватить оптекс

    WaitForSingleObject(hOtherThread, INFINITE);
    CloseHandle(hOtherThread);
    optex.Leave();
}

///////////////////////////////
int WINAPI _tWinMain(HINSTANCE hinstExe, HINSTANCE, PTSTR pszCmdLine, int) {

    // это событие используется только для того, чтобы
    // определить, какой это экземпляр – первый или второй
    HANDLE hevt = CreateEvent(NULL, FALSE, FALSE, TEXT("OptexTest"));

```

см. след. стр.

Рис. 10-1. *продолжение*

```
if (GetLastError() != ERROR_ALREADY_EXISTS) {  
  
    // это первый экземпляр тестовой программы  
  
    // сначала тестируем однопроцессный оптекс  
    COptex optexSingle; // создаем однопроцессный оптекс  
    FirstFunc(TRUE, optexSingle);  
  
    // теперь тестируем межпроцессный оптекс  
    COptex optexCross("CrossOptexTest"); // создаем межпроцессный оптекс  
    FirstFunc(FALSE, optexCross);  
  
} else {  
  
    // это второй экземпляр тестовой программы  
    DebugBreak(); // принудительно подключаем отладчик для трассировки  
  
    // тестируем межпроцессный оптекс  
    COptex optexCross("CrossOptexTest"); // создаем межпроцессный оптекс  
    SecondFunc((PVOID) &optexCross);  
}  
  
CloseHandle(hevt);  
return(0);  
}  
  
/////////////////////////////// Конец файла ///////////////////////////////
```

Создание инверсных семафоров и типов данных, безопасных в многопоточной среде

Как-то раз я писал одну программу, и мне понадобился объект ядра, который вел бы себя прямо противоположно тому, как ведет себя семафор. Мне нужно было, чтобы он переходил в свободное состояние, когда его счетчик текущего числа ресурсов обнуляется, и оставался в занятом состоянии, пока значение этого счетчика больше 0.

Я мог бы придумать много применений такому объекту. Например, поток должен пробудиться после того, как определенная операция будет выполнена 100 раз. Чтобы осуществить это, нужен объект ядра, счетчик которого можно было бы инициализировать этим значением. Пока он больше 0, объект остается в занятом состоянии. По окончании каждой операции Вы уменьшаете счетчик в объекте ядра на 1. Как только счетчик обнуляется, объект переходит в свободное состояние, сообщая другому потоку, что тот может пробудиться и чем-то заняться. Это типичная задача, и я не понимаю, почему в Windows нет подходящего синхронизирующего объекта.

В сущности, Microsoft могла бы легко решить эту задачу, предусмотрев в семафоре возможность присвоения отрицательных значений его счетчику текущего числа ресурсов. Тогда Вы инициализировали бы счетчик семафора значением -99 и по окончании каждой операции вызывали бы *ReleaseSemaphore*. Как только его счетчик достиг бы значения 1, объект перешел бы в свободное состояние. После этого мог бы пробудиться другой Ваш поток и выполнить свою работу. Увы, Microsoft запрещает

присвоение счетчику семафора отрицательных значений, и вряд ли здесь что-то переменится в обозримом будущем.

В этом разделе я познакомлю Вас со своим набором C++-классов, которые действуют как инверсный семафор и делают уйму всяких других вещей. Исходный код этих классов находится в файле Interlocked.h (см. листинг на рис. 10-2).

Когда я впервые взялся за решение этой проблемы, я понимал, что главное в нем — обеспечить безопасность манипуляций над переменной в многопоточной среде. Я хотел найти элегантное решение, которое позволило бы легко писать код, ссылающийся на эту переменную. Очевидно, что самый простой способ обезопасить какой-то ресурс от повреждения в многопоточной среде, — защитить его с помощью критической секции. В C++ это можно сделать без особого труда. Достаточно создать C++-класс, который содержит защищаемую переменную и структуру CRITICAL_SECTION. В конструкторе Вы вызываете *InitializeCriticalSection*, а в деструкторе — *DeleteCriticalSection*. Затем для любой переменной-члена Вы вызываете *EnterCriticalSection*, что-то делаете с этой переменной и вызываете *LeaveCriticalSection*. Если Вы именно так реализуете C++-класс, то писать безопасный код, обращающийся к какой-либо структуре данных, будет несложно. Этот принципложен мной в основу всех C++-классов, о которых я буду рассказывать в данном разделе. (Конечно, вместо критических секций я мог бы использовать оптекс, рассмотренный в предыдущем разделе.)

Первый класс, CResGuard, охраняет доступ к ресурсу. Он содержит два элемента данных: CRITICAL_SECTION и LONG. Последний используется для слежения за тем, сколько раз поток, владеющий ресурсом, входил в критическую секцию. Эта информация полезна при отладке. Конструктор и деструктор объекта CResGuard вызывают соответственно *InitializeCriticalSection* и *DeleteCriticalSection*. Поскольку создать объект может лишь единственный поток, конструктор и деструктор какого-либо C++-объекта не обязательно должен быть реентерабельным. Функция-член *IsGuarded* просто сообщает, была ли хоть раз вызвана *EnterCriticalSection* для данного объекта. Как я уже говорил, все это предназначено для отладки. Включение CRITICAL_SECTION в C++-объект гарантирует корректность инициализации и удаления критической секции.

Класс CResGuard также включает открытый вложенный C++-класс CGuard. Объект CGuard содержит ссылку на объект CResGuard и предусматривает лишь конструктор и деструктор. Конструктор обращается к функции-члену *Guard* класса CResGuard, вызывающей *EnterCriticalSection*, а деструктор — к функции-члену *Unguard* того же класса, вызывающей *LeaveCriticalSection*. Такая схема упрощает манипуляции с CRITICAL_SECTION. Вот небольшой фрагмент кода, иллюстрирующий применение этих классов:

```
struct SomeDataStruct {
    :
} g_SomeSharedData;

// Создаем объект CResGuard, защищающий g_SomeSharedData.
// Примечание: Конструктор инициализирует критическую секцию, а деструктор удаляет ее.
CResGuard g_rgSomeSharedData;

void AFunction() {
    // эта функция работает с разделяемой структурой данных

    // защищаем ресурс от одновременного доступа со стороны нескольких потоков
    CResGuard::CGuard gDummy(g_rgSomeSharedData); // входим в критическую секцию
```

см. след. стр.

```
// работаем с ресурсом g_SomeSharedData  
:  
} // Примечание: LeaveCriticalSection вызывается, когда gDummy  
// выходит за пределы области видимости.
```

Следующий C++-класс, CInterlockedType, содержит все, что нужно для создания объекта данных, безопасного в многопоточной среде. Я сделал CInterlockedType классом шаблона, чтобы его можно было применять для любых типов данных. Поэтому Вы можете использовать его, например, с целочисленной переменной, строкой или произвольной структурой данных.

Каждый экземпляр объекта CInterlockedType содержит два элемента данных. Первый — это экземпляр шаблонного типа данных, который Вы хотите сделать безопасным в многопоточной среде. Он является закрытым, и им можно манипулировать только через функции-члены класса CInterlockedType. Второй элемент данных представляет собой экземпляр объекта CResGuard, так что класс, производный от CInterlockedType, может легко защитить свои данные.

Предполагается, что Вы всегда будете создавать свой класс, используя класс CInterlockedType как базовый. Ранее я уже говорил, что класс CInterlockedType предоставляет все необходимое для создания объекта, безопасного в многопоточной среде, но производный класс должен сам позаботиться о корректном использовании элементов CInterlockedType.

Класс CInterlockedType содержит всего четыре открытые функции: конструктор, инициализирующий объект данных, и конструктор, не инициализирующий этот объект, а также виртуальный деструктор, который ничего не делает, и оператор приведения типа (cast operator). Последний просто гарантирует безопасный доступ к данным, охраняя ресурс и возвращая текущее значение объекта. (Ресурс автоматически разблокируется при выходе локальной переменной *x* за пределы ее области видимости.) Этот оператор упрощает безопасную проверку значения объекта данных, содержащегося в классе.

В классе CInterlockedType также присутствуют три невиртуальные защищенные функции, которые будут вызываться производным классом. Две функции *GetVal* возвращают текущее значение объекта данных. В отладочных версиях файла обе эти функции сначала проверяют, охраняется ли объект данных. Если бы он не охранялся, *GetVal* могла бы вернуть значение объекта, а затем позволить другому потоку изменить его до того, как первый поток успеет что-то сделать с этим значением. Я предполагаю, что вызывающий поток получает значение объекта для того, чтобы как-то изменить его. Поэтому функции *GetVal* требуют от вызывающего потока охраны доступа к данным. Определив, что данные охраняются, функции *GetVal* возвращают текущее значение.

Эти функции идентичны с тем исключением, что одна из них манипулирует константной версией объекта. Благодаря этому Вы можете без проблем писать код, работающий как с константными, так и с неконстантными данными.

Третья невиртуальная защищенная функция-член — *SetVal*. Желая модифицировать данные, любая функция-член производного класса должна защитить доступ к этим данным, а потом вызвать функцию *SetVal*. Как и *GetVal*, функция *SetVal* сначала проводит отладочную проверку, чтобы убедиться, не забыл ли код производного класса защитить доступ к данным. Затем *SetVal* проверяет, действительно ли данные изменяются. Если да, *SetVal* сохраняет старое значение, присваивает объекту новое значение и вызывает виртуальную защищенную функцию-член *OnValChanged*, передавая ей оба значения. В классе CInterlockedType последняя функция реализована так, что она

ничего не делает. Вы можете использовать эту функцию-член для того, чтобы расширить возможности своего производного класса, но об этом мы поговорим, когда дойдем до рассмотрения класса CWhenZero.

До сих пор речь шла в основном об абстрактных классах и концепциях. Теперь посмотрим, как пользоваться этой архитектурой на благо всего человечества. Я представлю Вам CInterlockedScalar — класс шаблона, производный от CInterlockedType. С его помощью Вы сможете создавать безопасные в многопоточной среде скалярные (простые) типы данных — байт, символ, 16-, 32- или 64-битное целое, вещественное значение (с плавающей точкой) и т. д. Поскольку CInterlockedScalar является производным от класса CInterlockedType, у него нет собственных элементов данных. Конструктор CInterlockedScalar просто обращается к конструктору CInterlockedType, передавая ему начальное значение объекта скалярных данных. Класс CInterlockedScalar работает только с числовыми значениями, и в качестве начального значения я выбрал нуль, чтобы наш объект всегда создавался в известном состоянии. Ну а деструктор класса CInterlockedScalar вообще ничего не делает.

Остальные функции-члены класса CInterlockedScalar отвечают за изменение скалярного значения. Для каждой операции над ним предусмотрена отдельная функция-член. Чтобы класс CInterlockedScalar мог безопасно манипулировать своим объектом данных, все функции-члены перед выполнением какой-либо операции блокируют доступ к этому объекту. Функции очень просты, и я не стану подробно объяснять их; просмотрев исходный код, Вы сами поймете, что они делают. Однако я покажу, как пользоваться этими классами. В следующем фрагменте кода объявляется безопасная в многопоточной среде переменная типа BYTE и над ней выполняется серия операций:

```
CInterlockedScalar<BYTE> b = 5; // безопасная переменная типа BYTE
BYTE b2 = 10; // небезопасная переменная типа BYTE
b2 = b++;
b2 *= 4;
b2 = b;
b += b;
b %= 2;
```

Работа с безопасной скалярной переменной так же проста, как и с небезопасной. Благодаря замещению (перегрузке) операторов в C++ даже код в таких случаях фактически одинаков! С помощью C++-классов, о которых я уже рассказал, любую небезопасную переменную можно легко превратить в безопасную, внеся лишь минимальные изменения в исходный код своей программы.

Проектируя все эти классы, я хотел создать объект, чье поведение было бы противоположно поведению семафора. Эту функциональность предоставляет мой C++-класс CWhenZero, производный от CInterlockedScalar. Когда скалярное значение равно 0, объект CWhenZero пребывает в свободном состоянии, а когда оно не равно 0 — в занятом.

Как Вам известно, C++-объекты не поддерживают такие состояния — в них могут находиться только объекты ядра. Значит, в CWhenZero нужны дополнительные элементы данных с описателями объектов ядра «событие». Я включил в объект CWhenZero два элемента данных: *m_beveZero* (описатель объекта ядра «событие», переходящего в свободное состояние, когда объект данных содержит нулевое значение) и *m_beveNotZero* (описатель объекта ядра «событие», переходящего в свободное состояние, когда объект данных содержит ненулевое значение).

Конструктор CWhenZero принимает начальное значение для объекта данных, а также позволяет указать, какими должны быть объекты ядра «событие» — со сбросом

вручную (по умолчанию) или с автосбросом. Далее конструктор, вызывая *CreateEvent*, создает два объекта ядра «событие» и переводит их в свободное или занятое состояние в зависимости от того, равно ли нулю начальное значение. Деструктор *CWhenZero* просто закрывает описатели этих двух объектов ядра. Поскольку *CWhenZero* открыто наследует от класса *CIinterlockedScalar*, все функции-члены перегруженного оператора доступны и пользователям объекта *CWhenZero*.

Помните защищенную функцию-член *OnValChanged*, объявленную внутри класса *CIinterlockedType*? Так вот, класс *CWhenZero* замещает эту виртуальную функцию. Она отвечает за перевод объектов ядра «событие» в свободное или занятое состояние в соответствии со значением объекта данных. *OnValChanged* вызывается при каждом изменении этого значения. Ее реализация в *CWhenZero* проверяет, равно ли нулю новое значение. Если да, функция устанавливает событие *m_hevtZero* и сбрасывает событие *m_hevtNotZero*. Нет — все делается наоборот.

Теперь, если Вы хотите, чтобы поток ждал нулевого значения объекта данных, от Вас требуется лишь следующее:

```
CWhenZero<BYTE> b = 0; // безопасная переменная типа BYTE

// немедленно возвращает управление, так как b равна 0
WaitForSingleObject(b, INFINITE);

b = 5;

// возвращает управление, только если другой поток присваивает b нулевое значение
WaitForSingleObject(b, INFINITE);
```

Вы можете вызывать *WaitForSingleObject* именно таким образом, потому что класс *CWhenZero* включает и функцию-член оператора приведения, которая приводит объект *CWhenZero* к типу HANDLE объекта ядра. Иначе говоря, передача C++-объекта *CWhenZero* любой Windows-функции, ожидающей HANDLE, приводит к автоматическому вызову функции-члена оператора приведения, возвращаемое значение которой и передается Windows-функции. В данном случае эта функция-член возвращает описатель объекта ядра «событие» *m_hevtZero*.

Описатель события *m_hevtNotZero* внутри класса *CWhenZero* позволяет писать код, ждущий ненулевого значения объекта данных. К сожалению, в класс нельзя включить второй оператор приведения HANDLE — для получения описателя *m_hevtNotZero*. Поэтому мне пришлось добавить функцию-член *GetNotZeroHandle*, которая используется так:

```
CWhenZero<BYTE> b = 5; // безопасная переменная типа BYTE

// немедленно возвращает управление, так как b не равна 0
WaitForSingleObject(b.GetNotZeroHandle(), INFINITE);

b = 0;

// возвращает управление, только если другой поток присваивает b ненулевое значение
WaitForSingleObject(b.GetNotZeroHandle(), INFINITE);
```

Программа-пример *InterlockedType*

Эта программа, «10 InterlockedType.exe» (см. листинг на рис. 10-2), предназначена для тестирования только что описанных классов. Файлы исходного кода и ресурсов этой

программы находятся в каталоге 10-InterlockedType на компакт-диске, прилагаемом к книге. Как я уже говорил, такие приложения я всегда запускаю под управлением отладчика, чтобы наблюдать за всеми функциями и переменными — членами классов.

Программа иллюстрирует типичный сценарий программирования, который выглядит так. Поток порождает несколько рабочих потоков, а затем инициализирует блок памяти. Далее основной поток пробуждает рабочие потоки, чтобы они начали обработку содержимого этого блока памяти. В данный момент основной поток должен приостановить себя до тех пор, пока все рабочие потоки не выполнят свои задачи. После этого основной поток записывает в блок памяти новые данные и вновь пробуждает рабочие потоки.

На примере этого кода хорошо видно, насколько тривиальным становится решение этой распространенной задачи программирования при использовании C++. Класс CWhenZero дает нам гораздо больше возможностей — не один лишь инверсный семафор. Мы получаем теперь безопасный в многопоточной среде объект данных, который переходит в свободное состояние, когда его значение обнуляется! Вы можете не только увеличивать и уменьшать счетчик семафора на 1, но и выполнять над ним любые математические и логические операции, в том числе сложение, вычитание, умножение, деление, вычисления по модулю! Так что объект CWhenZero намного функциональнее, чем объект ядра «семафор».

С этими классами шаблонов C++ можно много чего придумать. Например, создать класс CInterlockedString, производный от CInterlockedType, и с его помощью безопасно манипулировать символьными строками. А потом создать класс CWhenCertainString, производный от CInterlockedString, чтобы освобождать объект ядра «событие», когда строка принимает определенное значение (или значения). В общем, возможности безграничны.



IntLockTest.cpp

```

/*****
Modуль: IntLockTest.cpp
Автор: Copyright (c) 2000, Джейфри Рихтер (Jeffrey Richter)
*****/

#include "..\CmnHdr.h"      /* см. приложение A */
#include <tchar.h>
#include "Interlocked.h"

///////////////////////////////
// присваиваем TRUE, когда рабочие потоки должны завершиться
volatile BOOL g_fQuit = FALSE;
///////////////////////////////

DWORD WINAPI WorkerThread(PVOID pvParam) {
    CWhenZero<BYTE>& bVal = * (CWhenZero<BYTE> *) pvParam;

    // должен ли рабочий поток завершиться?
    while (!g_fQuit) {

```

Рис. 10-2. Программа-пример *InterlockedType*

см. след. стр.

Рис. 10-2. продолжение

```
// ждем какой-нибудь работы
WaitForSingleObject(bVal.GetNotZeroHandle(), INFINITE);

// если мы должны выйти - выходим
if (g_fQuit)
    continue;

// что-то делаем
chMB("Worker thread: We have something to do");

bVal--; // сделали

// ждем, когда остановятся все рабочие потоки
WaitForSingleObject(bVal, INFINITE);
}

chMB("Worker thread: terminating");
return(0);
}

///////////////////////////////
int WINAPI _tWinMain(HINSTANCE hinstExe, HINSTANCE, PTSTR pszCmdLine, int) {

// инициализируем объект так, чтобы указать:
// ни одному рабочему потоку делать нечего
CWhenZero<BYTE> bVal = 0;

// создаем рабочие потоки
const int nMaxThreads = 2;
HANDLE hThreads[nMaxThreads];
for (int nThread = 0; nThread < nMaxThreads; nThread++) {
    DWORD dwThreadId;
    hThreads[nThread] = CreateThread(NULL, 0,
        WorkerThread, (PVOID) &bVal, 0, &dwThreadId);
}

int n;
do {
    // делать что-то еще или остановиться?
    n = MessageBox(NULL,
        TEXT("Yes: Give worker threads something to do\nNo: Quit"),
        TEXT("Primary thread"), MB_YESNO);

    // сообщаем рабочим потокам, что мы выходим
    if (n == IDNO)
        InterlockedExchangePointer((PVOID*) &g_fQuit, (PVOID) TRUE);

    bVal = nMaxThreads; // пробуждаем рабочие потоки

    if (n == IDYES) {
```

Рис. 10-2. продолжение

```

    // есть работа, ждем, когда рабочие потоки ее сделают
    WaitForSingleObject(bVal, INFINITE);
}

} while (n == IDYES);

// работы больше нет, процесс надо завершить;
// ждем завершения рабочих потоков
WaitForMultipleObjects(nMaxThreads, hThreads, TRUE, INFINITE);

// закрываем описатели рабочих потоков
for (nThread = 0; nThread < nMaxThreads; nThread++)
    CloseHandle(hThreads[nThread]);

// сообщаем пользователю, что процесс завершается
chMB("Primary thread: terminating");

return(0);
}

//////////////////////////// Конец файла //////////////////////////////

```

Interlocked.h

```

*****
Модуль: Interlocked.h
Автор: Copyright (c) 2000, Джейфри Рихтер (Jeffrey Richter)
*****/

#pragma once

////////////////////////////

// к экземплярам этого класса будет обращаться множество потоков, поэтому
// все его члены (кроме конструктора и деструктора) должны быть безопасны
// в многопоточной среде
class CResGuard {
public:
    CResGuard() { m_lGrdCnt = 0; InitializeCriticalSection(&m_cs); }
    ~CResGuard() { DeleteCriticalSection(&m_cs); }

    // IsGuarded используется для отладки
    BOOL IsGuarded() const { return(m_lGrdCnt > 0); }

public:
    class CGuard {
    public:
        CGuard(CResGuard& rg) : m_rg(rg) { m_rg.Guard(); }
        ~CGuard() { m_rg.Unguard(); }
    };
};


```

см. след. стр.

Рис. 10-2. продолжение

```
private:  
    CResGuard& m_rg;  
};  
  
private:  
void Guard() { EnterCriticalSection(&m_cs); m_lGrdCnt++; }  
void Unguard() { m_lGrdCnt--; LeaveCriticalSection(&m_cs); }  
  
// к Guard/Unguard может обращаться только вложенный класс CGuard  
friend class CResGuard::CGuard;  
  
private:  
    CRITICAL_SECTION m_cs;  
    long m_lGrdCnt; // число вызовов EnterCriticalSection  
};  
  
/////////////////////////////////////////////////////////////////////////  
  
// к экземплярам этого класса будет обращаться множество потоков, поэтому  
// все его члены (кроме конструктора и деструктора) должны быть безопасны  
// в многопоточной среде  
template <class TYPE>  
class CInterlockedType {  
  
public: // открытые функции-члены  
    // Примечание: конструкторы и деструкторы всегда безопасны  
    // в многопоточной среде.  
    CInterlockedType() {}  
    CInterlockedType(const TYPE& TVal) { m_TVal = TVal; }  
    virtual ~CInterlockedType() {}  
  
    // оператор приведения, который упрощает написание кода с использованием  
    // безопасного в многопоточной среде типа данных  
    operator TYPE() const {  
        CResGuard::CGuard x(m_rg);  
        return(GetVal());  
    }  
  
protected: // защищенная функция, которую должен вызывать производный класс  
    TYPE& GetVal() {  
        chASSERT(m_rg.IsGuarded());  
        return(m_TVal);  
    }  
  
    const TYPE& GetVal() const {  
        assert(m_rg.IsGuarded());  
        return(m_TVal);  
    }  
  
    TYPE SetVal(const TYPE& TNewVal) {  
        chASSERT(m_rg.IsGuarded());
```

Рис. 10-2. продолжение

```

TYPE& TVal = GetVal();
if (TVal != TNewVal) {
    TYPE TPrevVal = TVal;
    TVal = TNewVal;
    OnValChanged(TNewVal, TPrevVal);
}
return(TVal);

protected: // замещаемые функции
virtual void OnValChanged(
    const TYPE& TNewVal, const TYPE& TPrevVal) const {
    // здесь ничего не делается
}

protected:
// защищенный член класса, охраняющий ресурс;
// используется функциями производного класса
mutable CResGuard m_rg;

private: // закрытые элементы данных
TYPE m_TVal;
};

///////////////////////////////
// к экземплярам этого класса будет обращаться множество потоков, поэтому
// все его члены (кроме конструктора и деструктора) должны быть безопасны
// в многопоточной среде
template <class TYPE>
class CInterlockedScalar : protected CInterlockedType<TYPE> {

public:
CInterlockedScalar(TYPE TVal = 0) : CInterlockedType<TYPE>(TVal) {
}

~CInterlockedScalar() { /* ничего не делает */ }

// C++ не разрешает наследование оператора приведения типа
operator TYPE() const {
    return(CInterlockedType<TYPE>::operator TYPE());
}

TYPE operator=(TYPE TVal) {
    CResGuard::CGuard x(m_rg);
    return(SetVal(TVal));
}

TYPE operator++(int) { // постфиксный оператор увеличения на 1
    CResGuard::CGuard x(m_rg);
    TYPE TPrevVal = GetVal();
}

```

см. след. стр.

Рис. 10-2. продолжение

```

        SetVal((TYPE) (TPrevVal + 1));
        return(TPrevVal); // возвращаем значение до увеличения
    }

TYPE operator--(int) { // постфиксный оператор уменьшения на 1
    CResGuard::CGuard x(m_rg);
    TYPE TPrevVal = GetVal();
    SetVal((TYPE) (TPrevVal - 1));
    return(TPrevVal); // возвращаем значение до уменьшения
}

TYPE operator += (TYPE op)
    { CResGuard::CGuard x(m_rg); return(SetVal(GetVal() + op)); }
TYPE operator++()
    { CResGuard::CGuard x(m_rg); return(SetVal(GetVal() + 1)); }
TYPE operator -= (TYPE op)
    { CResGuard::CGuard x(m_rg); return(SetVal(GetVal() - op)); }
TYPE operator--()
    { CResGuard::CGuard x(m_rg); return(SetVal(GetVal() - 1)); }
TYPE operator *= (TYPE op)
    { CResGuard::CGuard x(m_rg); return(SetVal(GetVal() * op)); }
TYPE operator /= (TYPE op)
    { CResGuard::CGuard x(m_rg); return(SetVal(GetVal() / op)); }
TYPE operator %= (TYPE op)
    { CResGuard::CGuard x(m_rg); return(SetVal(GetVal() % op)); }
TYPE operator ^= (TYPE op)
    { CResGuard::CGuard x(m_rg); return(SetVal(GetVal() ^ op)); }
TYPE operator &= (TYPE op)
    { CResGuard::CGuard x(m_rg); return(SetVal(GetVal() & op)); }
TYPE operator |= (TYPE op)
    { CResGuard::CGuard x(m_rg); return(SetVal(GetVal() | op)); }
TYPE operator <<=(TYPE op)
    { CResGuard::CGuard x(m_rg); return(SetVal(GetVal() << op)); }
TYPE operator >>=(TYPE op)
    { CResGuard::CGuard x(m_rg); return(SetVal(GetVal() >> op)); }
};

////////////////////////////////////////////////////////////////

// к экземплярам этого класса будет обращаться множество потоков, поэтому
// все его члены (кроме конструктора и деструктора) должны быть безопасны
// в многопоточной среде
template <class TYPE>
class CWhenZero : public CInterlockedScalar<TYPE> {
public:
    CWhenZero(TYPE TVal = 0, BOOL fManualReset = TRUE)
        : CInterlockedScalar<TYPE>(TVal) {

        // это событие должно освобождаться при TVal, равном 0
        m_hEvtZero = CreateEvent(NULL, fManualReset, (TVal == 0), NULL);
    }
};

```

Рис. 10-2. продолжение

```

// а это событие должно освобождаться при TVal, НЕ равном 0
m_hevtNotZero = CreateEvent(NULL, fManualReset, (TVal != 0), NULL);
}

~CWhenZero() {
    CloseHandle(m_hevtZero);
    CloseHandle(m_hevtNotZero);
}

// C++ не разрешает наследование оператора =
TYPE operator=(TYPE x) {
    return(CInterlockedScalar<TYPE>::operator=(x));
}

// возвращаем описатель события, которое переходит
// в свободное состояние при нулевом значении
operator HANDLE() const { return(m_hevtZero); }

// возвращаем описатель события, которое переходит
// в свободное состояние при ненулевом значении
HANDLE GetNotZeroHandle() const { return(m_hevtNotZero); }

// C++ не разрешает наследование оператора приведения типа
operator TYPE() const {
    return(CInterlockedScalar<TYPE>::operator TYPE());
}

protected:
void OnValChanged(const TYPE& TNewVal, const TYPE& TPrevVal) const {
    // для большего быстродействия избегайте перехода
    // в режим ядра без веских причин
    if ((TNewVal == 0) && (TPrevVal != 0)) {
        SetEvent(m_hevtZero);
        ResetEvent(m_hevtNotZero);
    }
    if ((TNewVal != 0) && (TPrevVal == 0)) {
        ResetEvent(m_hevtZero);
        SetEvent(m_hevtNotZero);
    }
}

private:
HANDLE m_hevtZero;      // освобождается, когда значение равно 0
HANDLE m_hevtNotZero;   // освобождается, когда значение не равно 0
};

//////////////////////////// Конец файла //////////////////////////////

```

Синхронизация в сценарии «один писатель/группа читателей»

Во многих приложениях возникает одна и та же проблема синхронизации, о которой часто говорят как о сценарии «один писатель/группа читателей» (single-writer/multiple-readers). В чем ее суть? Представьте: произвольное число потоков пытается

получить доступ к некоему разделяемому ресурсу. Каким-то потокам («писателям») нужно модифицировать данные, а каким-то («читателям») — лишь прочесть эти данные. Синхронизация такого процесса необходима хотя бы потому, что Вы должны соблюдать следующие правила:

1. Когда один поток что-то пишет в область общих данных, другие этого делать не могут.
2. Когда один поток что-то пишет в область общих данных, другие не могут ничего считывать оттуда.
3. Когда один поток считывает что-то из области общих данных, другие не могут туда ничего записывать.
4. Когда один поток считывает что-то из области общих данных, другие тоже могут это делать.

Посмотрим на проблему в контексте базы данных. Допустим, с ней работают пять конечных пользователей: двое вводят в нее записи, трое — считывают.

В этом сценарии правило 1 необходимо потому, что мы, конечно же, не можем позволить одновременно обновлять одну и ту же запись. Иначе информация в записи будет повреждена.

Правило 2 запрещает доступ к записи, обновляемой в данный момент другим пользователем. Будь то иначе, один пользователь считывал бы запись, когда другой пользователь изменял бы ее содержимое. Что увидел бы на мониторе своего компьютера первый пользователь, предсказать не берусь. Правило 3 служит тем же целям, что и правило 2. И действительно, какая разница, кто первый получит доступ к данным: тот, кто записывает, или тот, кто считывает, — все равно одновременно этого делать нельзя.

И, наконец, последнее правило. Оно введено для большей эффективности работы баз данных. Если никто не модифицирует записи в базе данных, все пользователи могут свободно читать любые записи. Также предполагается, что количество «читателей» превышает число «писателей».

О’кэй, суть проблемы Вы ухватили. А теперь вопрос: как ее решить?



Я представлю здесь совершенно новый код. Решения этой проблемы, которые я публиковал в прежних изданиях, часто критиковались по двум причинам. Во-первых, предыдущие реализации работали слишком медленно, так как я писал их в расчете на самые разные сценарии. Например, я шире использовал объекты ядра, стремясь синхронизировать доступ к базе данных потоков из разных процессов. Конечно, эти реализации работали и в сценарии для одного процесса, но интенсивное использование объектов ядра приводило в этом случае к существенным издержкам. Похоже, сценарий для одного процесса более распространен, чем я думал.

Во-вторых, в моей реализации был потенциальный риск блокировки потоков-«писателей». Из правил, о которых я рассказал в начале этого раздела, вытекает, что потоки-«писатели» — при обращении к базе данных очень большого количества потоков-«читателей» — могут вообще не получить доступ к этому ресурсу.

Все эти недостатки я теперь устранил. В новой реализации объекты ядра применяются лишь в тех случаях, когда без них не обойтись, и потоки синхронизируются в основном за счет использования критической секции.

Плоды своих трудов я инкапсулировал в C++-класс CSWMRG (я произношу его название как *swimerge*); это аббревиатура от «single writer/multiple reader guard». Он содержится в файлах SWMRG.h и SWMRG.cpp (см. листинг на рис. 10-3).

Использовать CSWMRG проще простого. Вы создаете объект C++-класса CSWMRG и вызываете нужные в Вашей программе функции-члены. В этом классе всего три метода (не считая конструктора и деструктора):

```
VOID CSWMRG::WaitToRead();    // доступ к разделяемому ресурсу для чтения
VOID CSWMRG::WaitToWrite();   // монопольный доступ к разделяемому ресурсу для записи
VOID CSWMRG::Done();         // вызывается по окончании работы с ресурсом
```

Первый метод (*WaitToRead*) вызывается перед выполнением кода, что-либо считающего из разделяемого ресурса, а второй (*WaitToWrite*) — перед выполнением кода, который считывает и записывает данные в разделяемом ресурсе. К последнему методу (*Done*) программа обращается, закончив работу с этим ресурсом. Куда уж проще, а?

Объект CSWMRG содержит набор переменных-членов (см. таблицу ниже), отражающих то, как потоки работают с разделяемым ресурсом на данный момент. Остальные подробности Вы узнаете из исходного кода.

Переменная	Описание
<i>m_cs</i>	Охраняет доступ к остальным членам класса, обеспечивая операции с ними на атомарном уровне.
<i>m_nActive</i>	Отражает текущее состояние разделяемого ресурса. Если она равна 0, ни один поток к ресурсу не обращается. Ее значение, большее 0, сообщает текущее число потоков, считающих данные из ресурса. Отрицательное значение (-1) свидетельствует о том, что какой-то поток записывает данные в ресурс.
<i>m_nWaitingReaders</i>	Сообщает количество потоков-«читателей», которым нужен доступ к ресурсу. Значение этой переменной инициализируется 0 и увеличивается на 1 всякий раз, когда поток вызывает <i>WaitToRead</i> в то время, как <i>m_nActive</i> равна -1.
<i>m_nWaitingWriters</i>	Сообщает количество потоков-«писателей», которым нужен доступ к ресурсу. Значение этой переменной инициализируется 0 и увеличивается на 1 всякий раз, когда поток вызывает <i>WaitToWrite</i> в то время, как <i>m_nActive</i> больше 0.
<i>m_bsemWriters</i>	Когда потоки-«писатели» вызывают <i>WaitToWrite</i> , но получают отказ в доступе, так как <i>m_nActive</i> больше 0, они переходят в состояние ожидания этого семафора. Пока ждет хотя бы один поток-«писатель», новые потоки-«читатели» получают отказ в доступе к ресурсу. Тем самым я не даю потокам-«читателям» монополизировать доступ к этому ресурсу. Когда последний поток-«читатель», работавший с ресурсом, вызывает <i>Done</i> , семафор освобождается со счетчиком, равным 1, и система пробуждает один ждущий поток-«писатель».
<i>m_bsemReaders</i>	Когда потоки-«читатели» вызывают <i>WaitToRead</i> , но получают отказ в доступе, так как <i>m_nActive</i> равна -1, они переходят в состояние ожидания этого семафора. Когда последний из ждущих потоков-«писателей» вызывает <i>Done</i> , семафор освобождается со счетчиком, равным <i>m_nWaitingReaders</i> , и система пробуждает все ждущие потоки-«читатели».

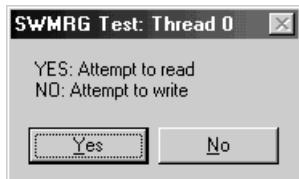
Программа-пример SWMRG

Эта программа, «10 SWMRG.exe» (см. листинг на рис. 10-3), предназначена для тестирования C++-класса CSWMRG. Файлы исходного кода и ресурсов этой программы

находятся в каталоге 10-SWMRG на компакт-диске, прилагаемом к книге. Я запускаю это приложение под управлением отладчика, чтобы наблюдать за всеми функциями и переменными — членами классов.

При запуске программы первичный поток создает несколько потоков, выполняющих одну и ту же функцию. Далее первичный поток вызывает *WaitForMultipleObjects* и ждет завершения этих потоков. Когда все они завершаются, их описатели закрываются и процесс прекращает свое существование.

Каждый вторичный поток выводит на экран такое сообщение:



Чтобы данный поток имитировал чтение ресурса, щелкните кнопку Yes, а чтобы он имитировал запись в ресурс — кнопку No. Эти действия просто заставляют его вызвать либо функцию *WaitToRead*, либо функцию *WaitToWrite* объекта CSWMRG.

После вызова одной из этих функций поток выводит соответствующее сообщение.



Пока окно с сообщением открыто, программа приостанавливает поток и делает вид, будто он сейчас работает с ресурсом.

Конечно, если какой-то поток читает данные из ресурса и Вы командуете другому потоку записать данные в ресурс, окно с сообщением от последнего на экране не появится, так как поток-«писатель» ждет освобождения ресурса, вызвав *WaitToWrite*. Аналогичным образом, если Вы скомандуете потоку считать данные из ресурса в то время, как показывается окно с сообщением от потока-«писателя», первый поток будет ждать в вызове *WaitToRead*, и его окно не появится до тех пор, пока все потоки-«писатели» не закончат имитировать свою работу с ресурсом.

Закрыв окно с сообщением (щелчком кнопки OK), Вы заставите поток, получивший доступ к ресурсу, вызвать *Done*, и объект CSWMRG переключится на другие ждущие потоки.

```
*****  
Модуль: SWMRG.cpp  
Автор: Copyright (c) 2000, Джейфри Рихтер (Jeffrey Richter)  
*****  
#include "..\CmnHdr.h"      /* см. приложение A */  
#include "SWMRG.h"  
//////////
```

Рис. 10-3. Программа-пример SWMRG

Рис. 10-3. продолжение

```

CSWMRG::CSWMRG() {

    // изначально ресурс никому не нужен, и никто к нему не обращается
    m_nWaitingReaders = m_nWaitingWriters = m_nActive = 0;
    m_hsemReaders = CreateSemaphore(NULL, 0, MAXLONG, NULL);
    m_hsemWriters = CreateSemaphore(NULL, 0, MAXLONG, NULL);
    InitializeCriticalSection(&m_cs);
}

//////////////////////////////



CSWMRG::~CSWMRG() {

#ifdef _DEBUG
    // SWMRG нельзя уничтожать, если потоки пользуются ресурсом
    if (m_nActive != 0)
        DebugBreak();
#endif

    m_nWaitingReaders = m_nWaitingWriters = m_nActive = 0;
    DeleteCriticalSection(&m_cs);
    CloseHandle(m_hsemReaders);
    CloseHandle(m_hsemWriters);
}

//////////////////////////////



VOID CSWMRG::WaitToRead() {

    // обеспечиваем монопольный доступ к переменным-членам
    EnterCriticalSection(&m_cs);

    // работает ли сейчас с ресурсом какой-нибудь поток- "писатель"
    // и есть ли "писатели", ждущие этот ресурс?
    BOOL fResourceWritePending = (m_nWaitingWriters || (m_nActive < 0));

    if (fResourceWritePending) {
        // этот "читатель" должен ждать,
        // увеличиваем счетчик числа ждущих "читателей" на 1
        m_nWaitingReaders++;
    } else {
        // этот "читатель" может читать,
        // увеличиваем счетчик числа активных "читателей" на 1
        m_nActive++;
    }

    // разрешаем другим потокам попытаться получить доступ для чтения или записи
    LeaveCriticalSection(&m_cs);

    if (fResourceWritePending) {
}

```

см. след. стр.

Рис. 10-3. продолжение

```
// этот поток должен ждать
WaitForSingleObject(m_hsemReaders, INFINITE);
}

//////////



VOID CSWMRG::WaitToWrite() {

    // обеспечиваем монопольный доступ к переменным-членам
    EnterCriticalSection(&m_cs);

    // работают ли сейчас с ресурсом какие-нибудь потоки?
    BOOL fResourceOwned = (m_nActive != 0);

    if (fResourceOwned) {

        // этот "писатель" должен ждать,
        // увеличиваем счетчик числа ждущих "писателей" на 1
        m_nWaitingWriters++;
    } else {

        // этот "писатель" может писать,
        // уменьшаем счетчик числа активных "писателей" до -1
        m_nActive = -1;
    }

    // разрешаем другим потокам попытаться получить доступ для чтения или записи
    LeaveCriticalSection(&m_cs);

    if (fResourceOwned) {

        // этот поток должен ждать
        WaitForSingleObject(m_hsemWriters, INFINITE);
    }
}

//////////



VOID CSWMRG::Done() {

    // обеспечиваем монопольный доступ к переменным-членам
    EnterCriticalSection(&m_cs);

    if (m_nActive > 0) {

        // ресурс контролируют "читатели", значит, убираем одного из них так
        m_nActive--;
    } else {

        // ресурс контролируют "писатели", значит, убираем одного из них так
    }
}
```

Рис. 10-3. продолжение

```

    m_nActive++;
}

HANDLE hsem = NULL; // предполагаем, что ждущих потоков нет
LONG lCount = 1; // предполагаем, что пробуждается только один ждущий поток
// (в отношении "писателей" это всегда так)

if (m_nActive == 0) {

    // Ресурс свободен, кого пробудить?
    // Примечание: "читатели" никогда не получат доступ к ресурсу,
    // если его всегда будут ждать "писатели".

    if (m_nWaitingWriters > 0) {

        // ресурс ждут "писатели", а они имеют приоритет перед "читателями"
        m_nActive = -1; // писатель получит доступ
        m_nWaitingWriters--; // одним ждущим "писателем" станет меньше
        hsem = m_hsemWriters; // "писатели" ждут на этом семафоре
        // Примечание: семафор откроет путь только одному потоку- "писателю".

    } else if (m_nWaitingReaders > 0) {

        // ресурс ждут "читатели", а "писателей" нет
        m_nActive = m_nWaitingReaders; // все "читатели" получат доступ
        m_nWaitingReaders = 0; // ждущих "читателей" не останется
        hsem = m_hsemReaders; // "читатели" ждут на этом семафоре
        lCount = m_nActive; // семафор откроет путь всем "читателям"
    } else {

        // ждущих потоков вообще нет
    }
}

// разрешаем другим потокам попытаться получить доступ для чтения или записи
LeaveCriticalSection(&m_cs);

if (hsem != NULL) {
    // некоторые потоки следует пробудить
    ReleaseSemaphore(hsem, lCount, NULL);
}
}

//////////////////////////// Конец файла ///////////////////////////////

```

SWMRG.h

```

*****
* Модуль: SWMRG.h
* Автор: Copyright (c) 2000, Джейфри Рихтер (Jeffrey Richter)
*****

```

см. след. стр.

Рис. 10-3. продолжение

```
#pragma once

/////////////////////////////// Конец файла //////////////////////////////

class CSWMRG {
public:
    CSWMRG();           // конструктор
    ~CSWMRG();          // деструктор

    VOID WaitToRead();   // предоставляет доступ к разделяемому ресурсу для чтения
    VOID WaitToWrite();  // предоставляет монопольный доступ к разделяемому
                        // ресурсу для записи
    VOID Done();         // вызывается по окончании работы с ресурсом

private:
    CRITICAL_SECTION m_cs; // обеспечивает монопольный доступ к другим элементам
    HANDLE m_hsemReaders; // "читатели" ждут на этом семафоре,
                          // если ресурс занят "писателем"
    HANDLE m_hsemWriters; // "писатели" ждут на этом семафоре,
                          // если ресурс занят "читателем"
    int    m_nWaitingReaders; // число ждущих "читателей"
    int    m_nWaitingWriters; // число ждущих "писателей"
    int    m_nActive;        // текущее число потоков, работающих с ресурсом
                          // (0 – таких потоков нет, >0 – число "читателей",
                           // -1 – один "писатель")
};

/////////////////////////////// Конец файла //////////////////////////////
```

SWMRGTest.cpp

```
*****
Modуль: SWMRGTest.Cpp
Автор: Copyright (c) 2000, Джейфри Рихтер (Jeffrey Richter)
*****
```

```
#include "..\CmnHdr.h"      /* см. приложение A */
#include <tchar.h>
#include <process.h>          // для доступа к _beginthreadex
#include "SWMRG.h"

/////////////////////////////// Конец файла //////////////////////////////

// глобальный синхронизирующий объект Single-Writer/Multiple-Reader Guard
CSWMRG g_swmgr;

/////////////////////////////// Конец файла //////////////////////////////

DWORD WINAPI Thread(PVOID pvParam) {

    TCHAR sz[50];
```

Рис. 10-3. продолжение

```

wsprintf(sz, TEXT("SWMRG Test: Thread %d"), PtrToShort(pvParam));
int n = MessageBox(NULL,
    TEXT("YES: Attempt to read\nNO: Attempt to write"), sz, MB_YESNO);

// попытка чтения или записи
if (n == IDYES)
    g_swmrg.WaitToRead();
else
    g_swmrg.WaitToWrite();

MessageBox(NULL,
    (n == IDYES) ? TEXT("OK stops READING") : TEXT("OK stops WRITING"),
    sz, MB_OK);

// прекращаем чтение или запись
g_swmrg.Done();
return(0);
}

/////////// Конец файла ///////////////

```

```

int WINAPI _tWinMain(HINSTANCE hinstExe, HINSTANCE, PTSTR pszCmdLine, int) {

// порождаем серию потоков, пытающихся читать или записывать
HANDLE hThreads[MAXIMUM_WAIT_OBJECTS];
for (int nThreads = 0; nThreads < 8; nThreads++) {
    DWORD dwThreadId;
    hThreads[nThreads] =
        chBEGINTHREADEX(NULL, 0, Thread, (PVOID) (DWORD_PTR) nThreads,
            0, &dwThreadId);
}

// ждем завершения всех потоков
WaitForMultipleObjects(nThreads, hThreads, TRUE, INFINITE);
while (nThreads--)
    CloseHandle(hThreads[nThreads]);

return(0);
}

```

```

/////////// Конец файла ///////////////

```

Реализация функции *WaitForMultipleExpressions*

Некоторое время назад я разрабатывал одно приложение и столкнулся с весьма непростым случаем синхронизации потоков. Функции *WaitForMultipleObjects*, заставляющей поток ждать освобождения одного или всех объектов, оказалось недостаточно. Мне понадобилась функция, которая позволяла бы задавать более сложные критерии ожидания. У меня было три объекта ядра: процесс, семафор и событие. Мой поток должен был ждать до тех пор, пока не освободится либо процесс и семафор, либо процесс и событие.

Слегка поразмыслив и творчески использовав имеющиеся функции Windows, я создал именно то, что мне требовалось, — функцию *WaitForMultipleExpressions*. Ее прототип выглядит так:

```
DWORD WINAPI WaitForMultipleExpressions(
    DWORD nExpObjects,
    CONST HANDLE* phExpObjects,
    DWORD dwMilliseconds);
```

Перед ее вызовом Вы должны создать массив описателей (HANDLE) и инициализировать все его элементы. Параметр *nExpObjects* сообщает число элементов в массиве, на который указывает параметр *phExpObjects*. Этот массив содержит несколько наборов описателей объектов ядра; при этом каждый набор отделяется элементом, равным NULL. Функция *WaitForMultipleExpressions* считает все объекты в одном наборе объединяемыми логической операцией AND, а сами наборы — объединяемыми логической операцией OR. Поэтому *WaitForMultipleExpressions* приостанавливает вызывающий поток до тех пор, пока не освободятся сразу все объекты в одном из наборов.

Вот пример. Допустим, мы работаем с четырьмя объектами ядра (см. таблицу ниже).

Объект ядра	Значение описателя
Поток	0x1111
Семафор	0x2222
Событие	0x3333
Процесс	0x4444

Инициализировав массив описателей, как показано в следующей таблице, мы сообщаем функции *WaitForMultipleExpressions* приостановить вызывающий поток до тех пор, пока не освободятся поток AND семафор OR семафор AND событие AND процесс OR поток AND процесс.

Индекс	Значение описателя	Набор
0	0x1111 (поток)	0
1	0x2222 (семафор)	
2	0x0000 (OR)	
3	0x2222 (семафор)	1
4	0x3333 (событие)	
5	0x4444 (процесс)	
6	0x0000 (OR)	
7	0x1111 (поток)	2
8	0x4444 (процесс)	

Вы, наверное, помните, что функции *WaitForMultipleObjects* нельзя передать массив описателей, число элементов в котором превышает 64 (MAXIMUM_WAIT_OBJECTS). Так вот, при использовании *WaitForMultipleExpressions* массив описателей может быть гораздо больше. Однако у Вас не должно быть более 64 выражений, а в каждом — более 63 описателей. Кроме того, *WaitForMultipleExpressions* будет работать некорректно, если Вы передадите ей хотя бы один описатель мьютекса. (Почему — объясню позже.)

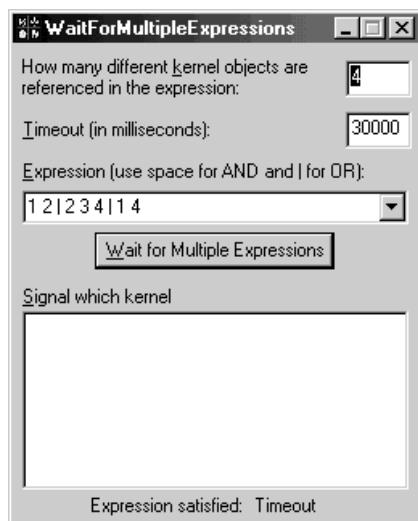
Возвращаемые значения функции *WaitForMultipleExpressions* показаны в следующей таблице. Если заданное выражение становится истинным, *WaitForMultipleExpressions*

возвращает индекс этого выражения относительно WAIT_OBJECT_0. Если взять тот же пример, то при освобождении объектов «поток» и «процесс» *WaitForMultipleExpressions* вернет индекс в виде WAIT_OBJECT_0 + 2.

Возвращаемое значение	Описание
От WAIT_OBJECT_0 до (WAIT_OBJECT_0 + число выражений – 1)	Указывает, какое выражение стало истинным.
WAIT_TIMEOUT	Ни одно выражение не стало истинным в течение заданного времени.
WAIT_FAILED	Произошла ошибка. Чтобы получить более подробную информацию, вызовите <i>GetLastError</i> . Код ERROR_TOO_MANY_SECRETS означает, что Вы указали более 64 выражений, а ERROR_SECRET_TOO_LONG — что по крайней мере в одном выражении указано более 63 объектов. Могут возвращаться коды и других ошибок.

Программа-пример *WaitForMultExp*

Эта программа, «10 WaitForMultExp.exe» (см. листинг на рис. 10-4), предназначена для тестирования функции *WaitForMultipleExpressions*. Файлы исходного кода и ресурсов этой программы находятся в каталоге 10-WaitForMultExp на компакт-диске, прилагаемом к книге. После запуска *WaitForMultExp* открывается диалоговое окно, показанное ниже.

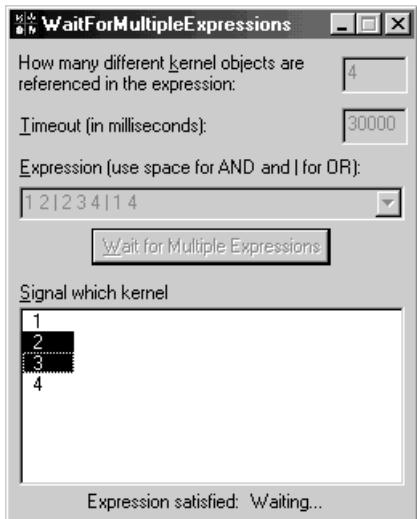


Если Вы не станете изменять предлагаемые параметры, а просто щелкнете кнопку Wait For Multiple Expressions, диалоговое окно будет выглядеть так, как показано на следующей иллюстрации.

Программа создает четыре объекта ядра «событие» в занятом состоянии и помещает в многоколоночный список (с возможностью выбора сразу нескольких элементов) по одной записи для каждого объекта ядра. Далее программа анализирует содержимое поля Expression и формирует массив описателей. По умолчанию я предлагаю объекты ядра и выражение, как в предыдущем примере.

Поскольку я задал время ожидания равным 30000 мс, у Вас есть 30 секунд на внесение изменений. Выбор элемента в нижнем списке приводит к вызову *SetEvent*, ко-

торая освобождает объект, а отказ от его выбора — к вызову *ResetEvent* и соответственно к переводу объекта в занятое состояние. После выбора достаточного числа элементов (удовлетворяющего одному из выражений) *WaitForMultipleExpressions* возвращает управление, и в нижней части диалогового окна показывается, какому выражению удовлетворяет Ваш выбор. Если Вы не уложитесь в 30 секунд, появится слово «Timeout».



Теперь обсудим мою функцию *WaitForMultipleExpressions*. Реализовать ее было не просто, и ее применение, конечно, приводит к некоторым издержкам. Как Вы знаете, в Windows есть функция *WaitForMultipleObjects*, которая позволяет потоку ждать по единственному AND-выражению:

```
DWORD WaitForMultipleObjects(
    DWORD dwObjects,
    CONST HANDLE* phObjects,
    BOOL fWaitAll,
    DWORD dwMilliseconds);
```

Чтобы расширить ее функциональность для поддержки выражений, объединяемых OR, я должен создать несколько потоков — по одному на каждое такое выражение. Каждый из этих потоков ждет в вызове *WaitForMultipleObjectsEx* по единственному AND-выражению. (Почему я использую эту функцию вместо более распространенной *WaitForMultipleObjects* — станет ясно позже.) Когда какое-то выражение становится истинным, один из созданных потоков пробуждается и завершается.

Поток, который вызвал *WaitForMultipleExpressions* (и который породил все OR-потоки), должен ждать, пока одно из OR-выражений не станет истинным. Для этого он вызывает функцию *WaitForMultipleObjectsEx*. В параметре *dwObjects* передается количество порожденных потоков (OR-выражений), а параметр *phObjects* указывает на массив описателей этих потоков. В параметр *fWaitAll* записывается FALSE, чтобы основной поток пробудился сразу после того, как станет истинным любое из выражений. И, наконец, в параметре *dwMilliseconds* передается значение, идентичное тому, которое было указано в аналогичном параметре при вызове *WaitForMultipleExpressions*.

Если в течение заданного времени ни одно из выражений не становится истинным, *WaitForMultipleObjectsEx* возвращает WAIT_TIMEOUT, и это же значение возвращается функцией *WaitForMultipleExpressions*. А если какое-нибудь выражение становит-

ся истинным, *WaitForMultipleObjectsEx* возвращает индекс, указывающий, какой поток завершился. Так как каждый поток представляет отдельное выражение, этот индекс сообщает и то, какое выражение стало истинным; этот же индекс возвращается и функцией *WaitForMultipleExpressions*.

На этом мы, пожалуй, закончим рассмотрение того, как работает функция *WaitForMultipleExpressions*. Но нужно обсудить еще три вещи. Во-первых, нельзя допустить, чтобы несколько OR-потоков одновременно пробудились в своих вызовах *WaitForMultipleObjectsEx*, так как успешное ожидание некоторых объектов ядра приводит к изменению их состояния (например, у семафора счетчик уменьшается на 1). *WaitForMultipleExpressions* ждет лишь до тех пор, пока одно из выражений не станет истинным, а значит, я должен предотвратить более чем однократное изменение состояния объекта.

Решить эту проблему на самом деле довольно легко. Прежде чем порождать OR-потоки, я создаю собственный объект-семафор с начальным значением счетчика, равным 1. Далее каждый OR-поток вызывает *WaitForMultipleObjectsEx* и передает ей не только описатели объектов, связанных с выражением, но и описатель этого семафора. Теперь Вы понимаете, почему в каждом наборе не может быть более 63 описателей? Чтобы OR-поток пробудился, должны освободиться все объекты, которые он ждет, — в том числе мой специальный семафор. Поскольку начальное значение его счетчика равно 1, более одного OR-потока никогда не пробудится, и, следовательно, случайного изменения состояния каких-либо других объектов не произойдет.

Второе, на что нужно обратить внимание, — как заставить ждущий поток прекратить ожидание для корректной очистки. Добавление семафора гарантирует, что пробудится не более чем один поток, но, раз мне уже известно, какое выражение стало истинным, я должен пробудить и остальные потоки, чтобы они корректно завершились. Вызова *TerminateThread* следует избегать, поэтому нужен какой-то другой механизм. Поразмыслив, я вспомнил, что потоки, ждущие в «тревожном» состоянии, принудительно пробуждаются, когда в APC-очереди появляется какой-нибудь элемент.

Моя реализация *WaitForMultipleExpressions* для принудительного пробуждения потоков использует *QueueUserAPC*. После того как *WaitForMultipleObjects*, вызванная основным потоком, возвращает управление, я ставлю APC-вызов в соответствующие очереди каждого из все еще ждущих OR-потоков:

```
// выводим все еще ждущие потоки из состояния сна,
// чтобы они могли корректно завершиться
for (dwExpNum = 0; dwExpNum < dwNumExps; dwExpNum++) {

    if ((WAIT_TIMEOUT == dwWaitRet) || (dwExpNum != (dwWaitRet - WAIT_OBJECT_0))) {
        QueueUserAPC(WFME_ExpressionAPC, ahThreads[dwExpNum], 0);
    }
}
```

Функция обратного вызова, *WFME_ExpressionAPC*, выглядит столь странно потому, что на самом деле от нее не требуется ничего, кроме одного: прервать ожидание потока.

```
// это APC-функция обратного вызова
VOID WINAPI WFME_ExpressionAPC(DWORD dwData) {
    // в тело функции преднамеренно не включено никаких операторов
}
```

Третье (и последнее) — правильная обработка интервалов ожидания. Если никакие выражения так и не стали истинными в течение заданного времени, функция

WaitForMultipleObjects, вызванная основным потоком, возвращает WAIT_TIMEOUT. В этом случае я должен позаботиться о том, чтобы ни одно выражение больше не стало бы истинным и тем самым не изменило бы состояние объектов. За это отвечает следующий код:

```
// ждем, когда выражение станет TRUE или когда истечет срок ожидания
dwWaitRet = WaitForMultipleObjects(dwExpNum, ahThreads, FALSE, dwMilliseconds);

if (WAIT_TIMEOUT == dwWaitRet) {

    // срок ожидания истек; выясняем, не стало ли какое-нибудь выражение
    // истинным, проверяя состояние семафора hsemOnlyOne
    dwWaitRet = WaitForSingleObject(hsemOnlyOne, 0);

    if (WAIT_TIMEOUT == dwWaitRet) {

        // если семафор не был переведен в свободное состояние,
        // какое-то выражение дало TRUE; надо выяснить – какое
        dwWaitRet = WaitForMultipleObjects(dwExpNum,
            ahThreads, FALSE, INFINITE);

    } else {

        // ни одно выражение не стало TRUE,
        // и WaitForSingleObject просто отдала нам семафор
        dwWaitRet = WAIT_TIMEOUT;
    }
}
```

Я не даю другим выражениям стать истинными за счет ожидания на семафоре. Это приводит к уменьшению счетчика семафора до 0, и никакой OR-поток не может пробудиться. Но где-то после вызова функции *WaitForMultipleObjects* из основного потока и обращения той к *WaitForSingleObject* одно из выражений может стать истинным. Вот почему я проверяю значение, возвращаемое *WaitForSingleObject*. Если она возвращает WAIT_OBJECT_0, значит, семафор захвачен основным потоком и ни одно из выражений не стало истинным. Но если она возвращает WAIT_TIMEOUT, какое-то выражение все же стало истинным, прежде чем основной поток успел захватить семафор. Чтобы выяснить, какое именно выражение дало TRUE, основной поток снова вызывает *WaitForMultipleObjects*, но уже с временем ожидания, равным INFINITE; здесь все в порядке, так как я знаю, что семафор захвачен OR-потоком и этот поток вот-вот завершится. Теперь я должен пробудить остальные OR-потоки, чтобы корректно завершить их. Это делается в цикле, из которого вызывается *QueueUserAPC* (о ней я уже рассказывал).

Поскольку реализация *WaitForMultipleExpressions* основана на использовании группы потоков, каждый из которых ждет на своем наборе объектов, объединяемых по AND, мьютексы в ней неприменимы. В отличие от остальных объектов ядра мьютексы могут передаваться потоку во владение. Значит, если какой-нибудь из моих AND-потоков заполучит мьютекс, то по его завершении произойдет отказ от мьютекса. Вот когда Microsoft добавит в Windows API функцию, позволяющую одному потоку передавать права на владение мьютексом другому потоку, тогда моя функция *WaitForMultipleExpressions* и сможет поддерживать мьютексы. А пока надежного и корректного способа ввести в *WaitForMultipleExpressions* такую поддержку я не вижу.

**WaitForMultExp.cpp**

```
*****
Modуль: WaitForMultExp.cpp
Автор: Copyright (c) 2000, Джейфри Рихтер (Jeffrey Richter)
*****
```

```
#include "..\CmnHdr.h"      /* см. приложение A */
#include <malloc.h>
#include <process.h>
#include "WaitForMultExp.h"

//////////



// внутренняя структура данных, которая представляет одно выражение;
// используется для того, чтобы сообщать OR-потокам,
// на каких объектах они должны ждать
typedef struct {
    PHANDLE m_phExpObjects; // указывает на набор описателей
    DWORD    m_nExpObjects; // количество описателей
} PEXPRESSION, *PEXPRESSION;

//////////



// функция OR-потока
DWORD WINAPI WFME_ThreadExpression(PVOID pvParam) {

    // Эта функция просто ждет, когда выражение станет истинным.
    // Ее поток ждет в "тревожном" состоянии, чтобы его ожидание
    // можно было принудительно прервать, поместив в его APC-очередь
    // APC-вызов.
    PEXPRESSION pExpression = (PEXPRESSION) pvParam;
    return( WaitForMultipleObjectsEx(
        pExpression->m_nExpObjects, pExpression->m_phExpObjects,
        TRUE, INFINITE, TRUE));
}

//////////


// это APC-функция обратного вызова
VOID WINAPI WFME_ExpressionAPC(ULONG_PTR dwData) {

    // в тело функции преднамеренно не включено никаких операторов
}

//////////


// эта функция ждет по нескольким булевым выражениям
DWORD WINAPI WaitForMultipleExpressions(DWORD nExpObjects,
    CONST HANDLE* phExpObjects, DWORD dwMilliseconds) {
```

Рис. 10-4. Программа-пример *WaitForMultExp*

см. след. стр.

Рис. 10-4. продолжение

```
// создаем временный массив, потому что нам придется модифицировать
// передаваемый массив и добавить в его конец описатель для
// семафора hsemOnlyOne
PHANDLE phExpObjectsTemp = (PHANDLE)
    _alloca(sizeof(HANDLE) * (nExpObjects + 1));
CopyMemory(phExpObjectsTemp, phExpObjects, sizeof(HANDLE) * nExpObjects);
phExpObjectsTemp[nExpObjects] = NULL; // ставим сюда часового

// этот семафор гарантирует, что только одно выражение станет истинным
HANDLE hsemOnlyOne = CreateSemaphore(NULL, 1, 1, NULL);

// информация о выражении
EXPRESSION Expression[MAXIMUM_WAIT_OBJECTS];

DWORD dwExpNum    = 0; // номер текущего выражения
DWORD dwNumExps   = 0; // общее количество выражений

DWORD dwObjBegin = 0; // первый индекс набора
DWORD dwObjCur   = 0; // текущий индекс объекта в наборе

DWORD dwThreadId, dwWaitRet = 0;

// массив описателей потоков
HANDLE ahThreads[MAXIMUM_WAIT_OBJECTS];

// разбираем список описателей вызывающих потоков, инициализируя структуру
// для каждого выражения и добавляя к нему hsemOnlyOne
while ((dwWaitRet != WAIT_FAILED) && (dwObjCur <= nExpObjects)) {

    // пока ошибок нет и описатели объектов находятся в списке вызывающего...

    // находим следующее выражение (OR-выражения разделяются NULL-описателями)
    while (phExpObjectsTemp[dwObjCur] != NULL)
        dwObjCur++;

    // инициализируем структуру Expression, на которой ждет OR-поток
    phExpObjectsTemp[dwObjCur] = hsemOnlyOne;
    Expression[dwNumExps].m_phExpObjects = &phExpObjectsTemp[dwObjBegin];
    Expression[dwNumExps].m_nExpObjects = dwObjCur - dwObjBegin + 1;

    if (Expression[dwNumExps].m_nExpObjects > MAXIMUM_WAIT_OBJECTS) {
        // ошибка: слишком много описателей в одном выражении
        dwWaitRet = WAIT_FAILED;
        SetLastError(ERROR_SECRET_TOO_LONG);
    }

    // переходим к следующему выражению
    dwObjBegin = ++dwObjCur;
    if (++dwNumExps == MAXIMUM_WAIT_OBJECTS) {
        // ошибка: слишком много выражений
        dwWaitRet = WAIT_FAILED;
```

Рис. 10-4. продолжение

```

        SetLastError(ERROR_TOO_MANY_SECRETS);
    }
}

if (dwWaitRet != WAIT_FAILED) {
    // при разборе списка описателей ошибки не обнаружены

    // порождаем поток для каждого выражения
    for (dwExpNum = 0; dwExpNum < dwNumExps; dwExpNum++) {

        ahThreads[dwExpNum] = chBEGINTHREADEX(NULL,
            1, // нам нужен лишь небольшой стек
            WFME_ThreadExpression, &Expression[dwExpNum],
            0, &dwThreadId);
    }

    // ждем, когда выражение станет TRUE или когда истечет срок ожидания
    dwWaitRet = WaitForMultipleObjects(dwExpNum, ahThreads,
        FALSE, dwMilliseconds);

    if (WAIT_TIMEOUT == dwWaitRet) {

        // срок ожидания истек; выясняем, не стало ли какое-нибудь выражение
        // истинным, проверяя состояние семафора hsemOnlyOne
        dwWaitRet = WaitForSingleObject(hsemOnlyOne, 0);

        if (WAIT_TIMEOUT == dwWaitRet) {

            // если семафор не был переведен в свободное состояние,
            // какое-то выражение дало TRUE; надо выяснить – какое
            dwWaitRet = WaitForMultipleObjects(dwExpNum,
                ahThreads, FALSE, INFINITE);

        } else {

            // ни одно выражение не стало TRUE,
            // и WaitForSingleObject просто отдала нам семафор
            dwWaitRet = WAIT_TIMEOUT;
        }
    }

    // прерываем ожидание всех потоков (ждущих свои выражения),
    // чтобы они могли корректно завершиться
    for (dwExpNum = 0; dwExpNum < dwNumExps; dwExpNum++) {

        if ((WAIT_TIMEOUT == dwWaitRet) ||
            (dwExpNum != (dwWaitRet - WAIT_OBJECT_0))) {
            QueueUserAPC(WFME_ExpressionAPC, ahThreads[dwExpNum], 0);
        }
    }
}

```

см. след. стр.

Рис. 10-4. продолжение

```
#ifdef _DEBUG
    // при отладочной сборке ждем завершения всех потоков выражений,
    // а при окончательной сборке считаем, что все работает, как надо,
    // и не заставляем этот поток ждать их завершения
    WaitForMultipleObjects(dwExpNum, ahThreads, TRUE, INFINITE);
#endif

    // закрываем свои описатели всех потоков выражений
    for (dwExpNum = 0; dwExpNum < dwNumExps; dwExpNum++) {
        CloseHandle(ahThreads[dwExpNum]);
    }
} // при разборе произошла ошибка

CloseHandle(hsemOnlyOne);
return(dwWaitRet);
}

/////////////////// Конец файла //////////////////
```

WaitForMultExp.h

```
*****
Модуль: WaitForMultExp.h
Автор: Copyright (c) 2000, Джейфри Рихтер (Jeffrey Richter)
*****
```

```
#pragma once

///////////////////

DWORD WINAPI WaitForMultipleExpressions(DWORD nExpObjects,
    CONST HANDLE* phExpObjects, DWORD dwMilliseconds);

/////////////////// Конец файла //////////////////
```

WfMETest.cpp

```
*****
Модуль: WfMETest.cpp
Автор: Copyright (c) 2000, Джейфри Рихтер (Jeffrey Richter)
*****
```

```
#include "..\CmnHdr.h"      /* см. приложение A */
#include <windowsx.h>
#include <tchar.h>
#include <process.h>
#include "resource.h"
#include "WaitForMultExp.h"

///////////////////
```

см. след. стр.

Рис. 10-4. продолжение

```

// g_ah0objs содержит список описателей объектов ядра "событие",
// на которые ссылается булево выражение
#define MAX_KERNEL_OBJS      1000
HANDLE g_ah0objs[MAX_KERNEL_OBJS];

// ahExp0objs содержит все выражения; каждое выражение состоит
// из набора описателей объектов ядра и становится TRUE, когда
// все эти объекты одновременно переходят в свободное состояние;
// чтобы отделить OR-выражения друг от друга, используем NULL-описатель

// один и тот же описатель может встречаться в AND-выражении только раз,
// но может повторяться в OR-выражениях

// выражение может содержать максимум 64 набора, а каждый набор –
// не более 63 описателей плюс NULL-описатель (разделитель наборов)
#define MAX_EXPRESSION_SIZE   ((64 * 63) + 63)

// m_nExp0bjects – число элементов, задействованных в массиве ahExp0bjects
typedef struct {
    HWND   m_hwnd;                      // куда посыпать результаты
    DWORD  m_dwMilliseconds;            // сколько ждать
    DWORD  m_nExp0bjects;                // количество элементов
                                         // в списке объектов
    HANDLE m_ahExp0objs[MAX_EXPRESSION_SIZE]; // список объектов
} AWFME, *PAWFME;
AWFME g_awfme;

// это сообщение асинхронно посылается потоку пользовательского интерфейса,
// когда какое-то выражение становится истинным или заканчивается заданное
// время ожидания
#define WM_WAITEND      (WM_USER + 101)

///////////////////////////////



BOOL Dlg_OnInitDialog(HWND hwnd, HWND hwndFocus, LPARAM lParam) {
    chSETDLGICONS(hwnd, IDI_WFMETEXT);

    // инициализируем элементы управления в диалоговом окне
    SetDlgItemInt(hwnd, IDC_NUMOJBS, 4, FALSE);
    SetDlgItemInt(hwnd, IDC_TIMEOUT, 30000, FALSE);
    SetDlgItemText(hwnd, IDC_EXPRESSION,
                  _T("1 2 | 2 3 4 | 1 4"));

    // устанавливаем размер колонки в окне многоколоночного списка
    ListBox_SetColumnWidth(GetDlgItem(hwnd, IDC_OBJLIST),
                           LOWORD(GetDialogBaseUnits()) * 4);

    return(TRUE);
}

```

см. след. стр.

Рис. 10-4. продолжение

```
//////////  
DWORD WINAPI AsyncWaitForMultipleExpressions(PVOID pvParam) {  
  
    PAWFME pawfme = (PAWFME) pvParam;  
  
    DWORD dw = WaitForMultipleExpressions(pawfme->m_nExpObjects,  
                                           pawfme->m_ahExpObjs, pawfme->m_dwMilliseconds);  
    PostMessage(pawfme->m_hwnd, WM_WAITEND, dw, 0);  
    return(0);  
}  
  
//////////  
  
LRESULT Dlg_OnWaitEnd(HWND hwnd, WPARAM wParam, LPARAM lParam) {  
  
    // закрываем все описатели объектов ядра "событие"  
    for (int n = 0; g_ah0bj[n] != NULL; n++)  
        CloseHandle(g_ah0bj[n]);  
  
    // сообщаем пользователю результат выполнения теста  
    if (wParam == WAIT_TIMEOUT)  
        SetDlgItemText(hwnd, IDC_RESULT, _T("Timeout"));  
    else  
        SetDlgItemInt(hwnd, IDC_RESULT, (DWORD) wParam - WAIT_OBJECT_0, FALSE);  
  
    // даем возможность изменить значения и повторно выполнить тест  
    EnableWindow(GetDlgItem(hwnd, IDC_NUMOJBS), TRUE);  
    EnableWindow(GetDlgItem(hwnd, IDC_TIMEOUT), TRUE);  
    EnableWindow(GetDlgItem(hwnd, IDC_EXPRESSION), TRUE);  
    EnableWindow(GetDlgItem(hwnd, IDOK), TRUE);  
    SetFocus(GetDlgItem(hwnd, IDC_EXPRESSION));  
  
    return(0);  
}  
  
//////////  
  
void Dlg_OnCommand(HWND hwnd, int id, HWND hwndCtl, UINT codeNotify) {  
  
    // получаем пользовательские настройки  
    // из элементов управления диалогового окна  
    TCHAR szExpression[100];  
    ComboBox_GetText(GetDlgItem(hwnd, IDC_EXPRESSION), szExpression,  
                    sizeof(szExpression) / sizeof(szExpression[0]));  
  
    int nObjects = GetDlgItemInt(hwnd, IDC_NUMOJBS, NULL, FALSE);  
  
    switch (id) {  
    case IDCANCEL:  
        EndDialog(hwnd, id);  
        break;
```

Рис. 10-4. продолжение

```

case IDC_OBJLIST:
    switch (codeNotify) {
        case LBN_SELCHANGE:
            // состояние элемента изменено, сбрасываем все элементы
            // и устанавливаем только выбранные
            for (int n = 0; n < nObjects; n++)
                ResetEvent(g_ahObjs[n]);

            for (n = 0; n < nObjects; n++) {
                if (ListBox_GetSel(GetDlgItem(hwnd, IDC_OBJLIST), n))
                    SetEvent(g_ahObjs[n]);
            }
            break;
    }
    break;

case IDOK:
    // запрещаем изменение значений в процессе выполнения теста
    SetFocus(GetDlgItem(hwnd, IDC_OBJLIST));
    EnableWindow(GetDlgItem(hwnd, IDC_NUMOJBS), FALSE);
    EnableWindow(GetDlgItem(hwnd, IDC_TIMEOUT), FALSE);
    EnableWindow(GetDlgItem(hwnd, IDC_EXPRESSION), FALSE);
    EnableWindow(GetDlgItem(hwnd, IDOK), FALSE);

    // уведомляем пользователя, что тест выполняется
    SetDlgItemText(hwnd, IDC_RESULT, TEXT("Waiting..."));

    // создаем все необходимые объекты ядра
    ZeroMemory(g_ahObjs, sizeof(g_ahObjs));
    g_awfme.m_nExpObjects = 0;
    ZeroMemory(g_awfme.m_ahExpObjs, sizeof(g_awfme.m_ahExpObjs));
    g_awfme.m_hwnd = hwnd;
    g_awfme.m_dwMilliseconds = GetDlgItemInt(hwnd, IDC_TIMEOUT, NULL, FALSE);

    ListBox_ResetContent(GetDlgItem(hwnd, IDC_OBJLIST));
    for (int n = 0; n < nObjects; n++) {
        TCHAR szBuf[20];
        g_ahObjs[n] = CreateEvent(NULL, FALSE, FALSE, NULL);

        wsprintf(szBuf, TEXT(" %d"), n + 1);
        ListBox_AddString(GetDlgItem(hwnd, IDC_OBJLIST),
                          &szBuf[lstrlen(szBuf) - 3]);
    }

    PTSTR p = _tcstok(szExpression, TEXT(" "));
    while (p != NULL) {
        g_awfme.m_ahExpObjs[g_awfme.m_nExpObjects++] =
            (*p == TEXT('|')) ? NULL : g_ahObjs[_ttoi(p) - 1];
        p = _tcstok(NULL, TEXT(" "));
    }
}

```

см. след. стр.

Рис. 10-4. продолжение

```
DWORD dwThreadId;
CloseHandle(chBEGINTHREADEX(NULL, 0,
    AsyncWaitForMultipleExpressions, &g_awfme,
    0, &dwThreadId));
break;
}

////////// INT_PTR WINAPI Dlg_Proc(HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam) {
switch (uMsg) {
    chHANDLE_DLGMMSG(hwnd, WM_INITDIALOG, Dlg_OnInitDialog);
    chHANDLE_DLGMMSG(hwnd, WM_COMMAND, Dlg_OnCommand);

    case WM_WAITEND:
        return(Dlg_OnWaitEnd(hwnd, wParam, lParam));
}

return(FALSE);
}

////////// int WINAPI _tWinMain(HINSTANCE hinstExe, HINSTANCE, PTSTR pszCmdLine, int) {
DialogBox(hinstExe, MAKEINTRESOURCE(IDD_TESTW4ME), NULL, Dlg_Proc);
return(0);
}

////////// Конец файла /////////////////
```

Пулы потоков

В главе 8 мы обсудили синхронизацию потоков без перехода в режим ядра. Замечательная особенность такой синхронизации — высокое быстродействие. И если Вы озабочены быстродействием потока, сначала подумайте, нельзя ли обойтись синхронизацией в пользовательском режиме.

Вы уже знаете, что создание многопоточных приложений — дело трудное. Вас подстерегают две серьезные проблемы: управление созданием и уничтожением потоков и синхронизация их доступа к ресурсам. Для решения второй проблемы в Windows предусмотрено множество синхронизирующих примитивов: события, семафоры, мьютексы, критические секции и др. Все они довольно просты в использовании. Но если бы система автоматически охраняла разделяемые ресурсы, вот тогда создавать многопоточные приложения было бы по-настоящему легко. Увы, операционной системе Windows до этого еще далеко.

Проблему того, как управлять созданием и уничтожением потоков, каждый решает по-своему. За прошедшие годы я создал несколько реализаций пулов потоков, рассчитанных на определенные сценарии. Однако в Windows 2000 появился ряд новых функций для операций с пулами потоков; эти функции упрощают создание, уничтожение и общий контроль за потоками. Конечно, встроенные в них механизмы носят общий характер и не годятся на все случаи жизни, но зачастую их вполне достаточно, и они позволяют экономить массу времени при разработке многопоточного приложения.

Эти функции дают возможность вызывать другие функции асинхронно, через определенные промежутки времени, при освобождении отдельных объектов ядра или при завершении запросов на асинхронный ввод-вывод.

Пул подразделяется на четыре компонента, которые описываются в таблице 11-1.

	таймера	Компонент поддержки ожидания	ввода-вывода	других операций
Начальное число потоков	Всегда 1	1	0	0
Когда поток создается	При вызове первой функции таймера пула потоков	Один поток для каждого из 63 зарегистрированных объектов	В системе применяются эвристические методы, но на создание потока влияют следующие факторы: <ul style="list-style-type: none"> ■ после добавления потока прошло определенное время ■ установлен флаг <code>WT_EXECUTEFUNCTION</code> ■ число элементов в очереди превышает пороговое значение 	

Таблица 11-1. Компоненты поддержки пула потоков

см. след. стр.

таймера	Компонент поддержки ожидания	ввода-вывода	других операций
Когда поток разрушается	При завершении процесса	При отсутствии зарегистрированных объектов ожидания	При простое потока в течение определенного порогового времени (около минуты)
Как поток ждет	В «тревожном» состоянии	WaitForMultipleObjectsEx	GetQueuedCompletionStatus
Когда поток пробуждается	При освобождении «ожидаемого таймера», который посылает в очередь APC-вызов	При освобождении объекта ядра	При поступлении запроса о статусе завершения или о завершении ввода-вывода (порт завершения требует, чтобы число потоков не превышало число процессоров более чем в 2 раза)

При инициализации процесса никаких издержек, связанных с перечисленными в таблице компонентами поддержки, не возникает. Однако, как только вызывается одна из функций пула потоков, для процесса создается набор этих компонентов, и некоторые из них сохраняются до его завершения. Как видите, издержки от применения этих функций отнюдь не малые: частью Вашего процесса становится целый набор потоков и внутренних структур данных. Так что, прежде чем пользоваться ими, тщательно взвесьте все «за» и «против».

О'кэй, теперь, когда я Вас предупредил, посмотрим, как все это работает.

Сценарий 1: асинхронный вызов функций

Допустим, у Вас есть серверный процесс с основным потоком, который ждет клиентский запрос. Получив его, он порождает отдельный поток для обработки этого запроса. Тём самым основной поток освобождается для приема следующего клиентского запроса. Такой сценарий типичен в клиент-серверных приложениях. Хотя он и такто незатейлив, при желании его можно реализовать с использованием новых функций пула потоков.

Получая клиентский запрос, основной поток вызывает:

```
BOOL QueueUserWorkItem(
    PTHREAD_START_ROUTINE pfnCallback,
    PVOID pvContext,
    ULONG dwFlags);
```

Эта функция помещает «рабочий элемент» (work item) в очередь потока в пуле и тут же возвращает управление. Рабочий элемент — это просто вызов функции (на которую ссылается параметр *pfnCallback*), принимающей единственный параметр, *pvContext*. В конечном счете какой-то поток из пула займется обработкой этого эле-

мента, в результате чего будет вызвана Ваша функция. У этой функции обратного вызова, за реализацию которой отвечаете Вы, должен быть следующий прототип:

```
DWORD WINAPI WorkItemFunc(PVOID pvContext);
```

Несмотря на то что тип возвращаемого значения определен как DWORD, на самом деле оно игнорируется.

Обратите внимание, что Вы сами никогда не вызываете *CreateThread*. Она вызывается из пула потоков, автоматически создаваемого для Вашего процесса, а к функции *WorkItemFunc* обращается один из потоков этого пула. Кроме того, данный поток не уничтожается сразу после обработки клиентского запроса, а возвращается в пул, оставаясь готовым к обработке любых других элементов, помещаемых в очередь. Ваше приложение может стать гораздо эффективнее, так как Вам больше не придется создавать и уничтожать потоки для каждого клиентского запроса. А поскольку потоки связаны с определенным портом завершения, количество одновременно работающих потоков не может превышать число процессоров более чем в 2 раза. За счет этого переключения контекста происходят реже.

Многое в пуле потоков происходит скрытно от разработчика: *QueueUserWorkItem* проверяет число потоков, включенных в сферу ответственности компонента поддержки других операций (не относящихся к вводу-выводу), и в зависимости от текущей нагрузки (количества рабочих элементов в очереди) может передать ему другие потоки. После этого *QueueUserWorkItem* выполняет операции, эквивалентные вызову *PostQueuedCompletionStatus*, пересыпая информацию о рабочем элементе в порт завершения ввода-вывода. В конечном счете поток, ждущий на этом объекте, извлекает Ваше сообщение (вызовом *GetQueuedCompletionStatus*) и обращается к Вашей функции. После того как она возвращает управление, поток вновь вызывает *GetQueuedCompletionStatus*, ожидая появления следующего рабочего элемента.

Пул рассчитан на частую обработку асинхронного ввода-вывода — всякий раз, когда поток помещает в очередь запрос на ввод-вывод к драйверу устройства. Пока драйвер выполняет его, поток, поставивший запрос в очередь, не блокируется и может заниматься другой работой. Асинхронный ввод-вывод — ключ к созданию высокоэффективных, масштабируемых приложений, так как позволяет одному потоку обрабатывать запросы от множества клиентов по мере их поступления; ему не придется обрабатывать их последовательно или останавливаться, ожидая завершения ввода-вывода.

Но Windows накладывает одно ограничение на запросы асинхронного ввода-вывода: если поток, послав такой запрос драйверу устройства, завершается, данный запрос теряется и никакие потоки о его судьбе не уведомляются. В хорошо продуманном пуле, число потоков увеличивается и уменьшается в зависимости от потребностей его клиентов. Поэтому, если поток посыпает запрос и уничтожается из-за сокращения пула, то уничтожается и этот запрос. Как правило, это не совсем то, что хотелось бы, и здесь нужно найти какое-то решение.

Если Вы хотите поместить в очередь рабочий элемент, который выдает запрос на асинхронный ввод-вывод, то не сможете передать этот элемент компоненту поддержки других операций в пуле потоков. Его примет лишь компонент поддержки ввода-вывода. Последний включает набор потоков, которые не завершаются, пока есть хотя бы один запрос на ввод-вывод; поэтому для выполнения кода, выдающего запросы на асинхронный ввод-вывод, Вы должны пользоваться только этими потоками.

Чтобы передать рабочий элемент компоненту поддержки ввода-вывода, Вы можете по-прежнему пользоваться функцией *QueueUserWorkItem*, но в параметре *dwFlags*

следует указать флаг WT_EXECUTEINIOTHREAD. А обычно Вы будете указывать в этом параметре флаг WT_EXECUTEDEFAULT (0) — он заставляет систему передать рабочий элемент компоненту поддержки других операций (не связанных с вводом-выводом).

В Windows есть функции (вроде *RegNotifyChangeKeyValue*), которые асинхронно выполняют операции, не относящиеся к вводу-выводу. Они также требуют, чтобы вызывающий поток не завершался преждевременно. С этой целью Вы можете использовать флаг WT_EXECUTEINPERSISTENTTHREAD, который заставляет поток таймера выполнять поставленную в очередь функцию обратного вызова для рабочего элемента. Так как этот компонент существует постоянно, асинхронная операция в конечном счете обязательно будет выполнена. Вы должны позаботиться о том, чтобы функция обратного вызова выполнялась быстро и не блокировала работу компонента поддержки таймера.

Хорошо продуманный пул должен также обеспечивать максимальную готовность потоков к обработке запросов. Если в пуле четыре потока, а в очереди сто рабочих элементов, то единовременно можно обработать только четыре элемента. Это не проблема, если на обработку каждого элемента уходит лишь несколько миллисекунд, но в ином случае программа не сумеет своевременно обслуживать запросы.

Естественно, система не настолько умна, чтобы предвидеть, чем будет заниматься функция Вашего рабочего элемента, но если Вам заранее известно, что на это уйдет длительное время, вызовите *QueueUserWorkItem* с флагом WT_EXECUTEONGFUNCTION — он заставит пул создать новый поток, если остальные потоки будут в это время заняты. Так, добавив в очередь 10 000 рабочих элементов (с флагом WT_EXECUTEONGFUNCTION), Вы получите 10 000 новых потоков в пуле. Чтобы избежать этого, делайте перерывы между вызовами *QueueUserWorkItem*, и тогда часть потоков успеет завершиться до порождения новых.

Ограничение на количество потоков в пуле накладывать нельзя, иначе может возникать взаимная блокировка потоков. Представьте очередь из 10 000 элементов, заблокированных 10 001-м и ждущих его освобождения. Установив предел в 10 000, Вы запретите выполнение 10 001-го потока, и в результате целых 10 000 потоков останутся навечно заблокированными.

Используя функции пула, будьте осторожны, чтобы не доводить дело до тупиковых ситуаций. Особую осторожность проявляйте, если функция Вашего рабочего элемента использует критические секции, семафоры, мьютексы и др. — это увеличивает вероятность взаимной блокировки. Вы должны всегда точно знать, поток какого компонента пула выполняет Ваш код. Также будьте внимательны, если функция рабочего элемента содержится в DLL, которая может быть динамически выгружена из памяти. Поток, вызывающий функцию из выгруженной DLL, приведет к нарушению доступа. Чтобы предотвратить выгрузку DLL при наличии рабочих элементов в очереди, создайте контрольный счетчик для таких элементов: его значение должно увеличиваться перед вызовом *QueueUserWorkItem* и уменьшаться после выполнения функции рабочего элемента. Выгрузка DLL допустима только после того, как этот счетчик обнулся.

Сценарий 2: вызов функций через определенные интервалы времени

Иногда какие-то операции приходится выполнять через определенные промежутки времени. В Windows имеется объект ядра «ожидаемый таймер», который позволяет легко получать уведомления по истечении заданного времени. Многие программисты создают такой объект для каждой привязанной к определенному времени задаче, но это ошибочный путь, ведущий к пустой трате системных ресурсов. Вместо этого

Вы можете создать единственный ожидаемый таймер и каждый раз перенастраивать его на другое время ожидания. Однако такой код весьма непрост. К счастью, теперь эту работу можно поручить новым функциям пула потоков.

Чтобы какой-то рабочий элемент выполнялся через определенные интервалы времени, первым делом создайте очередь таймеров, вызвав функцию:

```
HANDLE CreateTimerQueue();
```

Очередь таймеров обеспечивает организацию набора таймеров. Представьте, что один исполняемый файл предоставляет несколько сервисов. Каждый сервис может потребовать создания таймеров, скажем, для определения того, какой клиент перестал отвечать, для сбора и обновления некоей статистической информации по расписанию и т. д. Выделять каждому сервису ожидаемый таймер и отдельный поток крайне неэффективно. Вместо этого у каждого сервиса должна быть своя очередь таймеров (занимающая минимум системных ресурсов), а поток компонента поддержки таймера и объект ядра «ожидаемый таймер» должны разделяться всеми сервисами. По окончании работы сервиса его очередь вместе со всеми созданными в ней таймерами просто удаляется.

Вы можете создавать таймеры в очереди, вызывая функцию:

```
BOOL CreateTimerQueueTimer(
    PHANDLE phNewTimer,
    HANDLE hTimerQueue,
    WAITORTIMERCALLBACK pfnCallback,
    PVOID pvContext,
    DWORD dwDueTime,
    DWORD dwPeriod,
    ULONG dwFlags);
```

Во втором параметре Вы передаете описатель очереди, в которую нужно поместить новый таймер. Если таймеров немного, в этом параметре можно передать NULL и вообще не вызывать *CreateTimerQueue*. Такое значение параметра заставит функцию *CreateTimerQueueTimer* использовать очередь по умолчанию и упростит программирование. Параметры *pfnCallback* и *pvContext* указывают на Вашу функцию обратного вызова и данные, передаваемые ей в момент срабатывания таймера. Параметр *dwDueTime* задает время первого срабатывания, а *dwPeriod* — время последующих срабатываний. (Передача в *dwDueTime* нулевого значения заставляет систему вызвать Вашу функцию по возможности немедленно, что делает функцию *CreateTimerQueueTimer* похожей на *QueueUserWorkItem*.) Если *dwPeriod* равен 0, таймер сработает лишь раз, и рабочий элемент будет помещен в очередь только единожды. Описатель нового таймера возвращается в параметре *phNewTimer*.

Прототип Вашей функции обратного вызова должен выглядеть так:

```
VOID WINAPI WaitOrTimerCallback(
    PVOID pvContext,
    BOOL fTimerOrWaitFired);
```

Когда она вызывается, параметр *fTimerOrWaitFired* всегда принимает значение TRUE, сообщая тем самым, что таймер сработал.

Теперь поговорим о параметре *dwFlags* функции *CreateTimerQueueTimer*. Он сообщает функции, как обрабатывать рабочий элемент, помещаемый в очередь. Вы можете указать флаг *WT_EXECUTEDEFAULT*, если хотите, чтобы рабочий элемент был обработан одним из потоков пула, контролируемым компонентом поддержки других операций, *WT_EXECUTEINIOTHREAD* — если в определенный момент нужно выдать

асинхронный запрос на ввод-вывод, или `WT_EXECUTEINPERSISTENTTHREAD` — если элементом должен заняться один из постоянных потоков. Для рабочего элемента, требующего длительного времени обработки, следует задать флаг `WT_EXECUTELONGFUNCTION`.

Вы можете пользоваться еще одним флагом, `WT_EXECUTEINTIMERTHREAD`, который нуждается в более подробном объяснении. Как видно из таблицы 11-1, пул потоков включает компонент поддержки таймера. Этот компонент создает единственный объект ядра «ожидаемый таймер», управляя временем его срабатывания, и всегда состоит из одного потока. Вызывая `CreateTimerQueueTimer`, Вы заставляете его пробудиться, добавить Ваш таймер в очередь и перенастроить объект ядра «ожидаемый таймер». После этого поток компонента поддержки таймера переходит в режим «тревожного» ожидания APC-вызова от таймера. Обнаружив APC-вызов в своей очереди, поток пробуждается, обновляет очередь таймеров, перенастраивает объект ядра «ожидаемый таймер», а затем решает, что делать с рабочим элементом, который теперь следует обработать.

Далее поток проверяет наличие следующих флагов: `WT_EXECUTEDEFAULT`, `WT_EXECUTEINIOTHREAD`, `WT_EXECUTEINPERSISTENTTHREAD`, `WT_EXECUTELONGFUNCTION` и `WT_EXECUTEINTIMERTHREAD`. И сейчас Вы, наверное, поняли, что делает флаг `WT_EXECUTEINTIMERTHREAD`: он заставляет поток компонента поддержки таймера обработать рабочий элемент. Хотя такой механизм обработки элемента более эффективен, он очень опасен! Пока выполняется функция рабочего элемента, поток компонента поддержки таймера ничем другим заниматься не может. Ожидаемый таймер будет по-прежнему ставить APC-вызовы в соответствующую очередь потока, но эти рабочие элементы не удастся обработать до завершения текущей функции. Так что, поток компонента поддержки таймера годится для выполнения лишь «быстрого» кода, не блокирующего этот ресурс надолго.

Флаги `WT_EXECUTEINIOTHREAD`, `WT_EXECUTEINPERSISTENTTHREAD` и `WT_EXECUTEINTIMERTHREAD` являются взаимоисключающими. Если Вы не передаете ни один из этих флагов (или используете `WT_EXECUTEDEFAULT`), рабочий элемент помещается в очередь одного из потоков в компоненте поддержки других операций (не связанных с вводом-выводом). Кроме того, `WT_EXECUTELONGFUNCTION` игнорируется, если задан флаг `WT_EXECUTEINTIMERTHREAD`.

Ненужный таймер удаляется с помощью функции:

```
BOOL DeleteTimerQueueTimer(
    HANDLE hTimerQueue,
    HANDLE hTimer,
    HANDLE hCompletionEvent);
```

Вы должны вызывать ее даже для «одноразовых» таймеров, если они уже сработали. Параметр `hTimerQueue` указывает очередь, в которой находится таймер, а `hTimer` — удаляемый таймер; последний описатель возвращается `CreateTimerQueueTimer` при создании таймера.

Последний параметр, `hCompletionEvent`, определяет, каким образом Вас следует уведомлять об отсутствии необработанных рабочих элементов, поставленных в очередь этим таймером. Если в нем передать `INVALID_HANDLE_VALUE`, функция `DeleteTimerQueueTimer` вернет управление только после обработки всех поставленных в очередь элементов. Задумайтесь, что это значит: удалив таймер в процессе обработки запущенного им рабочего элемента, Вы создаете тупиковую ситуацию, так? Вы ждете окончания его обработки и сами же прерываете ее! Вот почему поток может

удалить таймер, только если это не он обрабатывает рабочий элемент, поставленный в очередь данным таймером.

Кроме того, используя поток компонента поддержки таймера, никогда не удаляйте какой-либо из таймеров во избежание взаимной блокировки. Попытка удалить таймер приводит к тому, что в очередь этого потока помещается APC-уведомление. Но если поток ждет удаления таймера, то сам удалить его он уже не в состоянии — вот и тупик.

Вместо значения `INVALID_HANDLE_VALUE` в параметре `hCompletionEvent` можно передать `NULL`. Это подскажет функции, что таймер следует удалить — и чем раньше, тем лучше. В таком случае `DeleteTimerQueueTimer` немедленно вернет управление, но Вы не узнаете, когда будут обработаны все элементы, поставленные в очередь этим таймером. И, наконец, в параметре `hCompletionEvent` можно передать описатель объекта ядра «событие». Тогда `DeleteTimerQueueTimer` немедленно вернет управление, а поток компонента поддержки таймера освободит событие, как только будут обработаны все элементы из очереди. Но прежде чем вызывать `DeleteTimerQueueTimer`, Вы должны позаботиться о том, чтобы это событие находилось в занятом состоянии, иначе Ваша программа ошибочно решит, что все элементы уже обработаны.

Вы можете изменять время первого и последующих срабатываний существующего таймера, используя функцию:

```
BOOL ChangeTimerQueueTimer(
    HANDLE hTimerQueue,
    HANDLE hTimer,
    ULONG dwDueTime,
    ULONG dwPeriod);
```

Ей передаются описатели очереди и самого таймера, который надо перенастроить, а также параметры `dwDueTime` и `dwPeriod` (время срабатывания и периодичность). Учтите: эта функция не влияет на уже сработавший «одноразовый» таймер. Вы можете применять ее совершенно свободно, без всяких опасений насчет тупиковых ситуаций.

Для удаления очереди таймеров предназначена функция:

```
BOOL DeleteTimerQueueEx(
    HANDLE hTimerQueue,
    HANDLE hCompletionEvent);
```

Она принимает описатель существующей очереди и удаляет все таймеры в ней, избавляя от необходимости вызова `DeleteTimerQueueTimer` для каждого таймера по отдельности. Параметр `hCompletionEvent` идентичен такому же параметру функции `DeleteTimerQueueTimer`, а это значит, что, как и в предыдущем случае, Вы должны помнить о возможности тупиковых ситуаций, — будьте осторожны.

Прежде чем рассматривать следующий вариант, позвольте обратить Ваше внимание на несколько нюансов. Компонент поддержки таймера создает объект ядра «ожидаемый таймер», и тот посылает APC-вызовы в очередь, а не переходит в свободное состояние. Иначе говоря, операционная система постоянно ставит APC-вызовы в очередь, и события таймера никогда не теряются. Такой механизм гарантирует, что написанная Вами функция обратного вызова будет срабатывать с заданной периодичностью. Только имейте в виду, что все это происходит с использованием множества потоков, а значит, какие-то части этой функции, видимо, потребуют синхронизации.

Если Вас это не устраивает и Вы хотите, чтобы новый вызов помещался в очередь, скажем, через 10 секунд после завершения обработки предыдущего, создавайте в конце функции рабочего элемента однократно срабатывающие таймеры. Или единствен-

ный таймер, но с длительным временем ожидания, а в конце все той же функции вызывайте *ChangeTimerQueueTimer* для перенастройки таймера.

Программа-пример TimedMsgBox

Эта программа, «11 TimedMsgBox.exe» (см. листинг на рис. 11-1), показывает, как пользоваться таймерными функциями пула потоков для создания окна, автоматически закрываемого через заданное время в отсутствие реакции пользователя. Файлы исходного кода и ресурсов этой программы находятся в каталоге 11-TimedMsgBox на компакт-диске, прилагаемом к книге.

При запуске программа присваивает глобальной переменной *g_nSecLeft* значение 10. Эта переменная определяет, сколько времени (в секундах) программа ждет реакции пользователя на сообщение, показанное в окне. Далее вызывается *CreateTimerQueueTimer*, настраивающая пул на ежесекундный вызов *MsgBoxTimeout*. Инициализировав все необходимые переменные, программа обращается к *MessageBox* и выводит окно, показанное ниже.



Пока ожидается ответ от пользователя, один из потоков пула каждую секунду вызывает функцию *MsgBoxTimeout*, которая находит описатель этого окна, уменьшает значение глобальной переменной *g_nSecLeft* на 1 и обновляет строку в окне. При первом вызове *MsgBoxTimeout* окно выглядит так:



Десятый вызов *MsgBoxTimeout* обнуляет *g_nSecLeft*, и тогда *MsgBoxTimeout* вызывает *EndDialog*, чтобы закрыть окно. После этого функция *MessageBox*, вызванная первичным потоком, возвращает управление, и вызывается *DeleteTimerQueueTimer*, заставляющая пул прекратить вызовы *MsgBoxTimeout*. В результате открывается другое окно, где сообщается о том, что никаких действий в отведенное время не предпринято.



Если же пользователь успел отреагировать на первое сообщение, на экране появляется то же окно, но с другим текстом.





TimedMsgBox.cpp

```
*****
Модуль: TimedMsgBox.cpp
Автор: Copyright (c) 2000, Джейфри Рихтер (Jeffrey Richter)
*****
```

```
#include "..\CmnHdr.h"      /* см. приложение A */
#include <tchar.h>

//////////



// заголовок нашего окна
TCHAR g_szCaption[] = TEXT("Timed Message Box");

// сколько секунд мы будем показывать это окно
int g_nSecLeft = 0;

// это статический управляющий идентификатор нашего окна
#define ID_MSGBOX_STATIC_TEXT 0x0000ffff

//////////



VOID WINAPI MsgBoxTimeout(PVOID pvContext, BOOLEAN fTimeout) {

    // Примечание: из-за конкуренции потоков возможно (но маловероятно),
    // что окно еще не будет создано, когда мы попадем сюда.
    HWND hwnd = FindWindow(NULL, g_szCaption);

    if (hwnd != NULL) {
        // окно уже существует; сообщаем, сколько времени осталось
        TCHAR sz[100];
        wsprintf(sz, TEXT("You have %d seconds to respond"), g_nSecLeft--);
        SetDlgItemText(hwnd, ID_MSGBOX_STATIC_TEXT, sz);

        if (g_nSecLeft == 0) {
            // время истекло; закрываем окно
            EndDialog(hwnd, IDOK);
        }
    } else {

        // окна пока нет; сейчас ничего не делаем,
        // попробуем через секунду
    }
}

//////////



int WINAPI _tWinMain(HINSTANCE hinstExe, HINSTANCE, PTSTR pszCmdLine, int) {

    chWindows9xNotAllowed();
```

Рис. 11-1. Программа-пример *TimedMsgBox*

см. след. стр.

Рис. 11-1. продолжение

```
// сколько секунд мы даем на ответ
g_nSecLeft = 10;

// создаем многократно срабатывающий таймер
// (первое срабатывание через 1 секунду)
HANDLE hTimerQTimer;
CreateTimerQueueTimer(&hTimerQTimer, NULL, MsgBoxTimeout, NULL,
    1000, 1000, 0);

// выводим окно
MessageBox(NULL, TEXT("You have 10 seconds to respond"), g_szCaption, MB_OK);

// закрываем таймер и удаляем его очередь
DeleteTimerQueueTimer(NULL, hTimerQTimer, NULL);

// выясняем, ответил пользователь или нет
MessageBox(NULL,
    (g_nSecLeft == 0) ? TEXT("Timeout") : TEXT("User responded"),
    TEXT("Result"), MB_OK);

return(0);
}

/////////////////////////////// Конец файла ///////////////////////////////
```

Сценарий 3: вызов функций при освобождении отдельных объектов ядра

Microsoft обнаружила, что во многих приложениях потоки порождаются только для того, чтобы ждать на тех или иных объектах ядра. Как только объект освобождается, поток посыпает уведомление и снова переходит к ожиданию того же объекта. Некоторые разработчики умудряются писать программы так, что в них создается несколько потоков, ждущих один объект. Это невероятное расточительство системных ресурсов. Конечно, издержки от создания потоков существенно меньше, чем от создания процессов, но и потоки не воздухом питаются. У каждого из них свой стек, не говоря уж об огромном количестве команд, выполняемых процессором при создании и уничтожении потока. Поэтому надо стараться сводить любые издержки к минимуму.

Если Вы хотите зарегистрировать рабочий элемент так, чтобы он обрабатывался при освобождении какого-либо объекта ядра, используйте еще одну новую функцию пула потоков:

```
BOOL RegisterWaitForSingleObject(
    PHANDLE phNewWaitObject,
    HANDLE hObject,
    WAITORTIMERCALLBACK pfnCallback,
    PVOID pvContext,
    ULONG dwMilliseconds,
    ULONG dwFlags);
```

Эта функция передает Ваши параметры компоненту поддержки ожидания в пуле потоков. Вы сообщаете ему, что рабочий элемент надо поставить в очередь, как толь-

ко освободится объект ядра (на который указывает *bObject*). Кроме того, Вы можете задать ограничение по времени, т. е. элемент будет помещен в очередь через определенное время, даже если объект ядра так и не освободится. (При этом допустимы значения *INFINITE* и 0.) В общем, эта функция похожа на хорошо известную функцию *WaitForSingleObject* (см. главу 9). Зарегистрировав рабочий элемент на ожидание указанного объекта, *RegisterWaitForSingleObject* возвращает в параметре *phNewWaitObject* описатель, идентифицирующий объект ожидания.

Данный компонент реализует ожидание зарегистрированных объектов через *WaitForMultipleObjects* и поэтому накладывает те же ограничения, что и эта функция. Одно из них заключается в том, что нельзя ожидать тот же объект несколько раз. Так что придется вызывать *DuplicateHandle* и отдельно регистрировать исходный и продублированный описатель. Вам должно быть известно, что единовременно функция *WaitForMultipleObjects* способна отслеживать не более 64 (MAXIMUM_WAIT_OBJECTS) объектов. А что будет, если попробовать зарегистрировать с ее помощью более 64 объектов? Компонент поддержки ожидания создаст еще один поток, который тоже вызовет *WaitForMultipleObjects*. (На самом деле новый поток создается на каждые 63 объекта, потому что потокам приходится использовать объект ядра «ожидаемый таймер», контролирующий таймауты.)

По умолчанию рабочий элемент, готовый к обработке, помещается в очередь к потокам компонента поддержки других операций (не связанных с вводом-выводом). В конечном счете один из его потоков пробудится и вызовет Вашу функцию, у которой должен быть следующий прототип:

```
VOID WINAPI WaitOrTimerCallbackFunc(
    PVOID pvContext,
    BOOLEAN fTimerOrWaitFired);
```

Параметр *fTimerOrWaitFired* принимает значение TRUE, если время ожидания истекло, или FALSE, если объект освободился раньше.

В параметре *dwFlags* функции *RegisterWaitForSingleObject* можно передать флаг *WT_EXECUTEINWAITTHREAD*, который заставляет выполнить функцию рабочего элемента в одном из потоков компонента поддержки ожидания. Это эффективнее, потому что тогда рабочий элемент не придется ставить в очередь компонента поддержки других операций. Но в то же время и опаснее, так как этот поток не сможет ждать освобождения других объектов. Используйте этот флаг, только если Ваша функция выполняется быстро.

Вы можете также передать флаг *WT_EXECUTEINIOTHREAD*, если Ваш рабочий элемент выдаст запрос на асинхронный ввод-вывод, или *WT_EXECUTEINPERSISTENTTHREAD*, если ему понадобится операция с использованием постоянно существующего потока. В случае длительного выполнения функции рабочего элемента можно применить флаг *WT_EXECUTELONGFUNCTION*. Указывайте этот флаг, только если рабочий элемент передается компоненту поддержки ввода-вывода или других операций, — функцию, требующую продолжительной обработки, нельзя выполнять в потоке, который относится к компоненту поддержки ожидания.

И последний флаг, о котором Вы должны знать, — *WT_EXECUTEONLYONCE*. Допустим, Вы зарегистрировались на ожидание объекта ядра «процесс». После перехода в свободное состояние он так и останется в этом состоянии, что заставит компонент поддержки ожидания постоянно включать в очередь рабочие элементы. Так вот, чтобы избежать этого, Вы можете использовать флаг *WT_EXECUTEONLYONCE* — он сообщает пулу потоков прекратить ожидание объекта после первой обработки рабочего элемента.

Теперь представьте, что Вы ждете объект ядра «событие с автосбросом»: сразу после освобождения он тут же возвращается в занятое состояние; при этом в очередь становится соответствующий рабочий элемент. На этом этапе пул продолжает отслеживать объект и снова ждет его освобождения или того момента, когда истечет время, выделенное на ожидание. Если состояние объекта Вас больше не интересует, Вы должны снять его с регистрации. Это необходимо даже для отработавших объектов, зарегистрированных с флагом `WT_EXECUTEONLYONCE`. Вот как выглядит требуемая для этого функция:

```
BOOL UnregisterWaitEx(  
    HANDLE hWaitHandle,  
    HANDLE hCompletionEvent);
```

Первый параметр указывает на объект ожидания (его описатель возвращается `RegisterWaitForSingleObject`), а второй определяет, каким образом Вас следует уведомлять о выполнении последнего элемента в очереди. Как и в `DeleteTimerQueueTimer`, Вы можете передать в этом параметре `NULL` (если уведомление Вас не интересует), `INVALID_HANDLE_VALUE` (функция блокируется до завершения обработки всех элементов в очереди) или описатель объекта-события (переходящего в свободное состояние при завершении обработки очередного элемента). В ответ на неблокирующий вызов `UnregisterWaitEx` возвращает `TRUE`, если очередь пуста, и `FALSE` в ином случае (при этом `GetLastError` возвращает `STATUS_PENDING`).

И вновь будьте осторожны, передавая значение `INVALID_HANDLE_VALUE`. Функция рабочего элемента заблокирует сама себя, если попытается снять с регистрации вызвавший ее объект ожидания. Такая попытка подобна команде: приостановить меня, пока я не закончу выполнение, — полный тупик. Но `UnregisterWaitEx` разработана так, чтобы предотвращать тупиковые ситуации, когда поток компонента поддержки ожидания выполняет рабочий элемент, а тот пытается снять с регистрации запустивший его объект ожидания. И еще один момент: не закрывайте описатель объекта ядра до тех пор, пока не снимете его с регистрации. Иначе недействительный описатель попадет в `WaitForMultipleObjects`, к которой обращается поток компонента поддержки ожидания. Функция моментально завершится с ошибкой, и этот компонент перестанет корректно работать.

И последнее: никогда не вызывайте `PulseEvent` для освобождения объекта-события, зарегистрированного на ожидание. Поток компонента поддержки ожидания скорее всего будет чем-то занят и пропустит этот импульс от `PulseEvent`. Но эта проблема для Вас не нова — `PulseEvent` создает ее почти во всех архитектурах поддержки потоков.

Сценарий 4: вызов функций по завершении запросов на асинхронный ввод-вывод

Последний сценарий самый распространенный. Ваше серверное приложение выдает запросы на асинхронный ввод-вывод, и Вам нужен пул потоков, готовых к их обработке. Это как раз тот случай, на который и были изначально рассчитаны порты завершения ввода-вывода. Если бы Вы управляли собственным пулом потоков, Вы создали бы порт завершения ввода-вывода и пул потоков, ждущих на этом порте. Кроме того, Вы открыли бы пару-тройку устройств ввода-вывода и связали бы их описатели с портом. По мере завершения асинхронных запросов на ввод-вывод, драйверы устройств помещали бы «рабочие элементы» в очередь порта завершения.

Это прекрасная архитектура, позволяющая небольшому количеству потоков эффективно обрабатывать несколько рабочих элементов, и очень хорошо, что она за-

ложена в функции пула потоков. Благодаря этому Вы сэкономите уйму времени и сил. Для использования преимуществ данной архитектуры надо лишь открыть требуемое устройство и сопоставить его с компонентом поддержки других операций (не связанных с вводом-выводом). Учтите, что все потоки в этом компоненте ждут на порте завершения. Чтобы сопоставить устройство с компонентом поддержки других операций, вызовите функцию:

```
BOOL BindIoCompletionCallback(
    HANDLE hDevice,
    POVERLAPPED_COMPLETION_ROUTINE pfnCallback,
    ULONG dwFlags);
```

Эта функция обращается к *CreateIoCompletionPort*, передавая ей *hDevice* и описатель внутреннего порта завершения. Ее вызов также гарантирует, что в компоненте поддержки других операций есть хотя бы один поток. Ключ завершения, сопоставленный с устройством, — это адрес перекрывающейся подпрограммы завершения. Так что, когда ввод-вывод на устройство завершается, компонент пула уже знает, какую функцию надо вызвать для обработки завершенного запроса. У подпрограммы завершения должен быть следующий прототип:

```
VOID WINAPI OverlappedCompletionRoutine(
    DWORD dwErrorCode,
    DWORD dwNumberOfBytesTransferred,
    POVERLAPPED pOverlapped);
```

Заметьте: структура OVERLAPPED передается не в *BindIoCompletionCallback*, а в функции типа *ReadFile* и *WriteFile*. Система внутренне отслеживает эту структуру вместе с запросом на ввод-вывод. После его завершения система поместит адрес структуры в порт завершения для последующей передачи Вашей *OverlappedCompletionRoutine*. А поскольку адрес подпрограммы завершения — это и ключ завершения, то для передачи дополнительной контекстной информации в *OverlappedCompletionRoutine* Вы должны прибегнуть к традиционному трюку и разместить эту информацию в конце структуры OVERLAPPED.

Также учтите, что закрытие устройства приводит к немедленному завершению всех текущих запросов на ввод-вывод и дает ошибку. Будьте готовы к этому в своей функции обратного вызова. Если Вы хотите, чтобы после закрытия устройства функции обратного вызова больше не выполнялись, создайте в своем приложении контрольный счетчик. При выдаче запроса на ввод-вывод Вы будете увеличивать его значение на 1, а при завершении — соответственно уменьшать.

Каких-то специальных флагов для функции *BindIoCompletionCallback* сейчас не предусматривается, поэтому Вы должны передавать 0 в параметре *dwFlags*. Но, по-моему, один флаг, *WT_EXECUTEINIOTHREAD*, ей следовало бы поддерживать. После завершения запроса на ввод-вывод он заставил бы поместить этот запрос в очередь одного из потоков компонента поддержки других операций (не связанных с вводом-выводом). Ведь *OverlappedCompletionRoutine*, вероятно, выдаст еще один запрос на асинхронный ввод-вывод. Однако, если поток завершается, все выданные им запросы на ввод-вывод автоматически уничтожаются. Кроме того, надо учесть, что потоки в компоненте поддержки других операций создаются и уничтожаются в зависимости от текущей нагрузки. При низкой нагрузке поток может быть закрыт, оставив незавершенные запросы. Если бы функция *BindIoCompletionCallback* поддерживала флаг *WT_EXECUTEINIOTHREAD*, то поток, ждущий на порте завершения, мог бы пробудиться и передать результат потоку компонента поддержки ввода-вывода. И поскольку эти

потоки никогда не завершаются при наличии запросов, Вы могли бы выдавать такие запросы, не опасаясь потерять их.

Флаг WT_EXECUTEINIOTHREAD был бы, конечно, очень удобен, но Вы можете легко эмулировать все то, о чем я сейчас говорил. В своей функции *OverlappedCompletionRoutine* просто вызовите *QueueUserWorkItem* с флагом WT_EXECUTEINIOTHREAD и передайте нужные данные (наверное, как минимум, структуру OVERLAPPED). Ничего другого функции пула Вам и не предложили бы.

Волокна

Microsoft добавила в Windows поддержку волокон (*fibers*), чтобы упростить портирование (перенос) существующих серверных приложений из UNIX в Windows. С точки зрения терминологии, принятой в Windows, такие серверные приложения следует считать однопоточными, но способными обслуживать множество клиентов. Иначе говоря, разработчики UNIX-приложений создали свою библиотеку для организации многопоточности и с ее помощью эмулируют истинные потоки. Она создает набор стеков, сохраняет определенные регистры процессора и переключает контексты при обслуживании клиентских запросов.

Разумеется, чтобы добиться большей производительности от таких UNIX-приложений, их следует перепроектировать, заменив библиотеку, эмулирующую потоки, на настоящие потоки, используемые в Windows. Но переработка может занять несколько месяцев, и поэтому компании сначала просто переносят существующий UNIX-код в Windows — это позволяет быстро предложить новый продукт на рынке Windows-приложений.

Но при переносе UNIX-программ в Windows могут возникнуть проблемы. В частности, механизм управления стеком потока в Windows куда сложнее простого выделения памяти. В Windows стеки начинают работать, располагая сравнительно малым объемом физической памяти, и растут по мере необходимости (об этом я расскажу в разделе «Стек потока» главы 16). Перенос усложняется и наличием механизма структурной обработки исключений (см. главы 23, 24 и 25).

Стремясь помочь быстрее (и с меньшим числом ошибок) переносить UNIX-код в Windows, Microsoft добавила в операционную систему механизм поддержки волокон. В этой главе мы рассмотрим концепцию волокон и функции, предназначенные для операций с ними. Кроме того, я покажу, как эффективнее работать с такими функциями. Но, конечно, при разработке новых приложений следует использовать настоящие потоки.

Работа с волокнами

Во-первых, потоки в Windows реализуются на уровне ядра операционной системы, которое отлично осведомлено об их существовании и «коммутирует» их в соответствии с созданным Microsoft алгоритмом. В то же время волокна реализованы на уровне кода пользовательского режима, ядро ничего не знает о них, и процессорное время распределяется между волокнами по алгоритму, определяемому Вами. А раз так, то о вытеснении волокон говорить не приходится — по крайней мере, когда дело касается ядра.

Второе, о чем следует помнить, — в потоке может быть одно или несколько волокон. Для ядра поток — все то, что можно вытеснить и что выполняет код. Единовременно поток будет выполнять код лишь одного волокна — какого, решать Вам (соответствующие концепции я поясню позже).

Приступая к работе с волокнами, прежде всего преобразуйте существующий поток в волокно. Это делает функция *ConvertThreadToFiber*:

```
PVOID ConvertThreadToFiber(PVOID pvParam);
```

Она создает в памяти контекст волокна (размером около 200 байтов). В него входят следующие элементы:

- определенное программистом значение; оно получает значение параметра *pvParam*, передаваемого в *ConvertThreadToFiber*;
- заголовок цепочки структурной обработки исключений;
- начальный и конечный адреса стека волокна; при преобразовании потока в волокно он служит и стеком потока;
- регистры процессора, включая указатели стека и команд.

Создав и инициализировав контекст волокна, Вы сопоставляете его адрес с потоком, преобразованным в волокно, и теперь оно выполняется в этом потоке. *ConvertThreadToFiber* возвращает адрес, по которому расположен контекст волокна. Этот адрес еще понадобится Вам, но ни считывать, ни записывать по нему напрямую ни в коем случае нельзя — с содержимым этой структуры работают только функции, управляющие волокнами. При вызове *ExitThread* завершаются и волокно, и поток.

Нет смысла преобразовывать поток в волокно, если Вы не собираетесь создавать дополнительные волокна в том же потоке. Чтобы создать другое волокно, поток (выполняющий в данный момент волокно), вызывает функцию *CreateFiber*:

```
PVOID CreateFiber(  
    DWORD dwStackSize,  
    PFIBER_START_ROUTINE pfnStartAddress,  
    PVOID pvParam);
```

Сначала она пытается создать новый стек, размер которого задан в параметре *dwStackSize*. Обычно передают 0, и тогда максимальный размер стека ограничивается 1 Мб, а изначально ему передается две страницы памяти. Если Вы укажете ненулевое значение, то для стека будет зарезервирован и передан именно такой объем памяти.

Создав стек, *CreateFiber* формирует новую структуру контекста волокна и инициализирует ее. При этом первый ее элемент получает значение, переданное функции как параметр *pvParam*, сохраняются начальный и конечный адреса стека, а также адрес функции волокна (переданный как аргумент *pfnStartAddress*).

У функции волокна, реализуемой Вами, должен быть такой прототип:

```
VOID WINAPI FiberFunc(PVOID pvParam);
```

Она выполняется, когда волокно впервые получает управление. В качестве параметра ей передается значение, изначально переданное как аргумент *pvParam* функции *CreateFiber*. Внутри функции волокна можно делать что угодно. Обратите внимание на тип возвращаемого значения — VOID. Причина не в том, что это значение несущественно, а в том, что функция никогда не возвращает управление! А иначе поток и все созданные внутри него волокна были бы тут же уничтожены.

Как и *ConvertThreadToFiber*, *CreateFiber* возвращает адрес контекста волокна, но с тем отличием, что новое волокно начинает работать не сразу, поскольку продолжается выполнение текущего. Единовременно поток может выполнять лишь одно волокно. И, чтобы новое волокно стало работать, надо вызвать *SwitchToFiber*:

```
VOID SwitchToFiber(PVOID pvFiberExecutionContext);
```

Эта функция принимает единственный параметр (*pvFiberExecutionContext*) — адрес контекста волокна, полученный в предшествующем вызове *ConvertThreadToFiber* или *CreateFiber*. По этому адресу она определяет, какому волокну предоставить процессорное время. *SwitchToFiber* осуществляет такие операции:

1. Сохраняет в контексте выполняемого в данный момент волокна ряд текущих регистров процессора, включая указатели команд и стека.
2. Загружает в регистры процессора значения, ранее сохраненные в контексте волокна, подлежащего выполнению. В их число входит указатель стека, и поэтому при переключении на другое волокно используется именно его стек.
3. Связывает контекст волокна с потоком, и тот выполняет указанное волокно.
4. Восстанавливает указатель команд. Поток (волокно) продолжает выполнение с того места, на котором волокно было прервано в последний раз.

Применение *SwitchToFiber* — единственный способ выделить волокну процессорное время. Поскольку Ваш код должен явно вызывать эту функцию в нужные моменты, Вы полностью управляете распределением процессорного времени для волокон. Помните: такой вид планирования не имеет ничего общего с планированием потоков. Поток, в рамках которого выполняются волокна, всегда может быть вытеснен операционной системой. Когда поток получает процессорное время, выполняется только выбранное волокно, и никакое другое не получит управление, пока Вы сами не вызовете *SwitchToFiber*.

Для уничтожения волокна предназначена функция *DeleteFiber*:

```
VOID DeleteFiber(PVOID pvFiberExecutionContext);
```

Она удаляет волокно, чей адрес контекста определяется параметром *pvFiberExecutionContext*, освобождает память, занятую стеком волокна, и уничтожает его контекст. Но, если Вы передаете адрес волокна, связанного в данный момент с потоком, *DeleteFiber* сама вызывает *ExitThread* — в результате поток и все созданные в нем волокна «погибают».

DeleteFiber обычно вызывается волокном, чтобы удалить другое волокно. Стек удаляемого волокна уничтожается, а его контекст освобождается. И здесь обратите внимание на разницу между волокнами и потоками: потоки, как правило, уничтожают себя сами, обращаясь к *ExitThread*. Использование с этой целью *TerminateThread* считается плохим тоном — ведь тогда система не уничтожает стек потока. Так вот, способность волокна корректно уничтожать другие волокна нам еще пригодится — как именно, я расскажу, когда мы дойдем до программы-примера.

Для удобства предусмотрено еще две функции, управляющие волокнами. В каждый момент потоком выполняется лишь одно волокно, и операционная система всегда знает, какое волокно связано сейчас с потоком. Чтобы получить адрес контекста текущего волокна, вызовите *GetCurrentFiber*:

```
VOID GetCurrentFiber();
```

Другая полезная функция — *GetFiberData*:

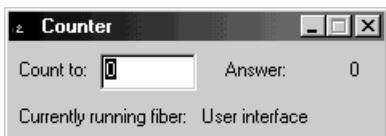
```
VOID GetFiberData();
```

Как я уже говорил, контекст каждого волокна содержит определяемое программистом значение. Оно инициализируется значением параметра *pvParam*, передаваемого функции *ConvertThreadToFiber* или *CreateFiber*, и служит аргументом функции волокна. *GetFiberData* просто «заглядывает» в контекст текущего волокна и возвращает хранящееся там значение.

Обе функции — *GetCurrentFiber* и *GetFiberData* — работают очень быстро и обычно реализуются компилятором как встраиваемые (т. е. вместо вызовов этих функций он подставляет их код).

Программа-пример Counter

Эта программа, «12 Counter.exe» (см. листинг на рис. 12-1), демонстрирует применение волокон для реализации фоновой обработки. Запустив ее, Вы увидите диалоговое окно, показанное ниже. (Настоятельно советую запустить программу Counter; тогда Вам будет легче понять, что происходит в ней и как она себя ведет.)



Считайте эту программу сверхминиатюрной электронной таблицей, состоящей всего из двух ячеек. В первую из них можно записывать — она реализована как поле, расположенное за меткой Count To. Вторая ячейка доступна только для чтения и реализована как статический элемент управления, размещенный за меткой Answer. Изменив число в поле, Вы заставите программу пересчитать значение в ячейке Answer. В этом простом примере пересчет заключается в том, что счетчик, начальное значение которого равно 0, постепенно увеличивается до максимума, заданного в ячейке Count To. Для наглядности статический элемент управления, расположенный в нижней части диалогового окна, показывает, какое из волокон — пользовательского интерфейса или расчетное — выполняется в данный момент.

Чтобы протестировать программу, введите в поле число 5 — строка Currently Running Fiber будет заменена строкой Recalculation, а значение в поле Answer постепенно возрастет с 0 до 5. Когда пересчет закончится, текущим волокном вновь станет интерфейсное, а поток заснет. Теперь введите число 50 и вновь понаблюдайте за пересчетом — на этот раз перемещая окно по экрану. При этом Вы заметите, что расчетное волокно вытесняется, а интерфейсное вновь получает процессорное время, благодаря чему программа продолжает реагировать на действия пользователя. Оставьте окно в покое, и Вы увидите, что расчетное волокно снова получило управление и возобновило работу с того значения, на котором было прервано.

Остается проверить лишь одно. Давайте изменим число в поле ввода в момент пересчета. Заметьте: интерфейс отреагировал на Ваши действия, но после ввода данных пересчет начинается заново. Таким образом, программа ведет себя как настоящая электронная таблица.

Обратите внимание и на то, что в программе не задействованы ни критические секции, ни другие объекты, синхронизирующие потоки, — все сделано на основе двух волокон в одном потоке.

Теперь обсудим внутреннюю реализацию программы Counter. Когда первый поток процесса приступает к выполнению *_tWinMain*, вызывается функция *ConvertThreadToFiber*, преобразующая поток в волокно, которое впоследствии позволит нам создать другое волокно. Затем мы создаем немодальное диалоговое окно, выступающее в роли главного окна программы. Далее инициализируем переменную — индикатор состояния фоновой обработки (background processing state, BPS). Она реализована как элемент *bps* в глобальной переменной *g_FiberInfo*. Ее возможные состояния описываются в следующей таблице.

Состояние	Описание
BPS_DONE	Пересчет завершен, пользователь ничего не изменял, новый пересчет не нужен
BPS_STARTOVER	Пользователь внес изменения, требуется пересчет с самого начала
BPS_CONTINUE	Пересчет еще продолжается, пользователь ничего не изменял, пересчет заново не нужен

Индикатор *bps* проверяется внутри цикла обработки сообщений потока, который здесь сложнее обычного. Вот что делает этот цикл.

- Если поступает оконное сообщение (активен пользовательский интерфейс), обрабатываем именно его. Своевременная обработка действий пользователя всегда приоритетнее пересчета.
- Если пользовательский интерфейс простоявает, проверяем, не нужен ли пересчет (т. е. не присвоено ли переменной *bps* значение BPS_STARTOVER или BPS_CONTINUE).
- Если вычисления не нужны (BPS_DONE), приостанавливаем поток, вызывая *WaitMessage*, — только событие, связанное с пользовательским интерфейсом, может потребовать пересчета.

Если интерфейсному волокну делать нечего, а пользователь только что изменил значение в поле ввода, начинаем вычисления заново (BPS_STARTOVER). Главное, о чем здесь надо помнить, — волокно, отвечающее за пересчет, может уже работать. Тогда это волокно следует удалить и создать новое, которое начнет все с начала. Чтобы уничтожить выполняющее пересчет волокно, интерфейсное вызывает *DeleteFiber*. Именно этим и удобны волокна. Удаление волокна, занятого пересчетом, — операция вполне допустимая, стек волокна и его контекст корректно уничтожаются. Если бы мы использовали потоки, а не волокна, интерфейсный поток не смог бы корректно уничтожить поток, занятый пересчетом, — нам пришлось бы задействовать какой-нибудь механизм межпоточного взаимодействия и ждать, пока поток пересчета не завершится сам. Зная, что волокна, отвечающего за пересчет, больше нет, мы вправе создать новое волокно для тех же целей, присвоив переменной *bps* значение BPS_CONTINUE.

Когда пользовательский интерфейс простоявает, а волокно пересчета чем-то занято, мы выделяем ему процессорное время, вызывая *SwitchToFiber*. Последняя не вернет управление, пока волокно пересчета тоже не обратится к *SwitchToFiber*, передав ей адрес контекста интерфейсного волокна.

FiberFunc является функцией волокна и содержит код, выполняемый волокном пересчета. Ей передается адрес глобальной структуры *g_FiberInfo*, и поэтому она знает описатель диалогового окна, адрес контекста интерфейсного волокна и текущее состояние индикатора фоновой обработки. Конечно, раз это глобальная переменная, то передавать ее адрес как параметр не обязательно, но я решил показать, как в функцию волокна передаются параметры. Кроме того, передача адресов позволяет добиться того, чтобы код меньше зависел от конкретных переменных, — именно к этому и следует стремиться.

Первое, что делает функция волокна, — обновляет диалоговое окно, сообщая, что сейчас выполняется волокно пересчета. Далее функция получает значение, введенное в поле, и запускает цикл, считающий от 0 до указанного значения. Перед каждым приращением счетчика вызывается *GetQueueStatus* — эта функция проверяет, не по-

явились ли в очереди потока новые сообщения. (Все волокна, работающие в рамках одного потока, делят его очередь сообщений.) Если сообщение появилось, значит, интерфейсному волокну есть чем заняться, и мы, считая его приоритетным по отношению к расчетному, сразу же вызываем *SwitchToFiber*, давая ему возможность обработать поступившее сообщение. Когда сообщение (или сообщения) будет обработано, интерфейсное волокно передаст управление волокну, отвечающему за пересчет, и фоновая обработка возобновится.

Если сообщений нет, расчетное волокно обновляет поле *Answer* диалогового окна и засыпает на 200 мс. В коде настоящей программы вызов *Sleep* надо, естественно, убрать — я поставил его, только чтобы «потянуть» время.

Когда волокно, отвечающее за пересчет, завершает свою работу, статус фоновой обработки устанавливается как *BPS_DONE*, и управление передается (через *SwitchToFiber*) интерфейсному волокну. В этот момент ему делать нечего, и оно вызывает *WaitMessage*, которая приостанавливает поток, чтобы не тратить процессорное время понапрасну.



Counter.cpp

```

/*
Модуль: Counter.cpp
Автор: Copyright (c) 2000, Джейфри Рихтер (Jeffrey Richter)
*/

#include "..\CmnHdr.h"      /* см. приложение A */
#include <WindowsX.h>
#include <tchar.h>
#include "Resource.h"

///////////////////////////////



// возможные состояния фоновой обработки
typedef enum {
    BPS_STARTOVER,      // начинаем фоновую обработку заново
    BPS_CONTINUE,       // продолжаем фоновую обработку
    BPS_DONE            // фоновая обработка не нужна
} BKNDPROCSTATE;

typedef struct {
    PVOID pFiberUI;        // контекст интерфейсного волокна
    HWND hwnd;             // описатель главного окна
    BKNDPROCSTATE bps;    // состояние фоновой обработки
} FIBERINFO, *PFIBERINFO;

// глобальная переменная, содержащая информацию о состоянии приложения;
// из интерфейсного волокна она доступна напрямую, а из волокна, отвечающего
// за фоновую обработку, - косвенно
FIBERINFO g_FiberInfo;

///////////////////////////////

```

Рис. 12-1. Программа-пример *Counter*

Рис. 12-1. продолжение

```

void WINAPI FiberFunc(PVOID pvParam) {
    PFIBERINFO pFiberInfo = (PFIBERINFO) pvParam;

    // показываем в окне, какое волокно выполняется сейчас
    SetDlgItemText(pFiberInfo->hwnd, IDC_FIBER, TEXT("Recalculation"));

    // получаем текущее значение из поля ввода
    int nCount = GetDlgItemInt(pFiberInfo->hwnd, IDC_COUNT, NULL, FALSE);

    // считаем от 0 до nCount, обновляя статический элемент управления
    for (int x = 0; x <= nCount; x++) {

        // события пользовательского интерфейса приоритетнее расчетов
        // (если такие события есть, обрабатываем их)
        if (HIWORD(GetQueueStatus(QS_ALLEVENTS)) != 0) {

            // интерфейсное волокно чем-то занято; временно приостанавливаем
            // пересчет и обрабатываем события пользовательского интерфейса
            SwitchToFiber(pFiberInfo->pFiberUI);

            // событий больше нет, возобновляем пересчет
            SetDlgItemText(pFiberInfo->hwnd, IDC_FIBER, TEXT("Recalculation"));
        }

        // обновляем статический элемент управления,
        // показывая последнее значение счетчика
        SetDlgItemInt(pFiberInfo->hwnd, IDC_ANSWER, x, FALSE);

        // засыпаем, чтобы "потянуть" время;
        // в рабочем коде вызов Sleep надо убрать
        Sleep(200);
    }

    // сообщаем, что пересчет закончен
    pFiberInfo->bps = BPS_DONE;

    // Выделяем процессорное время интерфейсному волокну. Если
    // событий, подлежащих обработке, нет, приостанавливаем его.
    // Примечание: если разрешить возврат из функции волокна,
    // поток и интерфейсное волокно завершатся, а мы этого не хотим!
    SwitchToFiber(pFiberInfo->pFiberUI);
}

///////////////////////////////
BOOL Dlg_OnInitDialog(HWND hwnd, HWND hwndFocus, LPARAM lParam) {

    chSETDLGICONS(hwnd, IDI_COUNTER);
    SetDlgItemInt(hwnd, IDC_COUNT, 0, FALSE);
    return(TRUE);
}

```

см. след. стр.

Рис. 12-1. продолжение

```
//////////  
void Dlg_OnCommand(HWND hwnd, int id, HWND hwndCtl, UINT codeNotify) {  
  
    switch (id) {  
        case IDCANCEL:  
            PostQuitMessage(0);  
            break;  
  
        case IDC_COUNT:  
            if (codeNotify == EN_CHANGE) {  
                // пользователь изменил значение счетчика,  
                // начинаем фоновую обработку заново  
                g_FiberInfo.bps = BPS_STARTOVER;  
            }  
            break;  
    }  
}  
//////////  
  
INT_PTR WINAPI DlgProc(HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam) {  
  
    switch (uMsg) {  
        chHANDLE_DLGMMSG(hwnd, WM_INITDIALOG, Dlg_OnInitDialog);  
        chHANDLE_DLGMMSG(hwnd, WM_COMMAND, Dlg_OnCommand);  
    }  
    return(FALSE);  
}  
//////////  
  
int _tWinMain(HINSTANCE hinstExe, HINSTANCE, PTSTR pszCmdLine, int) {  
  
    // контекст волокна, отвечающего за пересчет  
    PVOID pFiberCounter = NULL;  
  
    // преобразуем поток в волокно  
    g_FiberInfo.pFiberUI = ConvertThreadToFiber(NULL);  
  
    // создаем окно программы  
    g_FiberInfo.hwnd = CreateDialog(hinstExe, MAKEINTRESOURCE(IDD_COUNTER),  
        NULL, Dlg_Proc);  
  
    // обновляем окно, показывая, какое волокно  
    // выполняется в данный момент  
    SetDlgItemText(g_FiberInfo.hwnd, IDC_FIBER, TEXT("User interface"));  
  
    // изначально фоновая обработка отсутствует  
    g_FiberInfo.bps = BPS_DONE;
```

Рис. 12-1. продолжение

```

// пока пользовательское окно существует...
BOOL fQuit = FALSE;
while (!fQuit) {

    // интерфейсные сообщения приоритетнее фоновой обработки
    MSG msg;
    if (PeekMessage(&msg, NULL, 0, 0, PM_REMOVE)) {

        // если в очереди есть сообщение, обрабатываем его
        fQuit = (msg.message == WM_QUIT);
        if (!IsDialogMessage(g_FiberInfo.hwnd, &msg)) {
            TranslateMessage(&msg);
            DispatchMessage(&msg);
        }
    }

    // если сообщений нет, проверяем состояние фоновой обработки
    switch (g_FiberInfo.bps) {
        case BPS_DONE:
            // нет необходимости в фоновой обработке;
            // ждем событие, связанное с пользовательским интерфейсом
            WaitMessage();
            break;

        case BPS_STARTOVER:
            // пользователь изменил счетчик,
            // начинаем фоновую обработку заново

            if (pFiberCounter != NULL) {
                // если волокно, отвечающее за пересчет, уже существует,
                // удаляем его, и фоновая обработка начнется заново
                DeleteFiber(pFiberCounter);
                pFiberCounter = NULL;
            }

            // создаем новое волокно для пересчета, которое
            // начнет все с начала
            pFiberCounter = CreateFiber(0, FiberFunc, &g_FiberInfo);

            // фоновая обработка началась, ее нужно продолжать
            g_FiberInfo.bps = BPS_CONTINUE;

            // переходим к BPS_CONTINUE...

        case BPS_CONTINUE:
            // переключаемся на волокно пересчета...
            SwitchToFiber(pFiberCounter);

            // фоновая обработка приостановлена из-за появления сообщения от
            // пользовательского интерфейса или потому, что вычисления закончены
    }
}

```

см. след. стр.

Рис. 12-1. продолжение

```
// обновляем окно, показывая, какое волокно
// выполняется в данный момент
SetDlgItemText(g_FiberInfo.hwnd,
    IDC_FIBER, TEXT("User interface"));

if (g_FiberInfo.bps == BPS_DONE) {
    // фоновая обработка закончена, удаляем
    // это волокно, чтобы в следующий раз фоновая
    // обработка началась заново
    DeleteFiber(pFiberCounter);
    pFiberCounter = NULL;
}
break;
} // конец оператора switch
} // сообщений от пользовательского интерфейса нет
} // while (окно существует)
DestroyWindow(g_FiberInfo.hwnd);

return(0); // программа завершена
}

/////////////////////////////// Конец файла //////////////////////
```

ЧАСТЬ III

УПРАВЛЕНИЕ ПАМЯТЬЮ



Архитектура памяти в Windows

Архитектура памяти, используемая в операционной системе, — ключ к пониманию того, как система делает то, что она делает. Когда начинаешь работать с новой операционной системой, всегда возникает масса вопросов. Как разделить данные между двумя приложениями? Где хранится та или иная информация? Как оптимизировать свою программу? Список вопросов можно продолжить.

Обычно знание того, как система управляет памятью, упрощает и ускоряет поиск ответов на эти вопросы. Поэтому здесь мы рассмотрим архитектуру памяти, применяемую в Microsoft Windows.

Виртуальное адресное пространство процесса

Каждому процессу выделяется собственное виртуальное адресное пространство. Для 32-разрядных процессов его размер составляет 4 Гб. Соответственно 32-битный указатель может быть любым числом от 0x00000000 до 0xFFFFFFFF. Всего, таким образом, указатель может принимать 4 294 967 296 значений, что как раз и перекрывает четырехгигабайтовый диапазон. Для 64-разрядных процессов размер адресного пространства равен 16 экзабайтам, поскольку 64-битный указатель может быть любым числом от 0x00000000 00000000 до 0xFFFFFFFF FFFFFFFF и принимать 18 446 744 073 709 551 616 значений, охватывая диапазон в 16 экзабайтов. Весьма впечатляюще!

Поскольку каждому процессу отводится закрытое адресное пространство, то, когда в процессе выполняется какой-нибудь поток, он получает доступ только к той памяти, которая принадлежит его процессу. Память, отведенная другим процессам, скрыта от этого потока и недоступна ему.



В Windows 2000 память, принадлежащая собственно операционной системе, тоже скрыта от любого выполняемого потока. Иными словами, ни один поток не может случайно повредить ее данные. А в Windows 98 последнее, увы, не реализовано, и есть вероятность, что выполняемый поток, случайно получив доступ к данным операционной системы, тем самым нарушит ее нормальную работу. И все-таки в Windows 98, как и в Windows 2000, ни один поток не может получить доступ к памяти чужого процесса.

Итак, как я уже говорил, адресное пространство процесса закрыто. Отсюда вытекает, что процесс A в своем адресном пространстве может хранить какую-то структуру данных по адресу 0x12345678, и одновременно у процесса B по тому же адресу — но уже в его адресном пространстве — может находиться совершенно иная структура данных. Обращаясь к памяти по адресу 0x12345678, потоки, выполняемые в процессе A, получают доступ к структуре данных процесса A. Но, когда по тому же адресу

обращаются потоки, выполняемые в процессе В, они получают доступ к структуре данных процесса В. Иначе говоря, потоки процесса А не могут обратиться к структуре данных в адресном пространстве процесса В, и наоборот.

А теперь, пока Вы не перевозбудились от колоссального объема адресного пространства, предоставляемого Вашей программе, вспомните, что оно — *виртуальное*, а не физическое. Другими словами, адресное пространство — всего лишь диапазон адресов памяти. И, прежде чем Вы сможете обратиться к каким-либо данным, не вызывав нарушения доступа, придется спроектировать нужную часть адресного пространства на конкретный участок физической памяти. (Об этом мы поговорим чуть позже.)

Как адресное пространство разбивается на разделы

Виртуальное адресное пространство каждого процесса разбивается на разделы. Их размер и назначение в какой-то мере зависят от конкретного ядра Windows (таблица 13-1).

Как видите, ядра 32- и 64-разрядной Windows 2000 создают разделы, почти одинаковые по назначению, но отличающиеся по размеру и расположению. Однако ядро Windows 98 формирует другие разделы. Давайте рассмотрим, как система использует каждый из этих разделов.

Раздел	32-разрядная Windows 2000 (на x86 и Alpha)	32-разрядная Windows 2000 (на x86 с ключом /3GB)	64-разрядная Windows 2000 (на Alpha и IA-64)	Windows 98
Для выявления нулевых указателей	0x00000000 0x0000FFFF	0x00000000 0x0000FFFF	0x00000000 00000000 0x00000000 0000FFFF	0x00000000 0x00000FFF
Для совместимости с программами DOS и 16-разрядной Windows	Нет	Нет	Нет	0x00001000 0x003FFFFF
Для кода и данных пользовательского режима	0x00010000 0x7FFEFFFF	0x00010000 0xBFFEFFFF	0x00000000 00010000 0x000003FF FFFEFFFF	0x00400000 0x7FFFFFFF
Закрытый, размером 64 Кб	0x7FFF0000 0x7FFFFFFF	0xBFFF0000 0xBFFFFFFF	0x000003FF FFFF0000 0x000003FF FFFFFFFF	Нет
Для общих MMF (файлов, проецируемых в память)	Нет	Нет	Нет	0x80000000 0xBFFFFFFF
Для кода и данных режима ядра	0x8000000000 0xFFFFFFFF	0xC0000000 0xFFFFFFFF	0x00000400 00000000 0xFFFFFFFF FFFFFFFF	0xC0000000 0xFFFFFFFF

Таблица 13-1. Так адресное пространство процесса разбивается на разделы



Microsoft активно работает над 64-разрядной Windows 2000. На момент написания книги эта система все еще находилась в разработке. Информацию по 64-разрядной Windows 2000 следует учитывать при проектировании и реализации текущих проектов. Однако Вы должны понимать, что какие-то детали скорее всего изменятся к моменту выхода 64-разрядной Windows 2000. То же самое относится и к конкретным диапазонам разделов виртуального адресного пространства и размеру страниц памяти на процессорах IA-64 (64-разрядной архитектуры Intel).

Раздел для выявления нулевых указателей (Windows 2000 и Windows 98)

Этот раздел адресного пространства резервируется для того, чтобы программисты могли выявлять нулевые указатели. Любая попытка чтения или записи в память по этим адресам вызывает нарушение доступа.

Довольно часто в программах, написанных на C/C++, отсутствует скрупулезная обработка ошибок. Например, в следующем фрагменте кода такой обработки вообще нет:

```
int* pnSomeInteger = (int*) malloc(sizeof(int));  
*pnSomeInteger = 5;
```

При нехватке памяти *malloc* вернет NULL. Но код не учитывает эту возможность и при ошибке обратится к памяти по адресу 0x00000000. А поскольку этот раздел адресного пространства заблокирован, возникнет нарушение доступа и данный процесс завершится. Эта особенность помогает программистам находить «жучков» в своих приложениях.

Раздел для совместимости с программами DOS и 16-разрядной Windows (только Windows 98)

Этот регион размером 4 Мб в адресном пространстве процесса необходим Windows 98 для поддержки совместимости с программами MS-DOS и 16-разрядной Windows. Не пытайтесь обращаться к нему из 32-разрядных Windows-приложений. В идеале процессор должен был бы генерировать нарушение доступа при обращении потока к этому участку адресного пространства, но по техническим причинам Microsoft не смогла заблокировать эти 4 Мб адресного пространства.

В Windows 2000 программы для MS-DOS и 16-разрядной Windows выполняются в собственных адресных пространствах; 32-разрядные приложения повлиять на них не могут.

Раздел для кода и данных пользовательского режима (Windows 2000 и Windows 98)

В этом разделе располагается закрытая (неразделяемая) часть адресного пространства процесса. Ни один процесс не может получить доступ к данным другого процесса, размещенным в этом разделе. Основной объем данных, принадлежащих процессу, хранится именно здесь (это касается всех приложений). Поэтому приложения менее зависимы от взаимных «капризов», и вся система функционирует устойчивее.

WINDOWS 2000 В Windows 2000 сюда загружаются все EXE- и DLL-модули. В каждом процессе эти DLL можно загружать по разным адресам в пределах данного раздела, но так делается крайне редко. На этот же раздел отображаются все проецируемые в память файлы, доступные данному процессу.

WINDOWS 98 В Windows 98 основные 32-разрядные системные DLL (Kernel32.dll, AdvAPI32.dll, User32.dll и GDI32.dll) загружаются в раздел для общих MMF (проецируемых в память файлов), а EXE- и остальные DLL-модули — в раздел для кода и данных пользовательского режима. Общие DLL располагаются по одному и тому же виртуальному адресу во всех процессах, но другие DLL могут загружать их (общие DLL) по разным адресам в границах раздела для кода и данных пользовательского режима.

тельского режима (хотя это маловероятно). Проецируемые в память файлы в этот раздел никогда не помещаются.

Впервые увидев адресное пространство своего 32-разрядного процесса, я был удивлен тем, что его полезный объем чуть ли не вдвое меньше. Неужели раздел для кода и данных режима ядра должен занимать столько места? Оказывается — да. Это пространство нужно системе для кода ядра, драйверов устройств, кэш-буферов ввода-вывода, областей памяти, не сбрасываемых в файл подкачки, таблиц, используемых для контроля страниц памяти в процессе и т. д. По сути, Microsoft едва-едва втиснула ядро в эти виртуальные два гигабайта. В 64-разрядной Windows 2000 ядро наконец получит то пространство, которое ему нужно на самом деле.

Увеличение раздела для кода и данных пользовательского режима до 3 Гб на процессорах x86 (только Windows 2000)

Многие разработчики уже давно сетовали на нехватку адресного пространства для пользовательского режима. Microsoft пошла навстречу и предусмотрела в версиях Windows 2000 Advanced Server и Windows 2000 Data Center для процессоров x86 возможность увеличения этого пространства до 3 Гб. Чтобы все процессы использовали раздел для кода и данных пользовательского режима размером 3 Гб, а раздел для кода и данных режима ядра — объемом 1 Гб, Вы должны добавить ключ /3GB к нужной записи в системном файле Boot.ini. Как выглядит адресное пространство процесса в этом случае, показано в графике «32-разрядная Windows 2000 (на x86 с ключом /3GB)» таблицы 13-1.

Раньше, когда такого ключа не было, программа не видела адресов памяти по указателю с установленным старшим битом. Некоторые изобретательные разработчики самостоятельно использовали этот бит как флаг, который имел смысл только в их приложениях. При обращении программы по адресам за пределами 2 Гб предварительно выполнялся специальный код, который сбрасывал старший бит указателя. Но, как Вы понимаете, когда приложение на свой страх и риск создает себе трехгигабайтовую среду пользовательского режима, оно может с треском рухнуть.

Microsoft пришлось придумать решение, которое позволило бы подобным приложениям работать в трехгигабайтовой среде. Теперь система в момент запуска приложения проверяет, не скомпоновано ли оно с ключом /LARGEADDRESSAWARE. Если да, приложение как бы заявляет, что обязуется корректно обращаться с этими адресами памяти и действительно готово к использованию трехгигабайтowego адресного пространства пользовательского режима. А если нет, операционная система резервирует область памяти размером 1 Гб в диапазоне адресов от 0x80000000 до 0xBFFFFFFF. Это предотвращает выделение памяти по адресам с установленным старшим битом.

Заметьте, что ядро и так с трудом умещается в двухгигабайтовом разделе. Но при использовании ключа /3GB ядру остается всего 1 Гб. Тем самым уменьшается количество потоков, стеков и других ресурсов, которые система могла бы предоставить приложению. Кроме того, система в этом случае способна задействовать максимум 16 Гб оперативной памяти против 64 Гб в нормальных условиях — из-за нехватки виртуального адресного пространства для кода и данных режима ядра, необходимого для управления дополнительной оперативной памятью.



Флаг LARGEADDRESSAWARE в исполняемом файле проверяется в тот момент, когда операционная система создает адресное пространство процесса. Для DLL этот флаг игнорируется. При написании DLL Вы должны *сами* позаботиться об их корректном поведении в трехгигабайтовом разделе пользовательского режима.

Уменьшение раздела для кода и данных пользовательского режима до 2 Гб в 64-разрядной Windows 2000

Microsoft понимает, что многие разработчики захотят как можно быстрее перенести свои 32-разрядные приложения в 64-разрядную среду. Но в исходном коде любых программ полно таких мест, где предполагается, что указатели являются 32-разрядными значениями. Простая перекомпиляция исходного кода приведет к ошибочно-мому усечению указателей и некорректному обращению к памяти.

Однако, если бы система как-то гарантировала, что память никогда не будет выделяться по адресам выше 0x00000000 7FFFFFFF, приложение работало бы нормально. И усечение 64-разрядного адреса до 32-разрядного, когда старшие 33 бита равны 0, не создало бы никаких проблем. Так вот, система дает такую гарантию при запуске приложения в «адресной песочнице» (address space sandbox), которая ограничивает полезное адресное пространство процесса до нижних 2 Гб.

По умолчанию, когда Вы запускаете 64-разрядное приложение, система резервирует все адресное пространство пользовательского режима, начиная с 0x00000000 80000000, что обеспечивает выделение памяти исключительно в нижних 2 Гб 64-разрядного адресного пространства. Это и есть «адресная песочница». Большинству приложений этого пространства более чем достаточно. А чтобы 64-разрядное приложение могло адресоваться ко всему разделу пользовательского режима (объемом 4 Тб), его следует скомпоновать с ключом /LARGEADDRESSAWARE.



Флаг LARGEADDRESSAWARE в исполняемом файле проверяется в тот момент, когда операционная система создает адресное пространство 64-разрядного процесса. Для DLL этот флаг игнорируется. При написании DLL Вы должны *сами* позаботиться об их корректном поведении в четырехтерабайтовом разделе пользовательского режима.

Закрытый раздел размером 64 Кб (только Windows 2000)

Этот раздел заблокирован, и любая попытка обращения к нему приводит к нарушению доступа. Microsoft резервирует этот раздел специально, чтобы упростить внутреннюю реализацию операционной системы. Вспомните: когда Вы передаете Windows-функции адрес блока памяти и его размер, то она (функция), прежде чем приступить к работе, проверяет, действителен ли данный блок. Допустим, Вы написали код:

```
BYTE bBuf[70000];
DWORD dwNumBytesWritten;
WriteProcessMemory(GetCurrentProcess(), (PVOID) 0x7FFEEE90, bBuf,
    sizeof(bBuf), &dwNumBytesWritten);
```

В случае функций типа *WriteProcessMemory* область памяти, в которую предполагается запись, проверяется кодом, работающим в режиме ядра, — только он имеет право обращаться к памяти, выделяемой под код и данные режима ядра (в 32-разрядных системах — по адресам выше 0x80000000). Если по этому адресу есть память, вызов *WriteProcessMemory*, показанный выше, благополучно запишет данные в ту область памяти, которая, по идее, доступна только коду, работающему в режиме ядра. Чтобы предотвратить это и в то же время ускорить проверку таких областей памяти, Microsoft предпочла заблокировать данный раздел, и поэтому любая попытка чтения или записи в нем всегда вызывает нарушение доступа.

Раздел для общих MMF (только Windows 98)

В этом разделе размером 1 Гб система хранит данные, разделяемые всеми 32-разрядными процессами. Сюда, например, загружаются все системные DLL (Kernel32.dll, AdvAPI32.dll, User32.dll и GDI32.dll), и поэтому они доступны любому 32-разрядному процессу. Кроме того, эти DLL загружаются в каждом процессе по одному и тому же адресу памяти. На этот раздел система также отображает все проецируемые в память файлы. Об этих файлах мы поговорим в главе 17.

Раздел для кода и данных режима ядра (Windows 2000 и Windows 98)

В этот раздел помещается код операционной системы, в том числе драйверы устройств и код низкоуровневого управления потоками, памятью, файловой системой, сетевой поддержкой. Все, что находится здесь, доступно любому процессу. В Windows 2000 эти компоненты полностью защищены. Поток, который попытается обратиться по одному из адресов памяти в этом разделе, вызовет нарушение доступа, а это приведет к тому, что система в конечном счете просто закроет его приложение. (Подробнее на эту тему см. главы 23, 24 и 25.)

WINDOWS 2000 В 64-разрядной Windows 2000 раздел пользовательского режима (4 Тб) выглядит непропорционально малым по сравнению с 16 777 212 Тб, отведенными под раздел для кода и данных режима ядра. Дело не в том, что ядру так уж необходимо все это виртуальное пространство, а просто 64-разрядное адресное пространство настолько огромно, что его большая часть не задействована. Система разрешает нашим программам использовать 4 Тб, а ядру — столько, сколько ему нужно. К счастью, какие-либо внутренние структуры данных для управления незадействованными частями раздела для кода и данных режима ядра не требуются.

WINDOWS 98 В Windows 98 данные, размещенные в этом разделе, увы, не защищены — любое приложение может что-то считать или записать в нем и нарушить нормальную работу операционной системы.

Регионы в адресном пространстве

Адресное пространство, выделяемое процессу в момент создания, практически все *свободно* (незарезервировано). Поэтому, чтобы воспользоваться какой-нибудь его частью, нужно выделить в нем определенные регионы через функцию *VirtualAlloc* (о ней — в главе 15). Операция выделения региона называется *резервированием* (*reserving*).

При резервировании система обязательно выравнивает начало региона с учетом так называемой *гранулярности выделения памяти* (*allocation granularity*). Последняя величина в принципе зависит от типа процессора, но для процессоров, рассматриваемых в книге (*x86*, 32- и 64-разрядных *Alpha* и *IA-64*), — она одинакова и составляет 64 Кб.

Резервируя регион в адресном пространстве, система обеспечивает еще и кратность размера региона размеру *страницы*. Так называется единица объема памяти, используемая системой при управлении памятью. Как и гранулярность выделения ресурсов, размер страницы зависит от типа процессора. В частности, для процессоров *x86* он равен 4 Кб, а для *Alpha* (под управлением как 32-разрядной, так и 64-разрядной Windows 2000) — 8 Кб. На момент написания книги предполагалось, что *IA-64*

тоже будет работать со страницами размером 8 Кб. Однако в зависимости от результатов тестирования Microsoft может выбрать для него страницы большего размера (от 16 Кб).



Иногда система сама резервирует некоторые регионы адресного пространства в интересах Вашего процесса, например, для *хранения блока переменных окружения процесса* (process environment block, PEB). Этот блок — небольшая структура данных, создаваемая, контролируемая и разрушающаяся исключительно операционной системой. Выделение региона под PEB-блок осуществляется в момент создания процесса.

Кроме того, для управления потоками, существующими на данный момент в процессе, система создает блоки *переменных окружения потоков* (thread environment blocks, TEBs). Регионы под эти блоки резервируются и освобождаются по мере создания и разрушения потоков в процессе.

Но, требуя от Вас резервировать регионы с учетом гранулярности выделения памяти (а эта гранулярность на сегодняшний день составляет 64 Кб), сама система этих правил не придерживается. Поэтому вполне вероятно, что границы региона, зарезервированного под PEB- и TEB-блоки, не будут кратны 64 Кб. Тем не менее размер такого региона обязательно кратен размеру страниц, характерному для данного типа процессора.

Если Вы попытаетесь зарезервировать регион размером 10 Кб, система автоматически округлит заданное Вами значение до большей кратной величины. А это значит, что на x86 будет выделен регион размером 12 Кб, а на Alpha — 16 Кб.

И последнее в этой связи. Когда зарезервированный регион адресного пространства становится не нужен, его следует вернуть в общие ресурсы системы. Эта операция — *освобождение* (releasing) региона — осуществляется вызовом функции *VirtualFree*.

Передача региону физической памяти

Чтобы зарезервированный регион адресного пространства можно было использовать, Вы должны выделить физическую память и спроектировать ее на этот регион. Такая операция называется *передачей физической памяти* (committing physical storage). Чтобы передать физическую память зарезервированному региону, Вы обращаетесь все к той же функции *VirtualAlloc*.

Передавая физическую память регионам, нет нужды отводить ее целому региону. Можно, скажем, зарезервировать регион размером 64 Кб и передать физическую память только его второй и четвертой страницам. На рис. 13-1 представлен пример того, как может выглядеть адресное пространство процесса. Как видите, структура адресного пространства зависит от архитектуры процессора. Слева показано, что происходит с адресным пространством на процессоре x86 (страницы по 4 Кб), а справа — на процессоре Alpha (страницы по 8 Кб).

Когда физическая память, переданная зарезервированному региону, больше не нужна, ее освобождают. Эта операция — *возврат физической памяти* (decommitting physical storage) — выполняется вызовом функции *VirtualFree*.

Физическая память и страницный файл

В старых операционных системах физической памятью считалась вся оперативная память (RAM), установленная в компьютере. Иначе говоря, если в Вашей машине было 16 Мб оперативной памяти, Вы могли загружать и выполнять приложения, используя

ющие вплоть до 16 Мб памяти. Современные операционные системы умеют имитировать память за счет дискового пространства. При этом на диске создается страничный файл (paging file), который и содержит виртуальную память, доступную всем процессам.

Разумеется, операции с виртуальной памятью требуют соответствующей поддержки от самого процессора. Когда поток пытается обратиться к какому-то байту, процессор должен знать, где находится этот байт — в оперативной памяти или на диске.

С точки зрения прикладной программы, страничный файл просто увеличивает объем памяти, которой она может пользоваться. Если в Вашей машине установлено 64 Мб оперативной памяти, а размер страничного файла на жестком диске составляет 100 Мб, приложение считает, что объем оперативной памяти равен 164 Мб.

Конечно, 164 Мб оперативной памяти у Вас на самом деле нет. Операционная система в тесной координации с процессором сбрасывает содержимое части оперативной памяти в страничный файл и по мере необходимости подгружает его порции обратно в память. Если такого файла нет, система просто считает, что приложениям доступен меньший объем памяти, — вот и все. Но, поскольку страничный файл явным образом увеличивает объем памяти, доступный приложениям, его применение весьма желательно. Это позволяет приложениям работать с большими наборами данных.

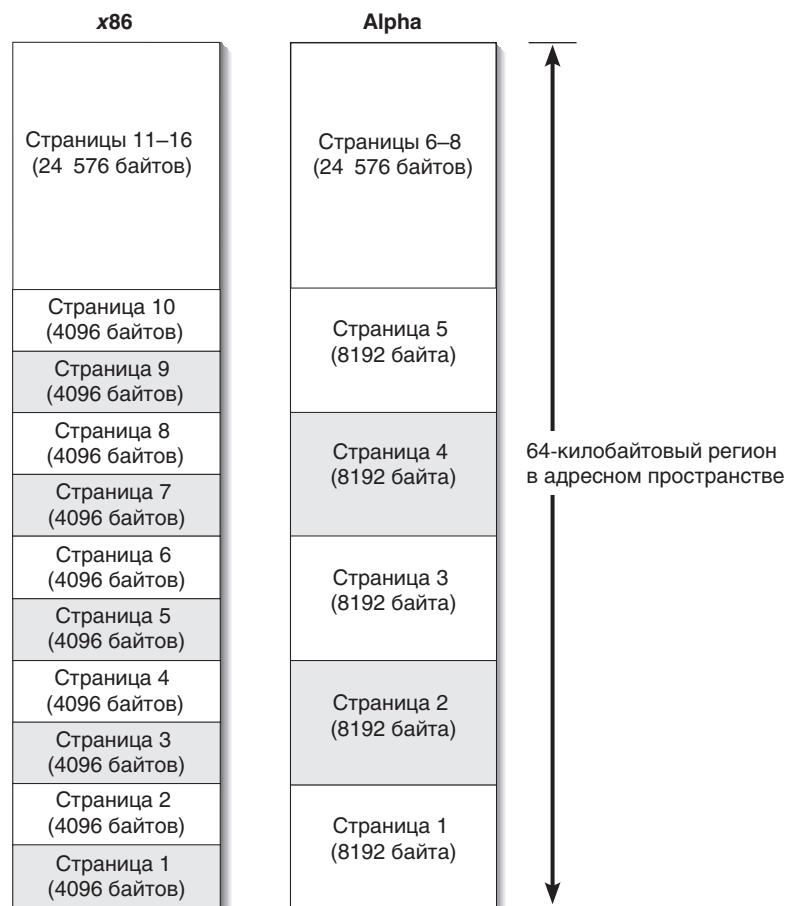


Рис. 13-1. Примеры адресных пространств процессов для разных типов процессоров

Физическую память следует рассматривать как данные, хранимые в дисковом файле со страничной структурой. Поэтому, когда приложение передает физическую память какому-нибудь региону адресного пространства (вызывая *VirtualAlloc*), она на самом деле выделяется из файла, размещенного на жестком диске. Размер страничного файла в системе — главный фактор, определяющий количество физической памяти, доступное приложению. Реальный объем оперативной памяти имеет гораздо меньшее значение.

Теперь посмотрим, что происходит, когда поток пытается получить доступ к блоку данных в адресном пространстве своего процесса. А произойти может одно из двух (рис. 13-2).

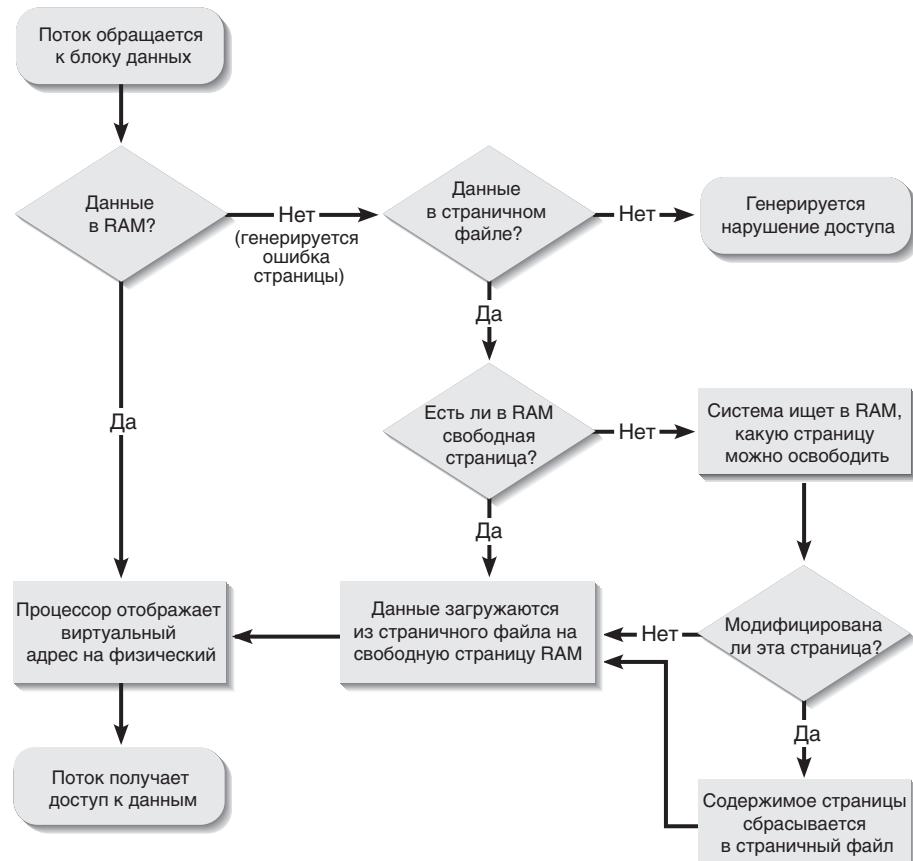


Рис. 13-2. Трансляция виртуального адреса на физический

В первом сценарии данные, к которым обращается поток, находятся в оперативной памяти. В этом случае процессор проецирует виртуальный адрес данных на физический, и поток получает доступ к нужным ему данным.

Во втором сценарии данные, к которым обращается поток, отсутствуют в оперативной памяти, но размещены где-то в страничном файле. Попытка доступа к данным генерирует ошибку страницы (page fault), и процессор таким образом уведомляет операционную систему об этой попытке. Тогда операционная система начинает искать свободную страницу в оперативной памяти; если таковой нет, система вынуждена освободить одну из занятых страниц. Если занятая страница не модифицировалась, она просто освобождается; в ином случае она сначала копируется из оператив-

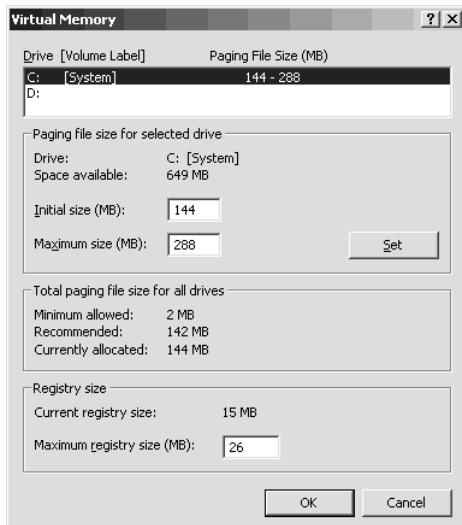
ной памяти в страничный файл. После этого система переходит к страничному файлу, отыскивает в нем запрошенный блок данных, загружает этот блок на свободную страницу оперативной памяти и, наконец, отображает (проецирует) адрес данных в виртуальной памяти на соответствующий адрес в физической памяти.

Чем чаще системе приходится копировать страницы памяти в страничный файл и наоборот, тем больше нагрузка на жесткий диск и тем медленнее работает операционная система. (При этом может получиться так, что операционная система будет тратить все свое время на подкачку страниц вместо выполнения программ.) Поэтому, добавив компьютеру оперативной памяти, Вы снизите частоту обращения к жесткому диску и тем самым увеличите общую производительность системы. Кстати, во многих случаях увеличение оперативной памяти дает больший выигрыш в производительности, чем замена старого процессора на новый.

Физическая память в страничном файле не хранится

Прочитав предыдущий раздел, Вы, должно быть, подумали, что страничный файл сильно разбухнет при одновременном выполнении в системе нескольких программ, — особенно если Вы сочли, будто при каждом запуске приложения система резервирует регионы адресного пространства для кода и данных процесса, передает им физическую память, а затем копирует код и данные из файла программы (расположенного на жестком диске) в физическую память, переданную из страничного файла.

WINDOWS 2000 Windows 2000 может использовать несколько страничных файлов, и, если они расположены на разных физических дисках, операционная система работает гораздо быстрее, поскольку способна вести запись одновременно на нескольких дисках. Чтобы добавить или удалить страничный файл, откройте в Control Panel апплет System, выберите вкладку Advanced и щелкните кнопку Performance Options. На экране появится следующее диалоговое окно:



Однако система действует не так, иначе на загрузку и подготовку программы к запуску уходило бы слишком много времени. На самом деле происходит вот что: при запуске приложения система открывает его исполняемый файл и определяет объем кода и данных. Затем резервирует регион адресного пространства и помечает, что

физическая память, связанная с этим регионом, — сам EXE-файл. Да-да, правильно: вместо выделения какого-то пространства из страничного файла система использует истинное содержимое, или *образ* (*image*) EXE-файла как зарезервированный регион адресного пространства программы. Благодаря этому приложение загружается очень быстро, а размер страничного файла удается заметно уменьшить.

Образ исполняемого файла (т. е. EXE- или DLL-файл), размещенный на жестком диске и применяемый как физическая память для того или иного региона адресного пространства, называется *проецируемым в память файлом* (*memory-mapped file*). При загрузке EXE или DLL система автоматически резервирует регион адресного пространства и проецирует на него образ файла. Помимо этого, система позволяет (с помощью набора функций) проецировать на регион адресного пространства еще и файлы данных. (О проецируемых в память файлах мы поговорим в главе 17.)



Когда EXE- или DLL-файл загружается с дискеты, Windows 98 и Windows 2000 целиком копируют его в оперативную память, а в страничном файле выделяют такое пространство, чтобы в нем мог уместиться образ загружаемого файла. Если нагрузка на оперативную память в системе невелика, EXE- или DLL-файл всегда запускается непосредственно из оперативной памяти.

Так сделано для корректной работы программ установки. Обычно программа установки запускается с первой дискеты, потом поочередно вставляются следующие диски, на которых собственно и содержится устанавливаемое приложение. Если системе понадобится какой-то фрагмент кода EXE- или DLL-модуля программы установки, на текущей дискете его, конечно же, нет. Но, поскольку система скопировала файл в оперативную память (и предусмотрела для него место в страничном файле), у нее не возникнет проблем с доступом к нужной части кода программы установки.

Система не копирует в оперативную память образы файлов, хранящихся на других съемных носителях (CD-ROM или сетевых дисках), если только требуемый файл не скомпонован с использованием ключа /SWAPRUN:CD или /SWAPRUN:NET. (Имейте в виду, что Windows 98 не поддерживает флаги SWAPRUN.)

Атрибуты защиты

Отдельным страницам физической памяти можно присвоить свои атрибуты защиты, показанные в следующей таблице.

Атрибут защиты	Описание
PAGE_NOACCESS	Попытки чтения, записи или исполнения содержимого памяти на этой странице вызывают нарушение доступа
PAGE_READONLY	Попытки записи или исполнения содержимого памяти на этой странице вызывают нарушение доступа
PAGE_READWRITE	Попытки исполнения содержимого памяти на этой странице вызывают нарушение доступа
PAGE_EXECUTE	Попытки чтения или записи на этой странице вызывают нарушение доступа
PAGE_EXECUTE_READ	Попытки записи на этой странице вызывают нарушение доступа
PAGE_EXECUTE_READWRITE	На этой странице возможны любые операции

продолжение

Атрибут защиты	Описание
PAGE_WRITECOPY	Попытки исполнения содержимого памяти на этой странице вызывают нарушение доступа; попытка записи приводит к тому, что процессу предоставляется «личная» копия данной страницы
PAGE_EXECUTE_WRITECOPY	На этой странице возможны любые операции; попытка записи приводит к тому, что процессу предоставляется «личная» копия данной страницы

На процессорных платформах *x86* и Alpha атрибут PAGE_EXECUTE не поддерживается, хотя в операционных системах такая поддержка предусмотрена. Перечисленные процессоры воспринимают запрос на чтение как запрос на исполнение. Поэтому присвоение памяти атрибута PAGE_EXECUTE приводит к тому, что на этих процессорах она считается доступной и для чтения. Но полагаться на эту особенность не стоит, поскольку в реализациях Windows на других процессорах все может встать на свои места.

WINDOWS 98 В Windows 98 страницам физической памяти можно присвоить только три атрибута защиты: PAGE_NOACCESS, PAGE_READONLY и PAGE_READWRITE.

Защита типа «копирование при записи»

Атрибуты защиты, перечисленные в предыдущей таблице, достаточно понятны, кроме двух последних: PAGE_WRITECOPY и PAGE_EXECUTE_WRITECOPY. Они предназначены специально для экономного расходования оперативной памяти и места в страницном файле. Windows поддерживает механизм, позволяющий двум и более процессам разделять один и тот же блок памяти. Например, если Вы запустите 10 экземпляров программы Notepad, все экземпляры будут совместно использовать одни и те же страницы с кодом и данными этой программы. И обычно никаких проблем не возникает — пока процессы ничего не записывают в общие блоки памяти. Только представьте, что творилось бы в системе, если потоки из разных процессов начали бы одновременно записывать в один и тот же блок памяти!

Чтобы предотвратить этот хаос, операционная система присваивает общему блоку памяти атрибут защиты «копирование при записи» (copy-on-write). Когда поток в одном процессе попытается что-нибудь записать в общий блок памяти, в дело тут же вступит система и проделает следующие операции:

1. Найдет свободную страницу в оперативной памяти. Заметьте, что при первом проецировании модуля на адресное пространство процесса эта страница будет скопирована на одну из страниц, выделенных в страницном файле. Поскольку система выделяет нужное пространство в страницном файле еще при первом проецировании модуля, сбои на этом этапе маловероятны.
2. Скопирует страницу с данными, которые поток пытается записать в общий блок памяти, на свободную страницу оперативной памяти, полученную на этапе 1. Последней присваивается атрибут защиты PAGE_WRITECOPY или PAGE_EXECUTE_WRITECOPY. Атрибут защиты и содержимое исходной страницы не меняются.
3. Отобразит виртуальный адрес этой страницы в процессе на новую страницу в оперативной памяти.

Когда система выполнит эти операции, процесс получит свою копию нужной страницы памяти. Подробнее о совместном использовании памяти и о защите типа «копирование при записи» я расскажу в главе 17.

Кроме того, при резервировании адресного пространства или передаче физической памяти через *VirtualAlloc* нельзя указывать атрибуты PAGE_WRITECOPY или PAGE_EXECUTE_WRITECOPY. Иначе вызов *VirtualAlloc* даст ошибку, а *GetLastError* вернет код ERROR_INVALID_PARAMETER. Дело в том, что эти два атрибута используются операционной системой, только когда она проецирует образы EXE- или DLL-файлов.

WINDOWS 98 Windows 98 не поддерживает «копирование при записи». Обнаружив запрос на применение такой защиты, Windows 98 тут же делает копии данных, не дождаясь попытки записи в память.

Специальные флаги атрибутов защиты

Кроме рассмотренных атрибутов защиты, существует три флага атрибутов защиты: PAGE_NOCACHE, PAGE_WRITECOMBINE и PAGE_GUARD. Они комбинируются с любыми атрибутами защиты (кроме PAGE_NOACCESS) побитовой операцией OR.

Флаг PAGE_NOCACHE отключает кэширование переданных страниц. Как правило, использовать этот флаг не рекомендуется; он предусмотрен главным образом для разработчиков драйверов устройств, которым нужно манипулировать буферами памяти.

Флаг PAGE_WRITECOMBINE тоже предназначен для разработчиков драйверов устройств. Он позволяет объединять несколько операций записи на устройство в один пакет, что увеличивает скорость передачи данных.

Флаг PAGE_GUARD позволяет приложениям получать уведомление (через механизм исключений) в тот момент, когда на страницу записывается какой-нибудь байт. Windows 2000 использует этот флаг при создании стека потока. Подробнее на эту тему см. раздел «Стек потока» в главе 16.

WINDOWS 98 Windows 98 игнорирует флаги атрибутов защиты PAGE_NOCACHE, PAGE_WRITECOMBINE и PAGE_GUARD.

Подводя итоги

А теперь попробуем осмыслить понятия адресных пространств, разделов, регионов, блоков и страниц как единое целое. Лучше всего начать с изучения карты виртуальной памяти, на которой изображены все регионы адресного пространства в пределах одного процесса. В качестве примера мы воспользуемся программой VMMap из главы 14. Чтобы в полной мере разобраться в адресном пространстве процесса, рассмотрим его в том виде, в каком оно формируется при запуске VMMap под управлением Windows 2000 на 32-разрядной процессорной платформе x86. Образец карты адресного пространства VMMap показан в таблице 13-2. На отличиях адресных пространств в Windows 2000 и Windows 98 я остановлюсь чуть позже.

Карта в таблице 13-2 показывает регионы, расположенные в адресном пространстве процесса. Каждому региону соответствует своя строка в таблице, а каждая строка состоит из шести полей.

В первом (крайнем слева) поле приводится базовый адрес региона. Наверное, Вы заметили, что просмотр адресного пространства мы начали с региона по адресу 0x00000000 и закончили последним регионом используемого адресного простран-

ства, который начинается по адресу 0x7FFE0000. Все регионы непрерывны. Почти все базовые адреса занятых регионов начинаются со значений, кратных 64 Кб. Это связано с гранулярностью выделения памяти в адресном пространстве. А если Вы увидите какой-нибудь регион, начало которого не выровнено по значению, кратному 64 Кб, значит, он выделен кодом операционной системы для управления Вашим процессом.

Базовый адрес	Тип	Размер	Блоки	Атрибут(ы) защиты	Описание
00000000	Free	65536			
00010000	Private	4096	1	-RW-	
00011000	Free	61440			
00020000	Private	4096	1	-RW-	
00021000	Free	61440			
00030000	Private	1048576	3	-RW-	Стек потока
00130000	Private	1048576	2	-RW-	
00230000	Mapped	65536	2	-RW-	
00240000	Mapped	90112	1	-R--	\Device\HarddiskVolume1\WINNT\system32\unicode.nls
00256000	Free	40960			
00260000	Mapped	208896	1	-R--	\Device\HarddiskVolume1\WINNT\system32\locale.nls
00293000	Free	53248			
002A0000	Mapped	266240	1	-R--	\Device\HarddiskVolume1\WINNT\system32\sortkey.nls
002E1000	Free	61440			
002F0000	Mapped	16384	1	-R--	\Device\HarddiskVolume1\WINNT\system32\sorttbls.nls
002F4000	Free	49152			
00300000	Mapped	819200	4	ER--	
003C8000	Free	229376			
00400000	Image	106496	5	ERWC	C:\CD\x86\Debug\14 VMMap.exe
0041A000	Free	24576			
00420000	Mapped	274432	1	-R--	
00463000	Free	53248			
00470000	Mapped	3145728	2	ER--	
00770000	Private	4096	1	-RW-	
00771000	Free	61440			
00780000	Private	4096	1	-RW-	
00781000	Free	61440			
00790000	Private	65536	2	-RW-	
007A0000	Mapped	8192	1	-R--	\Device\HarddiskVolume1\WINNT\system32\ctype.nls
007A2000	Free	1763893248			
699D0000	Image	45056	4	ERWC	C:\WINNT\System32\PSAPI.dll
699DB000	Free	238505984			
77D50000	Image	450560	4	ERWC	C:\WINNT\system32\RPCRT4.DLL
77DBE000	Free	8192			
77DC0000	Image	344064	5	ERWC	C:\WINNT\system32\ADVAPI32.dll
77E14000	Free	49152			
77E20000	Image	401408	4	ERWC	C:\WINNT\system32\USER32.dll

Таблица 13-2. Образец карты адресного пространства процесса в Windows 2000 на 32-разрядном процессоре типа x86

Таблица 13-2. продолжение

Базовый адрес	Тип	Размер	Блоки	Атрибут(ы) защиты	Описание
77E82000	Free	57344			
77E90000	Image	720896	5	ERWC	C:\WINNT\system32\KERNEL32.dll
77F40000	Image	241664	4	ERWC	C:\WINNT\system32\GDI32.DLL
77F7B000	Free	20480			
77F80000	Image	483328	5	ERWC	C:\WINNT\System32\ntdll.dll
77FF6000	Free	40960			
78000000	Image	290816	6	ERWC	C:\WINNT\system32\MSVCRT.dll
78047000	Free	124424192			
7F6F0000	Mapped	1048576	2	ER--	
7F7F0000	Free	8126464			
7FFB0000	Mapped	147456	1	-R--	
7FFD4000	Free	40960			
7FFDE000	Private	4096	1	ERW-	
7FFDF000	Private	4096	1	ERW-	
7FFE0000	Private	65536	2	-R--	

Во втором поле показывается тип региона: Free (свободный), Private (закрытый), Image (образ) или Mapped (проецируемый). Эти типы описаны в следующей таблице.

Тип	Описание
Free	Этот диапазон виртуальных адресов не сопоставлен ни с каким типом физической памяти. Его адресное пространство не зарезервировано; приложение может зарезервировать регион по указанному базовому адресу или в любом месте в границах свободного региона.
Private	Этот диапазон виртуальных адресов сопоставлен со страничным файлом.
Image	Этот диапазон виртуальных адресов изначально был сопоставлен с образом EXE- или DLL-файла, проецируемого в память, но теперь, возможно, уже нет. Например, при записи в глобальную переменную из образа модуля механизм поддержки «копирования при записи» выделяет соответствующую страницу памяти из страничного файла, а не исходного образа файла.
Mapped	Этот диапазон виртуальных адресов изначально был сопоставлен с файлом данных, проецируемым в память, но теперь, возможно, уже нет. Например, файл данных мог быть спроектирован с использованием механизма поддержки «копирования при записи». Любые операции записи в этот файл приведут к тому, что соответствующие страницы памяти будут выделены из страничного файла, а не из исходного файла данных.

Способ вычисления этого поля моей программой VMMap может давать неправильные результаты. Поясню почему. Когда регион занят, VMMap пытается «прикинуть», к какому из трех оставшихся типов он может относиться, — в Windows нет функций, способных подсказать точное предназначение региона. Я определяю это сканированием всех блоков в границах исследуемого региона, по результатам которого программа делает обоснованное предположение. Но предположение есть предположение. Если Вы хотите получше разобраться в том, как это делается, просмотрите исходный код VMMap, приведенный в главе 14.

В третьем поле сообщается размер региона в байтах. Например, система спроектировала образ User32.dll по адресу 0x77E20000. Когда она резервировала адресное

пространство для этого образа, ей понадобилось 401 408 байтов. Не забудьте, что в третьем поле всегда содержатся значения, кратные размеру страницы, характерному для данного процессора (4096 байтов для *x86*).

В четвертом поле показано количество блоков в зарезервированном регионе. Блок — это неразрывная группа страниц с одинаковыми атрибутами защиты, связанная с одним и тем же типом физической памяти (подробнее об этом мы поговорим в следующем разделе). Для свободных регионов это значение всегда равно 0, так как им не передается физическая память. (Поэтому в четвертой графе никаких данных для свободных регионов не приводится.) Но для занятых регионов это значение может колебаться в пределах от 1 до максимума (его вычисляют делением размера региона на размер страницы). Скажем, у региона, начинающегося с адреса 0x77E20000, размер — 401 408 байтов. Поскольку процесс выполняется на процессоре *x86* (страницы памяти по 4096 байтов), максимальное количество блоков в этом регионе равно 98 (401 408/4096); ну а, судя по карте, в нем содержится 4 блока.

В пятом поле — атрибуты защиты региона. Здесь используются следующие сокращения: *E* = execute (исполнение), *R* = read (чтение), *W* = write (запись), *C* = copy-on-write (копирование при записи). Если ни один из атрибутов в этой графе не указан, регион доступен без ограничений. Атрибуты защиты не присваиваются и свободным регионам. Кроме того, здесь Вы никогда не увидите флагов атрибутов защиты PAGE_GUARD или PAGE_NOCACHE — они имеют смысл только для физической памяти, а не для зарезервированного адресного пространства. Атрибуты защиты присваиваются регионам только эффективности ради и всегда замещаются атрибутами защиты, присвоенными физической памяти.

В шестом (и последнем) поле кратко описывается содержимое текущего региона. Для свободных регионов оно всегда пустое, а для закрытых — обычно пустое, так как у VMMap нет возможности выяснить, зачем приложение зарезервировало данный закрытый регион. Однако VMMap все же распознает назначение тех закрытых регионов, в которых содержатся стеки потоков. Стеки потоков выдают себя тем, что содержат блок физической памяти с флагом атрибутов защиты PAGE_GUARD. Если же стек полностью заполнен, такого блока у него нет, и тогда VMMap не в состоянии распознать стек потока.

Для регионов типа Image программе VMMap удается определить полное имя файла, проецируемого на этот регион. Она получает эту информацию с помощью Tool-Help-функций, о которых я упоминал в конце главы 4. В Windows 2000 программа VMMap может идентифицировать регионы, сопоставленные с файлами данных; для этого она вызывает функцию *GetMappedFileName* (ее нет в Windows 98).

Блоки внутри регионов

Попробуем увеличить детализацию адресного пространства (по сравнению с тем, что показано в таблице 13-2). Например, таблица 13-3 показывает ту же карту адресного пространства, но в другом «масштабе»: по ней можно узнать, из каких блоков состоит каждый регион.

Базовый адрес	Тип	Размер	Блоки	Атрибут(ы) защиты	Описание
00000000	Free	65536			
00010000	Private	4096	1	-RW-	
00010000	Private	4096		-RW- ---	

Таблица 13-3. Образец карты адресного пространства процесса (с указанием блоков внутри регионов) в Windows 2000 на 32-разрядном процессоре типа *x86*

Таблица 13-3. продолжение

Базовый адрес	Тип	Размер	Блоки	Атрибут(ы) защиты	Описание
00011000	Free	61440			
00020000	Private	4096	1	-RW-	
00020000	Private	4096		-RW- ---	
00021000	Free	61440			
00030000	Private	1048576	3	-RW-	Стек потока
00030000	Reserve	905216		-RW- ---	
0010D000	Private	4096		-RW- G--	
0010E000	Private	139264		-RW- ---	
00130000	Private	1048576	2	-RW-	
00130000	Private	36864		-RW- ---	
00139000	Reserve	1011712		-RW- ---	
00230000	Mapped	65536	2	-RW-	
00230000	Mapped	4096		-RW- ---	
00231000	Reserve	61440		-RW- ---	
00240000	Mapped	90112	1	-R--	\Device\HddiskVolume1\WINNT\system32\unicode.nls
00240000	Mapped	90112		-R-- ---	
00256000	Free	40960			
00260000	Mapped	208896	1	-R--	\Device\HddiskVolume1\WINNT\system32\locale.nls
00260000	Mapped	208896		-R-- ---	
00293000	Free	53248			
002A0000	Mapped	266240	1	-R--	\Device\HddiskVolume1\WINNT\system32\sortkey.nls
002A0000	Mapped	266240		-R-- ---	
002E1000	Free	61440			
002F0000	Mapped	16384	1	-R--	\Device\HddiskVolume1\WINNT\system32\sorttbls.nls
002F0000	Mapped	16384		-R-- ---	
002F4000	Free	49152			
00300000	Mapped	819200	4	ER--	
00300000	Mapped	16384		ER-- ---	
00304000	Reserve	770048		ER-- ---	
003C0000	Mapped	8192		ER-- ---	
003C2000	Reserve	24576		ER-- ---	
003C8000	Free	229376			
00400000	Image	106496	5	ERWC	C:\CD\x86\Debug\14 VMMap.exe
00400000	Image	4096		-R-- ---	
00401000	Image	81920		ER-- ---	
00415000	Image	4096		-R-- ---	
00416000	Image	8192		-RW- ---	
00418000	Image	8192		-R-- ---	
0041A000	Free	24576			
00420000	Mapped	274432	1	-R--	
00420000	Mapped	274432		-R-- ---	
00463000	Free	53248			
00470000	Mapped	3145728	2	ER--	

Таблица 13-3. продолжение

Базовый адрес	Тип	Размер	Блоки	Атрибут(ы) защиты	Описание
00470000	Mapped	274432		ER-- ---	
004B3000	Reserve	2871296		ER-- ---	
00770000	Private	4096	1	-RW-	
00770000	Private	4096		-RW- ---	
00771000	Free	61440			
00780000	Private	4096	1	-RW-	
00780000	Private	4096		-RW- ---	
00781000	Free	61440			
00790000	Private	65536	2	-RW-	
00790000	Private	20480		-RW- ---	
00795000	Reserve	45056		-RW- ---	
007A0000	Mapped	8192	1	-R--	\Device\HarddiskVolume1\WINNT\system32\ctype.nls
007A0000	Mapped	8192		-R-- ---	
007A2000	Free	57344			
007B0000	Private	524288	2	-RW-	
007B0000	Private	4096		-RW- ---	
007B1000	Reserve	520192		-RW- ---	
00830000	Free	1763311616			
699D0000	Image	45056	4	ERWC	C:\WINNT\System32\PSAPI.dll
699D0000	Image	4096		-R-- ---	
699D1000	Image	16384		ER-- ---	
699D5000	Image	16384		-RWC ---	
699D9000	Image	8192		-R-- ---	
699DB000	Free	238505984			
77D50000	Image	450560	4	ERWC	C:\WINNT\system32\RPCRT4.DLL
77D50000	Image	4096		-R-- ---	
77D51000	Image	421888		ER-- ---	
77DB8000	Image	4096		-RW- ---	
77DB9000	Image	20480		-R-- ---	
77DBE000	Free	8192			
77DC0000	Image	344064	5	ERWC	C:\WINNT\system32\ADVAPI32.dll
77DC0000	Image	4096		-R-- ---	
77DC1000	Image	307200		ER-- ---	
77E0C000	Image	4096		-RW- ---	
77E0D000	Image	4096		-RWC ---	
77E0E000	Image	24576		-R-- ---	
77E14000	Free	49152			
77E20000	Image	401408	4	ERWC	C:\WINNT\system32\USER32.dll
77E20000	Image	4096		-R-- ---	
77E21000	Image	348160		ER-- ---	
77E76000	Image	4096		-RW- ---	
77E77000	Image	45056		-R-- ---	
77E82000	Free	57344			

Таблица 13-3. продолжение

Базовый адрес	Тип	Размер	Блоки	Атрибут(ы) защиты	Описание
77E90000	Image	720896	5	ERWC	C:\WINNT\system32\KERNEL32.dll
77E90000	Image	4096		-R-- ---	
77E91000	Image	368640		ER-- ---	
77EEB000	Image	8192		-RW- ---	
77EED000	Image	4096		-RWC ---	
77EEE000	Image	335872		-R-- ---	
77F40000	Image	241664	4	ERWC	C:\WINNT\system32\GDI32.DLL
77F40000	Image	4096		-R-- ---	
77F41000	Image	221184		ER-- ---	
77F77000	Image	4096		-RW- ---	
77F78000	Image	12288		-R-- ---	
77F7B000	Free	20480			
77F80000	Image	483328	5	ERWC	C:\WINT\System32\ntdll.dll
77F80000	Image	4096		-R-- ---	
77F81000	Image	299008		ER-- ---	
77FCA000	Image	8192		-RW- ---	
77FCC000	Image	4096		-RWC ---	
77FCD000	Image	167936		-R-- ---	
77FF6000	Free	40960			
78000000	Image	290816	6	ERWC	C:\WINNT\system32\MSVCRT.dll
78000000	Image	4096		-R-- ---	
78001000	Image	208896		ER-- ---	
78034000	Image	32768		-R-- ---	
7803C000	Image	12288		-RW- ---	
7803F000	Image	16384		RWC- ---	
78043000	Image	16384		-R-- ---	
78047000	Free	124424192			
7F6F0000	Mapped	1048576	2	ER--	
7F6F0000	Mapped	28672		ER-- ---	
7F6F7000	Reserve	1019904		ER-- ---	
7F7F0000	Free	8126464			
7FFB0000	Mapped	147456	1	-R--	
7FFB0000	Mapped	147456		-R-- ---	
7FFD4000	Free	40960			
7FFDE000	Private	4096	1	ERW-	
7FFDE000	Private	4096		ERW- ---	
7FFDF000	Private	4096	1	ERW-	
7FFDF000	Private	4096		ERW- ---	
7FFE0000	Private	65536	2	-R--	
7FFE0000	Private	4096		-R-- ---	
7FFE1000	Reserve	61440		-R-- ---	

Разумеется, в свободных регионах блоков нет, поскольку им не переданы страници физической памяти. Строки с описанием блоков состоят из пяти полей.

В первом поле показывается адрес группы страниц с одинаковыми состоянием и атрибутами защиты. Например, по адресу 0x77E20000 передана единственная страница (4096 байтов) физической памяти с атрибутом защиты, разрешающим только чтение. А по адресу 0x77E21000 присутствует блок размером 85 страниц (348 160 байтов) переданной памяти с атрибутами, разрешающими и чтение, и исполнение. Если бы атрибуты защиты этих блоков совпадали, их можно было бы объединить, и тогда на карте памяти появился бы единый элемент размером в 86 страниц (352 256 байтов).

Во втором поле сообщается тип физической памяти, с которой связан тот или иной блок, расположенный в границах зарезервированного региона. В нем появляется одно из пяти возможных значений: Free (свободный), Private (закрытый), Mapped (проецируемый), Image (образ) или Reserve (резервный). Значения Private, Mapped и Image говорят о том, что блок поддерживается физической памятью соответственно из страницного файла, файла данных, загруженного EXE- или DLL-модуля. Если же в поле указано значение Free или Reserve, блок вообще не связан с физической памятью.

Чаще всего блоки в пределах одного региона связаны с однотипной физической памятью. Однако регион вполне может содержать несколько блоков, связанных с физической памятью разных типов. Например, образ файла, проецируемого в память, может быть связан с EXE- или DLL-файлом. Если Вам понадобится что-то записать на одну из страниц в таком регионе с атрибутом защиты PAGE_WRITECOPY или PAGE_EXECUTE_WRITECOPY, система подсунет Вашему процессу закрытую копию, связанную со страницным файлом, а не с образом файла. Эта новая страница получит те же атрибуты, что и исходная, но без защиты по типу «копирование при записи».

В третьем поле проставляется размер блока. Все блоки непрерывны в границах региона, и никаких разрывов между ними быть не может.

В четвертом поле показывается количество блоков внутри зарезервированного региона.

В пятом поле выводятся атрибуты защиты и флаги атрибутов защиты текущего блока. Атрибуты защиты блока замещают атрибуты защиты региона, содержащего данный блок. Их допустимые значения идентичны применяемым для регионов; кроме того, блоку могут быть присвоены флаги PAGE_GUARD, PAGE_WRITECOMBINE и PAGE_NOCACHE, недопустимые для региона.

Особенности адресного пространства в Windows 98

В таблице 13-4 показана карта адресного пространства при выполнении все той же программы VMMAP, но уже под управлением Windows 98. Для экономии места диапазон виртуальных адресов между 0x80018000 и 0x85620000 не приведен.

Базовый адрес	Тип	Размер	Блоки	Атрибут(ы) защиты	Описание
00000000	Free	4194304			
00400000	Private	131072	6	----	C:\CD\X86\DEBUG\14_VMMAP.EXE
00400000	Private	8192		-R-- ---	
00402000	Private	8192		-RW- ---	
00404000	Private	73728		-R-- ---	
00416000	Private	8192		-RW- ---	
00418000	Private	8192		-R-- ---	

Таблица 13-4. Образец карты адресного пространства процесса (с указанием блоков внутри регионов) в Windows 98

Таблица 13-4. продолжение

Базовый адрес	Тип	Размер	Блоки	Атрибут(ы) защиты	Описание
0041A000	Reserve	24576		---- ---	
00420000	Private	1114112	4	----	
00420000	Private	20480		-RW- ---	
00425000	Reserve	1028096		---- ---	
00520000	Private	4096		-RW- ---	
00521000	Reserve	61440		---- ---	
00530000	Private	65536	2	-RW-	
00530000	Private	4096		-RW- ---	
00531000	Reserve	61440		-RW- ---	
00540000	Private	1179648	6	----	Стек потока
00540000	Reserve	942080		---- ---	
00626000	Private	4096		-RW- ---	
00627000	Reserve	24576		---- ---	
0062D000	Private	4096		---- ---	
0062E000	Private	139264		-RW- ---	
00650000	Reserve	65536		---- ---	
00660000	Private	1114112	4	----	
00660000	Private	20480		-RW- ---	
00665000	Reserve	1028096		---- ---	
00760000	Private	4096		-RW- ---	
00761000	Reserve	61440		---- ---	
00770000	Private	1048576	2	-RW-	
00770000	Private	32768		-RW- ---	
00778000	Reserve	1015808		-RW- ---	
00870000	Free	2004418560			
78000000	Private	262144	3	----	C:\WINDOWS\SYSTEM\MSVCRT.DLL
78000000	Private	188416		-R-- ---	
7802E000	Private	57344		-RW- ---	
7803C000	Private	16384		-R-- ---	
78040000	Free	133955584			
80000000	Private	4096	1	----	
80000000	Reserve	4096		---- ---	
80001000	Private	4096	1	----	
80001000	Private	4096		-RW- ---	
80002000	Private	4096	1	----	
80002000	Private	4096		-RW- ---	
80003000	Private	4096	1	----	
80003000	Private	4096		-RW- ---	
80004000	Private	65536	2	----	
80004000	Private	32768		-RW- ---	
8000C000	Reserve	32768		---- ---	
80014000	Private	4096	1	----	
80014000	Private	4096		-RW- ---	

Таблица 13-4. продолжение

Базовый адрес	Тип	Размер	Блоки	Атрибут(ы) защиты	Описание
80015000	Private	4096	1	----	
80015000	Private	4096		-RW- ---	
80016000	Private	4096	1	----	
80016000	Private	4096		-RW- ---	
80017000	Private	4096	1	----	
80017000	Private	4096		-RW- ---	
85620000	Free	9773056			
85F72000	Private	151552	1	----	
85F72000	Private	151552		-R-- ---	
85F97000	Private	327680	1	----	
85F97000	Private	327680		-R-- ---	
85FE7000	Free	22052864			
874EF000	Private	4194304	1	----	
874EF000	Reserve	4194304		---- ---	
878EF000	Free	679219200			
B00B0000	Private	880640	3	----	
B00B0000	Private	233472		-R-- ---	
B00E9000	Private	20480		-RW- ---	
B00EE000	Private	626688		-R-- ---	
B0187000	Free	177311744			
BAAA0000	Private	315392	7	----	
BAAA0000	Private	4096		-R-- ---	
BAAA1000	Private	4096		-RW- ---	
BAAA2000	Private	241664		-R-- ---	
BAADD000	Private	4096		-RW- ---	
BAADE000	Private	4096		-R-- ---	
BAADF000	Private	32768		-RW- ---	
BAAE7000	Private	24576		-R-- ---	
BAAED000	Free	86978560			
BFDE0000	Private	20480	1	----	
BFDE0000	Private	20480		-R-- ---	
BFDE5000	Free	45056			
BFDF0000	Private	65536	3	----	
BFDF0000	Private	40960		-R-- ---	
BFDFA000	Private	4096		-RW- ---	
BFDFB000	Private	20480		-R-- ---	
BFE00000	Free	131072			
BFE20000	Private	16384	3	----	
BFE20000	Private	8192		-R-- ---	
BFE22000	Private	4096		-RW- ---	
BFE23000	Private	4096		-R-- ---	
BFE24000	Free	245760			
BFE60000	Private	24576	3	----	

Таблица 13-4. продолжение

Базовый адрес	Тип	Размер	Блоки	Атрибут(ы) защиты	Описание
BFE60000	Private	8192		-R-- ---	
BFE62000	Private	4096		-RW- ---	
BFE63000	Private	12288		-R-- ---	
BFE66000	Free	40960			
BFE70000	Private	24576	3	----	
BFE70000	Private	8192		-R-- ---	
BFE72000	Private	4096		-RW- ---	
BFE73000	Private	12288		-R-- ---	
BFE76000	Free	40960			
BFE80000	Private	65536	3	----	C:\WINDOWS\SYSTEM\ADVAPI32.DLL
BFE80000	Private	49152		-R-- ---	
BFE8C000	Private	4096		-RW- ---	
BFE8D000	Private	12288		-R-- ---	
BFE90000	Private	573440	3	----	
BFE90000	Private	425984		-R-- ---	
BFFEF8000	Private	4096		-RW- ---	
BFFEF9000	Private	143360		-R-- ---	
BFF1C000	Free	16384			
BFF20000	Private	155648	5	----	C:\WINDOWS\SYSTEM\GDI32.DLL
BFF20000	Private	126976		-R-- ---	
BFF3F000	Private	8192		-RW- ---	
BFF41000	Private	4096		-R-- ---	
BFF42000	Private	4096		-RW- ---	
BFF43000	Private	12288		-R-- ---	
BFF46000	Free	40960			
BFF50000	Private	69632	3	----	C:\WINDOWS\SYSTEM\USER32.DLL
BFF50000	Private	53248		-R-- ---	
BFF5D000	Private	4096		-RW- ---	
BFF5E000	Private	12288		-R-- ---	
BFF61000	Free	61440			
BFF70000	Private	585728	5	----	C:\WINDOWS\SYSTEM\KERNEL32.DLL
BFF70000	Private	352256		-R-- ---	
BFFC6000	Reserve	12288		---- ---	
BFFC9000	Private	16384		-RW- ---	
BFFCD000	Private	90112		-R-- ---	
BFFE3000	Reserve	114688		---- ---	
BFFFF000	Free	4096			

Главное отличие двух карт адресного пространства в том, что под управлением Windows 98 информации получаешь значительно меньше. Например, о регионах и блоках можно узнать лишь, свободные они, резервные или закрытые. Распознать тип физической памяти Mapped или Image нельзя; Windows 98 не позволяет получить дополнительную информацию, по которой можно было бы судить, что с регионом связан проецируемый в память файл или образ исполняемого файла.

Наверное, Вы заметили, что размер большинства регионов кратен 64 Кб (это значение определяется гранулярностью выделения памяти). Если размеры блоков, составляющих регион, не дают в сумме величины, кратной 64 Кб, то в конце региона часто появляется резервный блок адресного пространства. Его размер выбирается системой так, чтобы довести общий объем региона до величины, кратной 64 Кб. Например, регион, который начинается с адреса 0x00530000, включает в себя два блока: четырехкилобайтовый блок переданной памяти и резервный блок, занимающий 60 Кб адресного пространства.

Заметьте также, что на последней карте не встречаются атрибуты защиты, разрешающие исполнение или копирование при записи, поскольку Windows 98 не поддерживает их. Кроме того, она не поддерживает и флаги атрибутов защиты (PAGE_GUARD, PAGE_WRITECOMBINE и PAGE_NOCACHE). Из-за этого программе VMMap приходится использовать более сложный метод, чтобы определить, не выделен ли данный регион под стек потока.

И последнее. В Windows 98 (в отличие от Windows 2000) можно исследовать регион адресного пространства 0x80000000–0xBFFFFFFF. Это раздел, в котором находится адресное пространство, общее для всех 32-разрядных приложений. По карте видно, что в него загружены четыре системные DLL, и поэтому они доступны любому процессу.

Выравнивание данных

Здесь мы отвлечемся от виртуального адресного пространства процесса и обсудим такую важную тему, как выравнивание данных. Кстати, выравнивание данных — не столько часть архитектуры памяти в операционной системе, сколько часть архитектуры процессора.

Процессоры работают эффективнее, когда имеют дело с правильно выровненными данными. Например, значение типа WORD всегда должно начинаться с четного адреса, кратного 2, значение типа DWORD — с четного адреса, кратного 4, и т. д. При попытке считать невыровненные данные процессор сделает одно из двух: либо возбудит исключение, либо считает их в несколько приемов.

Вот фрагмент кода, обращающийся к невыровненным данным:

```
VOID SomeFunc(PVOID pvDataBuffer) {
    // первый байт в буфере содержит значение типа BYTE
    char c = * (PBYTE) pvDataBuffer;

    // увеличиваем указатель для перехода за этот байт
    pvDataBuffer = (PVOID)((PBYTE) pvDataBuffer + 1);

    // байты 2-5 в буфере содержат значение типа DWORD
    DWORD dw = * (DWORD *) pvDataBuffer;

    // на процессорах Alpha предыдущая строка приведет к исключению
    // из-за некорректного выравнивания данных
    :
}
```

Очевидно, что быстродействие программы снизится, если процессору придется обращаться к памяти в несколько приемов. В лучшем случае система потратит на доступ к невыровненному значению в 2 раза больше времени, чем на доступ к выровненному.

ненному! Так что, если Вы хотите оптимизировать работу своей программы, позаботьтесь о правильном выравнивании данных.

Рассмотрим, как справляется с выравниванием данных процессор типа x86. Такой процессор в регистре EFLAGS содержит специальный битовый флаг, называемый флагом AC (alignment check). По умолчанию, при первой подаче питания на процессор он сброшен. Когда этот флаг равен 0, процессор автоматически выполняет инструкции, необходимые для успешного доступа к невыровненным данным. Однако, если этот флаг установлен (равен 1), то при каждой попытке доступа к невыровненным данным процессор инициирует прерывание INT 17h. Версия Windows 2000 для процессоров типа x86 и Windows 98 никогда не изменяют этот битовый флаг процессора. Поэтому в программе, работающей на процессоре типа x86, исключения, связанные с попыткой доступа к невыровненным данным, никогда не возникают.

Теперь обратим внимание на процессор Alpha. Он не умеет оперировать с невыровненными данными. Когда происходит попытка доступа к таким данным, этот процессор уведомляет операционную систему. Далее Windows 2000 решает, что делать — генерировать соответствующее исключение или самой устраниить возникшую проблему, выдав процессору дополнительные инструкции. По умолчанию Windows 2000, установленная на компьютере с процессором Alpha, сама исправляет все ошибки обращения к невыровненным данным. Однако Вы можете изменить ее поведение. При загрузке Windows 2000 проверяет раздел реестра:

```
HKEY_LOCAL_MACHINE\CurrentControlSet\Control\Session Manager
```

В этом разделе может присутствовать параметр EnableAlignmentFaultExceptions. Если его нет (что чаще всего и бывает), Windows 2000 сама исправляет ошибки, связанные с доступом к невыровненным данным. Но, если он есть, система учитывает его значение. При его нулевом значении система действует так же, как и в отсутствие этого параметра. Если же он равен 1, система не исправляет такие ошибки, а генерирует исключения. Никогда не модифицируйте этот параметр в реестре без особой необходимости, потому что иначе некоторые приложения будут вызывать исключения из-за доступа к невыровненным данным и аварийно завершаться.

Чтобы упростить изменение этого параметра реестра, с Microsoft Visual C++ для платформы Alpha поставляется утилита AXPAlign.exe. Она используется так, как показано ниже.

```
Alpha AXP alignment fault exception control
```

```
Usage: axpalign [option]
```

```
Options:
```

```
/enable    to enable alignment fault exceptions.  
/disable   to disable alignment fault exceptions.  
/show      to display the current alignment exception setting.
```

```
Enable alignment fault exceptions:
```

В этом режиме любое обращение к невыровненным данным приведет к исключению.

Приложение может быть закрыто. В своем коде Вы можете найти источник ошибок, связанных с выравниванием данных, с помощью отладчика.

Действие этого параметра распространяется на все выполняемые процессы, и использовать его следует с осторожностью, так как в старых приложениях могут возникать необрабатываемые ими исключения.

Заметьте, что SetErrorMode(SEM_NOALIGNMENTFAULTEXCEPT) позволяет подавить генерацию таких исключений даже в этом режиме.

Disable alignment fault exceptions:

Этот режим действует по умолчанию в Windows NT for Alpha AXP версий 3.1 и 3.5.

Операционная система сама исправляет любые ошибки, связанные с доступом к невыровненным данным (если такие ошибки возникают), и приложения или отладчики их не замечают. Если программа часто обращается к невыровненным данным, производительность системы может заметно снизиться. Для наблюдения за частотой появления таких ошибок можно использовать Perfmon или wperf.

Эта утилита просто модифицирует нужный параметр реестра или показывает его текущее значение. Изменив значение этого параметра, перезагрузите компьютер, чтобы изменения вступили в силу.

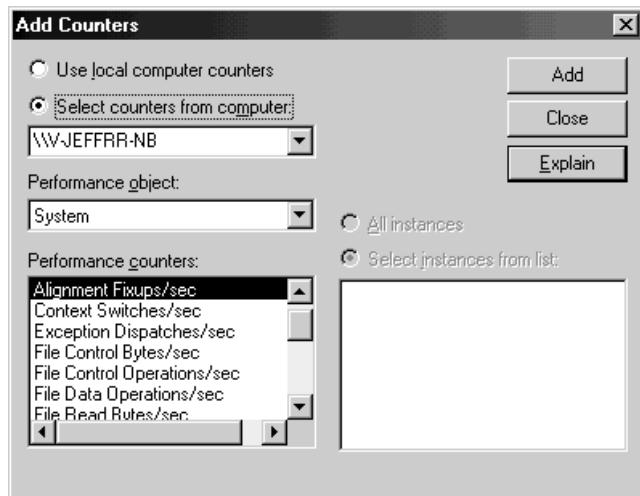
Но, даже не пользуясь утилитой AXPAalign, Вы все равно можете заставить систему молча исправлять ошибки обращения к невыровненным данным во всех потоках Вашего процесса. Для этого один из потоков должен вызвать функцию *SetErrorMode*:

```
UINT SetErrorMode(UINT fuErrorMode);
```

В данном случае Вам нужен флаг SEM_NOALIGNMENTFAULTEXCEPT. Когда он установлен, система автоматически исправляет ошибки обращения к невыровненным данным, а когда он сброшен, система вместо этого генерирует соответствующие исключения. Заметьте, что изменение этого флага влияет на потоки только того процесса, из которого была вызвана функция *SetErrorMode*. Иначе говоря, его модификация не отражается на потоках других процессов. Также учтите, что любые флаги режимов обработки ошибок наследуются всеми дочерними процессами. Поэтому перед вызовом функции *CreateProcess* Вам может понадобиться временно сбросить этот флаг.

SetErrorMode можно вызывать с флагом SEM_NOALIGNMENTFAULTEXCEPT независимо от того, на какой платформе выполняется Ваше приложение. Но результаты ее вызова не всегда одинаковы. На платформе x86 сбросить этот флаг просто нельзя, а на платформе Alpha его разрешается сбросить, только если параметр EnableAlignmentFaultExceptions в реестре равен 1.

Для наблюдения за частотой возникновения ошибок, связанных с доступом к невыровненным данным, в Windows 2000 можно использовать Performance Monitor, подключаемый к MMC. На следующей иллюстрации показано диалоговое окно Add Counters, которое позволяет добавить нужный показатель в Performance Monitor.



Этот показатель сообщает, сколько раз в секунду процессор уведомляет операционную систему о доступе к невыровненным данным. На компьютере с процессором типа *x86* он всегда равен 0. Это связано с тем, что такой процессор сам справляется с проблемами обращения к невыровненным данным и не уведомляет об этом операционную систему. А поскольку он обходится без помощи со стороны операционной системы, падение производительности при частом доступе к невыровненным данным не столь значительно, как на процессорах, требующих с той же целью участия операционной системы.

Как видите, простого вызова *SetErrorMode* вполне достаточно для того, чтобы Ваше приложение работало корректно. Но это решение явно не самое эффективное. Так, в *Alpha Architecture Reference Manual*, опубликованном Digital Press, утверждается, что системный код, автоматически устраняющий ошибки обращения к невыровненным данным, может снизить быстродействие в 100 раз! Издержки слишком велики. К счастью, есть более эффективное решение этой проблемы.

Компилятор Microsoft C/C++ для процессоров Alpha поддерживает ключевое слово *_unaligned*. Этот модификатор используется так же, как *const* или *volatile*, но применим лишь для переменных-указателей. Когда Вы обращаетесь к данным через невыровненный указатель (*unaligned pointer*), компилятор генерирует код, исходя из того, что данные скорее всего не выровнены, и вставляет дополнительные машинные инструкции, необходимые для доступа к таким данным. Ниже показан тот же фрагмент кода, что и в начале раздела, но с использованием ключевого слова *_unaligned*.

```
VOID SomeFunc(PVOID pvDataBuffer) {  
  
    // первый байт в буфере содержит значение типа BYTE  
    char c = * (PBYTE) pvDataBuffer;  
  
    // увеличиваем указатель для перехода за этот байт  
    pvDataBuffer = (PVOID)((PBYTE) pvDataBuffer + 1);  
  
    // байты 2-5 в буфере содержат значение типа DWORD  
    DWORD dw = * (_unaligned DWORD *) pvDataBuffer;  
  
    // Предыдущая строка заставит компилятор сгенерировать дополнительные  
    // машинные инструкции, которые позволят считать значение типа DWORD  
    // в несколько приемов. При этом исключение из-за попытки доступа  
    // к невыровненным данным не возникнет.  
    :  
}
```

При компиляции следующей строки на процессоре Alpha, генерируется 7 машинных инструкций.

```
DWORD dw = * (_unaligned DWORD *) pvDataBuffer;
```

Но если я уберу ключевое слово *_unaligned*, то получу всего 3 машинные инструкции. Как видите, модификатор *_unaligned* на процессорах Alpha приводит к увеличению числа генерируемых машинных инструкций более чем в 2 раза. Но инструкции, добавляемые компилятором, все равно намного эффективнее, чем перехват процессором попыток доступа к невыровненным данным и исправление таких ошибок операционной системой.

И последнее. Ключевое слово *_unaligned* на процессорах типа *x86* компилятором Visual C/C++ не поддерживается. На этих процессорах оно просто не нужно. Но это

означает, что версия компилятора для процессоров *x86*, встретив в исходном коде ключевое слово *_unaligned*, сообщит об ошибке. Поэтому, если Вы хотите создать единую базу исходного кода приложения для обеих процессорных платформ, используйте вместо *_unaligned* макрос UNALIGNED. Он определен в файле WinNT.h так:

```
#if defined(_M_MRX000) || defined(_M_ALPHA) || defined(_M_IA64)
#define UNALIGNED __unaligned
#if defined(_WIN64)
#define UNALIGNED64 __unaligned
#else
#define UNALIGNED64
#endif
#else
#define UNALIGNED
#define UNALIGNED64
#endif
```

Исследование виртуальной памяти

В предыдущей главе мы выяснили, как система управляет виртуальной памятью, как процесс получает свое адресное пространство и что оно собой представляет. А сейчас мы перейдем от теории к практике и рассмотрим некоторые Windows-функции, сообщающие о состоянии системной памяти и виртуального адресного пространства в том или ином процессе.

Системная информация

Многие параметры операционной системы (размер страницы, гранулярность выделения памяти и др.) зависят от используемого в компьютере процессора. Поэтому нельзя жестко «зашивать» их значения в исходный код программ. Эту информацию надо считывать в момент инициализации процесса с помощью функции *GetSystemInfo*:

```
VOID GetSystemInfo(LPSYSTEM_INFO psinf);
```

Вы должны передать в *GetSystemInfo* адрес структуры *SYSTEM_INFO*, и функция инициализирует элементы этой структуры:

```
typedef struct _SYSTEM_INFO {
    union {
        DWORD dwOemId; // не используйте этот элемент, он устарел
        struct {
            WORD wProcessorArchitecture;
            WORD wReserved;
        };
    };
    DWORD     dwPageSize;
    LPVOID    lpMinimumApplicationAddress;
    LPVOID    lpMaximumApplicationAddress;
    DWORD_PTR dwActiveProcessorMask;
    DWORD     dwNumberOfProcessors;
    DWORD     dwProcessorType;
    DWORD     dwAllocationGranularity;
    WORD      wProcessorLevel;
    WORD      wProcessorRevision;
} SYSTEM_INFO, *LPSYSTEM_INFO;
```

При загрузке система определяет значения элементов этой структуры; для конкретной системы их значения постоянны. Функция *GetSystemInfo* предусмотрена специально для того, чтобы и приложения могли получать эту информацию. Из всех элементов структуры лишь четыре имеют отношение к памяти. Они описаны в следующей таблице.

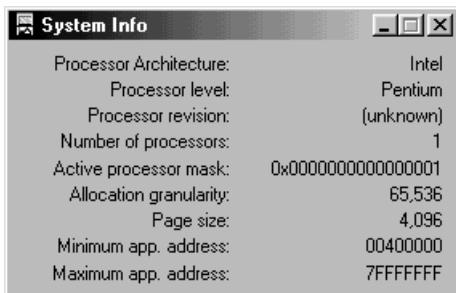
Элемент	Описание
<i>dwPageSize</i>	Размер страницы памяти. На процессорах x86 это значение равно 4096, а на процессорах Alpha — 8192 байтам.
<i>lpMinimumApplicationAddress</i>	Минимальный адрес памяти доступного адресного пространства для каждого процесса. В Windows 98 это значение равно 4 194 304, или 0x00400000, поскольку нижние 4 Мб адресного пространства каждого процесса недоступны. В Windows 2000 это значение равно 65 536, или 0x00010000, так как в этой системе резервируются лишь первые 64 Кб адресного пространства каждого процесса.
<i>lpMaximumApplicationAddress</i>	Максимальный адрес памяти доступного адресного пространства, отведенного в «личное пользование» каждому процессу. В Windows 98 этот адрес равен 2 147 483 647, или 0x7FFFFFFF, так как верхние 2 Гб занимают общие файлы, проецируемые в память, и разделяемый код операционной системы. В Windows 2000 этот адрес соответствует началу раздела для кода и данных режима ядра за вычетом 64 Кб.
<i>dwAllocationGranularity</i>	Гранулярность резервирования регионов адресного пространства. На момент написания книги это значение составляет 64 Кб для всех платформ Windows.

Остальные элементы этой структуры показаны в таблице ниже.

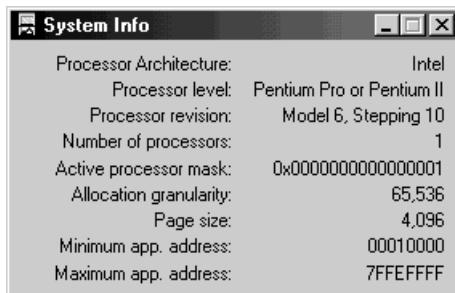
Элемент	Описание
<i>dwOemId</i>	Устарел; больше не используется
<i>wReserved</i>	Зарезервирован на будущее; пока не используется
<i>dwNumberOfProcessors</i>	Число процессоров в компьютере
<i>dwActiveProcessorMask</i>	Битовая маска, которая сообщает, какие процессоры активны (выполняют потоки)
<i>dwProcessorType</i>	Используется только в Windows 98; сообщает тип процессора, например Intel 386, 486 или Pentium
<i>wProcessorArchitecture</i>	Используется только в Windows 2000; сообщает тип архитектуры процессора, например Intel, Alpha, 64-разрядный Intel или 64-разрядный Alpha
<i>wProcessorLevel</i>	Используется только в Windows 2000; сообщает дополнительные подробности об архитектуре процессора, например Intel Pentium Pro или Pentium II
<i>wProcessorRevision</i>	Используется только в Windows 2000; сообщает дополнительные подробности об уровне данной архитектуры процессора

Программа-пример SysInfo

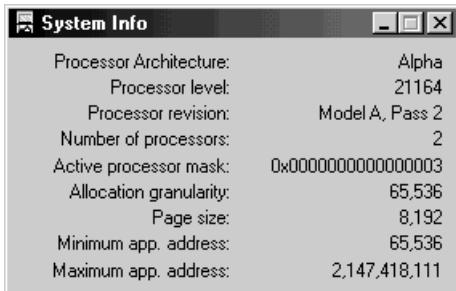
Эта программа, «14 SysInfo.exe» (см. листинг на рис. 14-1), весьма проста; она вызывает функцию *GetSystemInfo* и выводит на экран информацию, возвращенную в структуре *SYSTEM_INFO*. Файлы исходного кода и ресурсов этой программы находятся в каталоге 14-SysInfo на компакт-диске, прилагаемом к книге. Диалоговые окна с результатами выполнения программы SysInfo на разных процессорных plataформах показаны ниже.



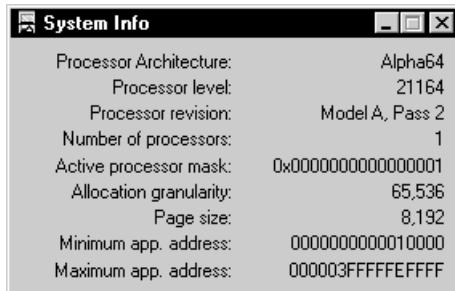
Windows 98 на процессоре x86



32-разрядная Windows 2000 на процессоре x86



32-разрядная Windows 2000
на процессоре Alpha



64-разрядная Windows 2000 на процессоре Alpha

 SysInfo.cpp

```

 ****
 Модуль: SysInfo.cpp
 Автор: Copyright (c) 2000, Джейфри Рихтер (Jeffrey Richter)
 ****

 #include "..\CmnHdr.h"      /* см. приложение A */
 #include <windowsx.h>
 #include <tchar.h>
 #include <stdio.h>
 #include "Resource.h"

 /////////////////////////////////
 // устанавливаем TRUE, если программа выполняется в Windows 9x
 BOOL g_fWin9xIsHost = FALSE;

 /////////////////////////////////

 // эта функция принимает число и преобразует его в строку,
 // вставляя в нужных местах запятые
 PTSTR BigNumToString(LONG lNum, PTSTR szBuf) {

    TCHAR szNum[100];

```

Рис. 14-1. Программа-пример SysInfo

Рис. 14-1. продолжение

```

wsprintf(szNum, TEXT("%d"), lNum);
NUMBERFMT nf;
nf.NumDigits = 0;
nf.LeadingZero = FALSE;
nf.Grouping = 3;
nf.lpDecimalSep = TEXT(".");
nf.lpThousandSep = TEXT(",");
nf.NegativeOrder = 0;
GetNumberFormat(LOCALE_USER_DEFAULT, 0, szNum, &nf, szBuf, 100);
return(szBuf);
}

//////////////////////////////



void ShowCPUInfo(HWND hwnd, WORD wProcessorArchitecture, WORD wProcessorLevel,
WORD wProcessorRevision) {

TCHAR szCPURArch[64] = TEXT("(unknown)");
TCHAR szCPULevel[64] = TEXT("(unknown)");
TCHAR szCPURRev[64] = TEXT("(unknown)");

switch (wProcessorArchitecture) {
    case PROCESSOR_ARCHITECTURE_INTEL:
        lstrcpy(szCPURArch, TEXT("Intel"));
        switch (wProcessorLevel) {
            case 3: case 4:
                wsprintf(szCPULevel, TEXT("80%c86"), wProcessorLevel + '0');
                if (!g_fWin9xIsHost)
                    wsprintf(szCPURRev, TEXT("%c%d"),
                            HIBYTE(wProcessorRevision) + TEXT('A'), LOBYTE(wProcessorRevision));
                break;

            case 5:
                wsprintf(szCPULevel, TEXT("Pentium"));
                if (!g_fWin9xIsHost)
                    wsprintf(szCPURRev, TEXT("Model %d, Stepping %d"),
                            HIBYTE(wProcessorRevision), LOBYTE(wProcessorRevision));
                break;

            case 6:
                wsprintf(szCPULevel, TEXT("Pentium Pro or Pentium II"));
                if (!g_fWin9xIsHost)
                    wsprintf(szCPURRev, TEXT("Model %d, Stepping %d"),
                            HIBYTE(wProcessorRevision), LOBYTE(wProcessorRevision));
                break;
        }
        break;

    case PROCESSOR_ARCHITECTURE_ALPHA:
        lstrcpy(szCPURArch, TEXT("Alpha"));
        wsprintf(szCPULevel, TEXT("%d"), wProcessorLevel);
}

```

см. след. стр.

Рис. 14-1. продолжение

```

wsprintf(szCPURev, TEXT("Model %c, Pass %d"),
         HIBYTE(wProcessorRevision) + TEXT('A'), LOBYTE(wProcessorRevision));
break;

case PROCESSOR_ARCHITECTURE_IA64:
    lstrcpy(szCPUArch, TEXT("IA-64"));
    wsprintf(szCPULevel, TEXT("%d"), wProcessorLevel);
    wsprintf(szCPURev, TEXT("Model %c, Pass %d"),
             HIBYTE(wProcessorRevision) + TEXT('A'), LOBYTE(wProcessorRevision));
break;

case PROCESSOR_ARCHITECTURE_ALPHA64:
    lstrcpy(szCPUArch, TEXT("Alpha64"));
    wsprintf(szCPULevel, TEXT("%d"), wProcessorLevel);
    wsprintf(szCPURev, TEXT("Model %c, Pass %d"),
             HIBYTE(wProcessorRevision) + TEXT('A'), LOBYTE(wProcessorRevision));
break;

case PROCESSOR_ARCHITECTURE_UNKNOWN:
default:
    wsprintf(szCPUArch, TEXT("Unknown"));
    break;
}

SetDlgItemText(hwnd, IDC_PROCARCH, szCPUArch);
SetDlgItemText(hwnd, IDC_PROCLEVEL, szCPULevel);
SetDlgItemText(hwnd, IDC_PROCREV, szCPURev);
}

////////////////////////////////////////////////////////////////

BOOL Dlg_OnInitDialog(HWND hwnd, HWND hwndFocus, LPARAM lParam) {
    chSETDLGICONS(hwnd, IDI_SYSINFO);

    SYSTEM_INFO sinf;
    GetSystemInfo(&sinf);

    if (g_fWin9xIsHost) {
        sinf.wProcessorLevel = (WORD) (sinf.dwProcessorType / 100);
    }

    ShowCPUInfo(hwnd, sinf.wProcessorArchitecture,
                sinf.wProcessorLevel, sinf.wProcessorRevision);

    TCHAR szBuf[50];
    SetDlgItemText(hwnd, IDC_PAGESIZE, BigNumToString(sinf.dwPageSize, szBuf));

    _stprintf(szBuf, TEXT("%p"), sinf.lpMinimumApplicationAddress);
    SetDlgItemText(hwnd, IDC_MINAPPADDR, szBuf);

    _stprintf(szBuf, TEXT("%p"), sinf.lpMaximumApplicationAddress);
    SetDlgItemText(hwnd, IDC_MAXAPPADDR, szBuf);
}

```

Рис. 14-1. продолжение

```

_stprintf(szBuf, TEXT("0x%016I64X"), (_int64) sinf.dwActiveProcessorMask);
SetDlgItemText(hwnd, IDC_ACTIVEPROCMASK, szBuf);

SetDlgItemText(hwnd, IDC_NUMOFPROCS,
    BigNumToString(sinf.dwNumberOfProcessors, szBuf));

SetDlgItemText(hwnd, IDC_ALLOCGRAN,
    BigNumToString(sinf.dwAllocationGranularity, szBuf));

return(TRUE);
}

//////////



void Dlg_OnCommand(HWND hwnd, int id, HWND hwndCtl, UINT codeNotify) {

switch (id) {
    case IDCANCEL:
        EndDialog(hwnd, id);
        break;
}
}

//////////



INT_PTR WINAPI Dlg_Proc(HWND hDlg, UINT uMsg, WPARAM wParam, LPARAM lParam) {

switch (uMsg) {
    chHANDLE_DLMSG(hDlg, WM_INITDIALOG, Dlg_OnInitDialog);
    chHANDLE_DLMSG(hDlg, WM_COMMAND, Dlg_OnCommand);
}
return(FALSE);
}

//////////



int WINAPI _tWinMain(HINSTANCE hinstExe, HINSTANCE, PTSTR pszCmdLine, int) {

OSVERSIONINFO vi = { sizeof(vi) };
GetVersionEx(&vi);
g_fWin9xIsHost = (vi.dwPlatformId == VER_PLATFORM_WIN32_WINDOWS);
DialogBox(hinstExe, MAKEINTRESOURCE(IDD_SYSINFO), NULL, Dlg_Proc);
return(0);
}

////////// Конец файла ///////////////

```

Статус виртуальной памяти

Windows-функция *GlobalMemoryStatus* позволяет отслеживать текущее состояние памяти:

```
VOID GlobalMemoryStatus(LPMEMORYSTATUS pmst);
```

На мой взгляд, она названа крайне неудачно; имя *GlobalMemoryStatus* подразумевает, что функция каким-то образом связана с глобальными кучами в 16-разрядной Windows. Мне кажется, что лучше было бы назвать функцию *GlobalMemoryStatus* по-другому — скажем, *VirtualMemoryStatus*.

При вызове функции *GlobalMemoryStatus* Вы должны передать адрес структуры *MEMORYSTATUS*. Вот эта структура:

```
typedef struct _MEMORYSTATUS {
    DWORD dwLength;
    DWORD dwMemoryLoad;
    SIZE_T dwTotalPhys;
    SIZE_T dwAvailPhys;
    SIZE_T dwTotalPageFile;
    SIZE_T dwAvailPageFile;
    SIZE_T dwTotalVirtual;
    SIZE_T dwAvailVirtual;
} MEMORYSTATUS, *LPMEMORYSTATUS;
```

Перед вызовом *GlobalMemoryStatus* надо записать в элемент *dwLength* размер структуры в байтах. Такой принцип вызова функции дает возможность Microsoft расширять эту структуру в будущих версиях Windows, не нарушая работу существующих приложений. После вызова *GlobalMemoryStatus* инициализирует остальные элементы структуры и возвращает управление. Назначение элементов этой структуры Вы узнаете из следующего раздела, в котором рассматривается программа-пример VMStat.

Если Вы полагаете, что Ваше приложение будет работать на машинах с объемом оперативной памяти более 4 Гб или файлом подкачки более 4 Гб, используйте новую функцию *GlobalMemoryStatusEx*:

```
BOOL GlobalMemoryStatusEx(LPMEMORYSTATUSEX pmst);
```

Вы должны передать ей адрес новой структуры *MEMORYSTATUSEX*:

```
typedef struct _MEMORYSTATUSEX {
    DWORD dwLength;
    DWORD dwMemoryLoad;
    DWORDLONG ullTotalPhys;
    DWORDLONG ullAvailPhys;
    DWORDLONG ullTotalPageFile;
    DWORDLONG ullAvailPageFile;
    DWORDLONG ullTotalVirtual;
    DWORDLONG ullAvailVirtual;
    DWORDLONG ullAvailExtendedVirtual;
} MEMORYSTATUSEX, *LPMEMORYSTATUSEX;
```

Эта структура идентична первоначальной структуре *MEMORYSTATUS* с одним исключением: все ее элементы имеют размер по 64 бита, что позволяет оперировать со значениями, превышающими 4 Гб. Последний элемент, *ullAvailExtendedVirtual*, указывает размер незарезервированной памяти в самой большой области памяти виртуального адресного пространства вызывающего процесса. Этот элемент имеет смысл только для процессоров определенных архитектур при определенных конфигурациях.

Программа-пример VMStat

Эта программа, «14 VMStat.exe» (см. листинг на рис. 14-2), выводит на экран окно с результатами вызова *GlobalMemoryStatus*. Информация в окне обновляется каждую секунду, так что VMStat вполне пригодна для мониторинга памяти в системе. Файлы

исходного кода и ресурсов этой программы находятся в каталоге 14-VMStat на компакт-диске, прилагаемом к книге. Окно этой программы после запуска в Windows 2000 на машине с процессором Intel Pentium II и 128 Мб оперативной памяти показано ниже.

VMStat	
Memory load:	60
TotalPhys:	133677056
AvailPhys:	52719616
TotalPageFile:	318541824
AvailPageFile:	254005248
TotalVirtual:	2147352576
AvailVirtual:	2136846336

Элемент *dwMemoryLoad* (показываемый как Memory Load) позволяет оценить, насколько занята подсистема управления памятью. Это число может быть любым в диапазоне от 0 до 100. В Windows 98 и Windows 2000 алгоритмы, используемые для его подсчета, различны. Кроме того, в будущих версиях операционных систем этот алгоритм почти наверняка придется модифицировать. Но, честно говоря, на практике от значения этого элемента толку немного.

Элемент *dwTotalPhys* (показываемый как TotalPhys) отражает общий объем физической (оперативной) памяти в байтах. На данной машине с Pentium II и 128 Мб оперативной памяти его значение составляет 133 677 056, что на 540 672 байта меньше 128 Мб. Причина, по которой *GlobalMemoryStatus* не сообщает о полных 128 Мб, кроется в том, что система при загрузке резервирует небольшой участок оперативной памяти, недоступный даже ядру. Этот участок никогда не сбрасывается на диск. А элемент *dwAvailPhys* (показываемый как AvailPhys) дает число байтов свободной физической памяти.

Элемент *dwTotalPageFile* (показываемый как TotalPageFile) сообщает максимальное количество байтов, которое может содержаться в страничном файле (файлах) на жестком диске (дисках). Хотя VMStat показывает, что текущий размер страничного файла составляет 318 574 592 байта, система может варьировать его по своему усмотрению. Элемент *dwAvailPageFile* (показываемый как AvailPageFile) подсказывает, что в данный момент 233 046 016 байтов в страничном файле свободно и может быть передано любому процессу.

Элемент *dwTotalVirtual* (показываемый как TotalVirtual) отражает общее количество байтов, отведенных под закрытое адресное пространство процесса. Значение 2 147 352 576 ровно на 128 Кб меньше 2 Гб. Два раздела недоступного адресного пространства — от 0x00000000 до 0x0000FFFF и от 0x7FFF0000 до 0x7FFFFFFF — как раз и составляют эту разницу в 128 Кб. Запустив VMStat в Windows 98, Вы увидите, что значение этого элемента поменялось на 2 143 289 344 (2 Гб за вычетом 4 Мб). Разница в 4 Мб возникает из-за того, что Windows 98 блокирует нижний раздел от 0x00000000 до 0x003FFFFF (размером в 4 Мб).

И, наконец, *dwAvailVirtual* (показываемый как AvailVirtual) — единственный элемент структуры, специфичный для конкретного процесса, вызывающего *GlobalMemoryStatus* (остальные элементы относятся исключительно к самой системе и не зависят от того, какой именно процесс вызывает эту функцию). При подсчете значения *dwAvailVirtual* функция суммирует размеры всех свободных регионов в адресном пространстве вызывающего процесса. В данном случае его значение говорит о том, что в распоряжении программы VMStat имеется 2 136 846 336 байтов свободного адресного пространства. Вычтя из значения *dwTotalVirtual* величину *dwAvailVirtual*, Вы получите 10 506 240 байтов — такой объем памяти VMStat зарезервировала в своем виртуальном адресном

пространстве. Отдельного элемента, который сообщал бы количество физической памяти, используемой процессом в данный момент, не предусмотрено.



VMStat.cpp

```

 ****
 Модуль: VMStat.cpp
 Автор: Copyright (c) 2000, Джейфри Рихтер (Jeffrey Richter)
 ****

#include "..\CmnHdr.h"      /* см. приложение A */
#include <windowsx.h>
#include <tchar.h>
#include <stdio.h>
#include "Resource.h"

///////////

// идентификатор таймера, отвечающего за обновление информации
#define IDT_UPDATE 1

///////////

BOOL Dlg_OnInitDialog(HWND hwnd, HWND hwndFocus, LPARAM lParam) {

    chSETDLGICONS(hwnd, IDI_VMSTAT);

    // устанавливаем таймер так, чтобы периодически обновлять информацию
    SetTimer(hwnd, IDT_UPDATE, 1 * 1000, NULL);
    // выдаем сообщение таймера для первого обновления
    FORWARD_WM_TIMER(hwnd, IDT_UPDATE, SendMessage);
    return(TRUE);
}

///////////

void Dlg_OnTimer(HWND hwnd, UINT id) {

    // прежде чем передать структуру функции GlobalMemoryStatus,
    // заносим в элемент dwLength ее длину
    MEMORYSTATUS ms = { sizeof(ms) };
    GlobalMemoryStatus(&ms);

    TCHAR szData[512] = { 0 };
    _stprintf(szData, TEXT("%d\n%d\n%I64d\n%I64d\n%I64d\n%I64d\n%I64d"),
              ms.dwMemoryLoad, ms.dwTotalPhys,
              (_int64) ms.dwAvailPhys, (_int64) ms.dwTotalPageFile,
              (_int64) ms.dwAvailPageFile, (_int64) ms.dwTotalVirtual,
              (_int64) ms.dwAvailVirtual);
    SetDlgItemText(hwnd, IDC_DATA, szData);
}

```

Рис. 14-2. Программа-пример VMStat

Рис. 14-2. продолжение

```
//////////  

void Dlg_OnCommand(HWND hwnd, int id, HWND hwndCtl, UINT codeNotify) {  

    switch (id) {  

        case IDCANCEL:  

            KillTimer(hwnd, IDT_UPDATE);  

            EndDialog(hwnd, id);  

            break;  

    }  

}  

//////////  

INT_PTR WINAPI Dlg_Proc(HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam) {  

    switch (uMsg) {  

        chHANDLE_DLGMMSG(hwnd, WM_INITDIALOG, Dlg_OnInitDialog);  

        chHANDLE_DLGMMSG(hwnd, WM_COMMAND, Dlg_OnCommand);  

        chHANDLE_DLGMMSG(hwnd, WM_TIMER, Dlg_OnTimer);  

    }  

    return(FALSE);  

}  

//////////  

int WINAPI _tWinMain(HINSTANCE hinstExe, HINSTANCE, PTSTR pszCmdLine, int) {  

    DialogBox(hinstExe, MAKEINTRESOURCE(IDD_VMSSTAT), NULL, Dlg_Proc);  

    return(0);  

}  

////////// Конец файла ///////////
```

Определение состояния адресного пространства

В Windows имеется функция, позволяющая запрашивать определенную информацию об участке памяти по заданному адресу (в пределах адресного пространства вызывающего процесса): размер, тип памяти и атрибуты защиты. В частности, с ее помощью программа VMMap (ее листинг см. на рис. 14-4) выводит карты виртуальной памяти, с которыми мы познакомились в главе 13. Вот эта функция:

```
DWORD VirtualQuery(  

    LPCVOID pvAddress,  

    PMEMORY_BASIC_INFORMATION pmbi,  

    DWORD dwLength);
```

Парная ей функция, *VirtualQueryEx*, сообщает ту же информацию о памяти, но в другом процессе:

```
DWORD VirtualQueryEx(  

    HANDLE hProcess,  

    LPCVOID pvAddress,  

    PMEMORY_BASIC_INFORMATION pmbi,  

    DWORD dwLength);
```

Эти функции идентичны с тем исключением, что *VirtualQueryEx* принимает описатель процесса, об адресном пространстве которого Вы хотите получить информацию. Чаще всего функцией *VirtualQueryEx* пользуются отладчики и системные утилиты — остальные приложения обращаются к *VirtualQuery*. При вызове *VirtualQuery(Ex)* параметр *pvAddress* должен содержать адрес виртуальной памяти, о которой Вы хотите получить информацию. Параметр *pmhi* — это адрес структуры **MEMORY_BASIC_INFORMATION**, которую надо создать перед вызовом функции. Данная структура определена в файле WinNT.h так:

```
typedef struct _MEMORY_BASIC_INFORMATION {
    PVOID BaseAddress;
    PVOID AllocationBase;
    DWORD AllocationProtect;
    SIZE_T RegionSize;
    DWORD State;
    DWORD Protect;
    DWORD Type;
} MEMORY_BASIC_INFORMATION, *PMEMORY_BASIC_INFORMATION;
```

Параметр *dwLength* задает размер структуры **MEMORY_BASIC_INFORMATION**. Функция *VirtualQuery(Ex)* возвращает число байтов, скопированных в буфер.

Используя адрес, указанный Вами в параметре *pvAddress*, функция *VirtualQuery(Ex)* заполняет структуру информацией о диапазоне смежных страниц, имеющих одинаковые состояние, атрибуты защиты и тип. Описание элементов структуры приведено в таблице ниже.

Элемент	Описание
<i>BaseAddress</i>	Сообщает то же значение, что и параметр <i>pvAddress</i> , но округленное до ближайшего меньшего адреса, кратного размеру страницы.
<i>AllocationBase</i>	Идентифицирует базовый адрес региона, включающего в себя адрес, указанный в параметре <i>pvAddress</i> .
<i>AllocationProtect</i>	Идентифицирует атрибут защиты, присвоенный региону при его резервировании.
<i>RegionSize</i>	Сообщает суммарный размер (в байтах) группы страниц, которые начинаются с базового адреса <i>BaseAddress</i> и имеют те же атрибуты защиты, состояние и тип, что и страница, расположенная по адресу, указанному в параметре <i>pvAddress</i> .
<i>State</i>	Сообщает состояние (MEM_FREE, MEM_RESERVE или MEM_COMMIT) всех смежных страниц, которые имеют те же атрибуты защиты, состояние и тип, что и страница, расположенная по адресу, указанному в параметре <i>pvAddress</i> .
<i>Protect</i>	При MEM_FREE элементы <i>AllocationBase</i> , <i>AllocationProtect</i> , <i>Protect</i> и <i>Type</i> содержат неопределенные значения, а при MEM_RESERVE неопределенное значение содержит элемент <i>Protect</i> .
<i>Type</i>	Идентифицирует тип физической памяти (MEM_IMAGE, MEM_MAPPED или MEM_PRIVATE), связанной с группой смежных страниц, которые имеют те же атрибуты защиты, состояние и тип, что и страница, расположенная по адресу, указанному в параметре <i>pvAddress</i> . В Windows 98 этот элемент всегда дает MEM_PRIVATE.

Функция *VMQuery*

Начиная изучать архитектуру памяти в Windows, я пользовался функцией *VirtualQuery* как «поводырем». Если Вы читали первое издание моей книги, то заметите, что программа VMMap была гораздо проще ее нынешней версии, представленной в следующем разделе. Прежняя была построена на очень простом цикле, из которого периодически вызывалась функция *VirtualQuery*, и для каждого вызова я формировал одну строку, содержащую элементы структуры `MEMORY_BASIC_INFORMATION`. Изучая полученные дампы и сверяясь с документацией из SDK (в то время весьма неудачной), я пытался разобраться в архитектуре подсистемы управления памятью. Что ж, с тех пор я многому научился и теперь знаю, что функция *VirtualQuery* и структура `MEMORY_BASIC_INFORMATION` не дают полной картины.

Проблема в том, что в `MEMORY_BASIC_INFORMATION` возвращается отнюдь не вся информация, имеющаяся в распоряжении системы. Если Вам нужны простейшие данные о состоянии памяти по конкретному адресу, *VirtualQuery* действительно незаменим. Она отлично работает, если Вас интересует, передана ли по этому адресу физическая память и доступен ли он для операций чтения или записи. Но попробуйте с ее помощью узнать общий размер зарезервированного региона и количество блоков в нем или выяснить, не содержит ли этот регион стек потока, — ничего не выйдет.

Чтобы получать более полную информацию о памяти, я создал собственную функцию и назвал ее *VMQuery*:

```
BOOL VMQuery(
    HANDLE hProcess,
    PVOID pvAddress,
    PVMQUERY pVMQ);
```

По аналогии с *VirtualQueryEx* она принимает в *hProcess* описатель процесса, в *pvAddress* — адрес памяти, а в *pVMQ* — указатель на структуру, заполняемую самой функцией. Структура `VMQUERY` (тоже определенная мной) представляет собой вот что:

```
typedef struct {
    // информация о регионе
    PVOID pvRgnBaseAddress;
    DWORD dwRgnProtection;      // PAGE_*
    SIZE_T RgnSize;
    DWORD dwRgnStorage;         // MEM_*: Free, Image, Mapped, Private
    DWORD dwRgnBlocks;
    DWORD dwRgnGuardBlks;       // если > 0, регион содержит стек потока
    BOOL fRgnIsAStack;          // TRUE, если регион содержит стек потока

    // информация о блоке
    PVOID pvBlkBaseAddress;
    DWORD dwBlkProtection;      // PAGE_*
    SIZE_T BlkSize;
    DWORD dwBlkStorage;         // MEM_*: Free, Reserve, Image, Mapped, Private
} VMQUERY, *PVMQUERY;
```

С первого взгляда заметно, что моя структура `VMQUERY` содержит куда больше информации, чем `MEMORY_BASIC_INFORMATION`. Она разбита (условно, конечно) на две части: в одной — информация о регионе, в другой — информация о блоке (адрес которого указан в параметре *pvAddress*). Элементы этой структуры описываются в следующей таблице.

Элемент	Описание
<i>pvRgnBaseAddress</i>	Идентифицирует базовый адрес региона виртуального адресного пространства, включающего адрес, указанный в параметре <i>pvAddress</i> .
<i>dwRgnProtection</i>	Сообщает атрибут защиты, присвоенный региону при его резервировании.
<i>RgnSize</i>	Указывает размер (в байтах) зарезервированного региона.
<i>dwRgnStorage</i>	Идентифицирует тип физической памяти, используемой группой блоков данного региона: MEM_FREE, MEM_IMAGE, MEM_MAPPED или MEM_PRIVATE. Поскольку Windows 98 не различает типы памяти, в этой операционной системе данный элемент содержит либо MEM_FREE, либо MEM_PRIVATE.
<i>dwRgnBlocks</i>	Содержит значение — число блоков в указанном регионе.
<i>dwRgnGuardBlks</i>	Указывает число блоков с установленным флагом атрибутов защиты PAGE_GUARD. Обычно это значение либо 0, либо 1. Если оно равно 1, то регион скорее всего зарезервирован под стек потока. В Windows 98 этот элемент всегда равен 0.
<i>fRgnIsAStack</i>	Сообщает, есть ли в данном регионе стек потока. Результат определяется на основе взвешенной оценки, так как невозможно дать стопроцентной гарантии тому, что в регионе содержится стек.
<i>pvBlkBaseAddress</i>	Идентифицирует базовый адрес блока, включающего адрес, указанный в параметре <i>pvAddress</i> .
<i>dwBlkProtection</i>	Идентифицирует атрибут защиты блока, включающего адрес, указанный в параметре <i>pvAddress</i> .
<i>BlkSize</i>	Содержит значение — размер блока (в байтах), включающего адрес, указанный в параметре <i>pvAddress</i> .
<i>dwBlkStorage</i>	Идентифицирует содержимое блока, включающего адрес, указанный в параметре <i>pvAddress</i> . Принимает одно из значений: MEM_FREE, MEM_RESERVE, MEM_IMAGE, MEM_MAPPED или MEM_PRIVATE. В Windows 98 этот элемент никогда не содержит значения MEM_IMAGE и MEM_MAPPED.

Чтобы получить всю эту информацию, *VMQuery*, естественно, приходится выполнять гораздо больше операций (в том числе многократно вызывать *VirtualQueryEx*), а потому она работает значительно медленнее *VirtualQueryEx*. Так что Вы должны все тщательно взвесить, прежде чем остановить свой выбор на одной из этих функций. Если Вам не нужна дополнительная информация, возвращаемая *VMQuery*, используйте *VirtualQuery* или *VirtualQueryEx*.

Листинг файла VMQuery.cpp (рис. 14-3) показывает, как я получаю и обрабатываю данные, необходимые для инициализации элементов структуры VMQUERY. (Файлы VMQuery.cpp и VMQuery.h содержатся в каталоге 14-VMMap на компакт-диске, прилагаемом к книге.) Чтобы не объяснять подробности обработки данных «на пальцах», я снабдил тексты программ массой комментариев, вольно разбросанных по всему коду.

VMQuery.cpp

```
/*
Modуль: VMQuery.cpp
Автор: Copyright (c) 2000, Джейфри Рихтер (Jeffrey Richter)
*/
#include "..\CmnHdr.h" /* см. приложение A */
```

Рис. 14-3. Функция VMQuery

Рис. 14-3. продолжение

```
#include <windowsx.h>
#include "VMQuery.h"

//////////////////////////////  

// вспомогательная структура
typedef struct {
    SIZE_T RgnSize;
    DWORD dwRgnStorage; // MEM_*: Free, Image, Mapped, Private
    DWORD dwRgnBlocks;
    DWORD dwRgnGuardBlks; // если > 0, в регионе содержится стек потока
    BOOL fRgnIsAStack; // TRUE, если в регионе содержится стек потока
} VMQUERY_HELP;

// глобальная статическая переменная, содержащая значение – гранулярность выделения
// памяти на данном типе процессора; инициализируется при первом вызове VMQuery
static DWORD gs_dwAllocGran = 0;

//////////////////////////////  

// эта функция проходит по всем блокам в регионе
// и инициализирует структуру найденными значениями
static BOOL VMQueryHelp(HANDLE hProcess, LPCVOID pvAddress,
    VMQUERY_HELP *pVMQHelp) {

    // каждый элемент содержит атрибут защиты страницы
    // (например, 0=зарезервирована, PAGE_NOACCESS, PAGE_READWRITE и т. д.)
    DWORD dwProtectBlock[4] = { 0 };

    ZeroMemory(pVMQHelp, sizeof(*pVMQHelp));

    // получаем базовый адрес региона, включающего переданный адрес памяти
    MEMORY_BASIC_INFORMATION mbi;
    BOOL fOk = (VirtualQueryEx(hProcess, pvAddress, &mbi, sizeof(mbi)) == sizeof(mbi));

    if (!fOk)
        return(fOk); // неверный адрес памяти, сообщаем об ошибке

    // проходим по региону, начиная с его базового адреса
    // (который никогда не изменится)
    PVOID pvRgnBaseAddress = mbi.AllocationBase;

    // начинаем с первого блока в регионе
    // (соответствующая переменная будет изменяться в цикле)
    PVOID pvAddressBlk = pvRgnBaseAddress;

    // запоминаем тип физической памяти, переданной данному блоку
    pVMQHelp->dwRgnStorage = mbi.Type;

    for (;;) {
        // получаем информацию о текущем блоке
        // ...
    }
}
```

см. след. стр.

Рис. 14-3. продолжение

```

f0k = (VirtualQueryEx(hProcess, pvAddressBlk, &mbi, sizeof(mbi)) == sizeof(mbi));
if (!f0k)
    break; // не удалось получить информацию; прекращаем цикл

// проверяем, принадлежит ли текущий блок запрошенному региону
if (mbi.AllocationBase != pvRgnBaseAddress)
    break; // блок принадлежит следующему региону; прекращаем цикл

// блок принадлежит запрошенному региону

// следующий оператор if служит для обнаружения стеков в Windows 98; в этой
// системе стеки размещаются в последних 4 блоках региона: "зарезервированный",
// PAGE_NOACCESS, PAGE_READWRITE и еще один "зарезервированный"
if (pVMQHelp->dwRgnBlocks < 4) {
    // если это блок 0-3, запоминаем тип защиты блока в массиве
    dwProtectBlock[pVMQHelp->dwRgnBlocks] =
        (mbi.State == MEM_RESERVE) ? 0 : mbi.Protect;
} else {
    // мы уже просмотрели 4 блока в этом регионе;
    // смещаем вниз элементы массива с атрибутами защиты
    MoveMemory(&dwProtectBlock[0], &dwProtectBlock[1],
               sizeof(dwProtectBlock) - sizeof(DWORD));

    // добавляем новые значения атрибутов защиты в конец массива
    dwProtectBlock[3] = (mbi.State == MEM_RESERVE) ? 0 : mbi.Protect;
}

pVMQHelp->dwRgnBlocks++;           // увеличиваем счетчик блоков
                                    // в этом регионе на 1
pVMQHelp->RgnSize += mbi.RegionSize; // добавляем размер блока к размеру региона

// если блок имеет флаг PAGE_GUARD, добавляем 1 к счетчику блоков
// с этим флагом
if ((mbi.Protect & PAGE_GUARD) == PAGE_GUARD)
    pVMQHelp->dwRgnGuardBlks++;

// Делаем наиболее вероятное предположение о типе физической памяти,
// переданной данному блоку. Стопроцентной гарантии дать нельзя,
// потому что некоторые блоки могли быть преобразованы из MEM_IMAGE
// в MEM_PRIVATE или из MEM_MAPPED в MEM_PRIVATE; MEM_PRIVATE в любой
// момент может быть замещен на MEM_IMAGE или MEM_MAPPED.
if (pVMQHelp->dwRgnStorage == MEM_PRIVATE)
    pVMQHelp->dwRgnStorage = mbi.Type;

// получаем адрес следующего блока
pvAddressBlk = (PVOID) ((PBYTE) pvAddressBlk + mbi.RegionSize);
}

// Обследовав регион, думаем: не стек ли это?
// Windows 2000: да - если в регионе содержится хотя бы 1 блок с флагом PAGE_GUARD.
// Windows 9x:   да - если в регионе содержится хотя бы 4 блока,

```

Рис. 14-3. продолжение

```

// и они имеют такие атрибуты:
// 3-й блок от конца: зарезервирован
// 2-й блок от конца: PAGE_NOACCESS
// 1-й блок от конца: PAGE_READWRITE
// последний блок: зарезервирован
pVMQHelp->fRgnIsAStack =
    (pVMQHelp->dwRgnGuardBlks > 0)      ||
    ((pVMQHelp->dwRgnBlocks >= 4)        &&
     (dwProtectBlock[0] == 0)                &&
     (dwProtectBlock[1] == PAGE_NOACCESS)   &&
     (dwProtectBlock[2] == PAGE_READWRITE)  &&
     (dwProtectBlock[3] == 0));

return(TRUE);
}

///////////////////////////////



BOOL VMQuery(HANDLE hProcess, LPCVOID pvAddress, PVMQUERY pVMQ) {
    if (gs_dwAllocGran == 0) {
        // если это первый вызов, надо выяснить гранулярность
        // выделения памяти в данной системе
        SYSTEM_INFO sinfo;
        GetSystemInfo(&sinfo);
        gs_dwAllocGran = sinfo.dwAllocationGranularity;
    }

    ZeroMemory(pVMQ, sizeof(*pVMQ));

    // получаем MEMORY_BASIC_INFORMATION для переданного адреса
    MEMORY_BASIC_INFORMATION mbi;
    BOOL fOk = (VirtualQueryEx(hProcess, pvAddress, &mbi, sizeof(mbi))
    == sizeof(mbi));

    if (!fOk)
        return(fOk); // неверный адрес памяти, сообщаем об ошибке

    // структура MEMORY_BASIC_INFORMATION содержит действительную
    // информацию – пора заполнить элементы нашей структуры VMQUERY

    // во-первых, заполним элементы, описывающие состояние блока;
    // данные по региону получим позже
    switch (mbi.State) {
        case MEM_FREE: // свободный блок (незарезервированный)
            pVMQ->pvBlkBaseAddress = NULL;
            pVMQ->BlkSize = 0;
            pVMQ->dwBlkProtection = 0;
            pVMQ->dwBlkStorage = MEM_FREE;
            break;
        case MEM_RESERVE: // зарезервированный блок, которому не передана физическая память
            pVMQ->pvBlkBaseAddress = mbi.BaseAddress;
    }
}

```

см. след. стр.

Рис. 14-3. продолжение

```

pVMQ->BlkSize = mbi.RegionSize;

// Для блока, которому не передана физическая память, элемент mbi.Protect
// недействителен. Поэтому мы покажем, что зарезервированный блок унаследовал
// атрибут защиты того региона, в котором он содержится.
pVMQ->dwBlkProtection = mbi.AllocationProtect;
pVMQ->dwBlkStorage = MEM_RESERVE;
break;

case MEM_COMMIT: // зарезервированный блок, которому
                  // передана физическая память
    pVMQ->pvBlkBaseAddress = mbi.BaseAddress;
    pVMQ->BlkSize = mbi.RegionSize;
    pVMQ->dwBlkProtection = mbi.Protect;
    pVMQ->dwBlkStorage = mbi.Type;
    break;

default:
    DebugBreak();
    break;
}

// теперь заполняем элементы, относящиеся к региону
VMQUERY_HELP VMQHelp;
switch (mbi.State) {
    case MEM_FREE: // свободный блок (незарезервированный)
        pVMQ->pvRgnBaseAddress = mbi.BaseAddress;
        pVMQ->dwRgnProtection = mbi.AllocationProtect;
        pVMQ->RgnSize = mbi.RegionSize;
        pVMQ->dwRgnStorage = MEM_FREE;
        pVMQ->dwRgnBlocks = 0;
        pVMQ->dwRgnGuardBlks = 0;
        pVMQ->fRgnIsAStack = FALSE;
        break;

    case MEM_RESERVE: // зарезервированный блок, которому не передана физическая память
        pVMQ->pvRgnBaseAddress = mbi.AllocationBase;
        pVMQ->dwRgnProtection = mbi.AllocationProtect;

        // чтобы получить полную информацию по региону, нам придется
        // пройти по всем его блокам
        VMQueryHelp(hProcess, pvAddress, &VMQHelp);

        pVMQ->RgnSize = VMQHelp.RgnSize;
        pVMQ->dwRgnStorage = VMQHelp.dwRgnStorage;
        pVMQ->dwRgnBlocks = VMQHelp.dwRgnBlocks;
        pVMQ->dwRgnGuardBlks = VMQHelp.dwRgnGuardBlks;
        pVMQ->fRgnIsAStack = VMQHelp.fRgnIsAStack;
        break;

    case MEM_COMMIT: // зарезервированный блок, которому передана физическая память
        pVMQ->pvRgnBaseAddress = mbi.AllocationBase;

```

Рис. 14-3. продолжение

```

pVMQ->dwRgnProtection = mbi.AllocationProtect;

// чтобы получить полную информацию по региону, нам придется
// пройти по всем его блокам
VMQueryHelp(hProcess, pvAddress, &VMQHelp);

pVMQ->RgnSize      = VMQHelp.RgnSize;
pVMQ->dwRgnStorage = VMQHelp.dwRgnStorage;
pVMQ->dwRgnBlocks  = VMQHelp.dwRgnBlocks;
pVMQ->dwRgnGuardBlks = VMQHelp.dwRgnGuardBlks;
pVMQ->fRgnIsAStack = VMQHelp.fRgnIsAStack;
break;

default:
    DebugBreak();
    break;
}

return(fOk);
}

//////////////////////////// Конец файла //////////////////////////////

```

VMQuery.h

```

/******************************************************************************/
Модуль: VMQuery.h
Автор: Copyright (c) 2000, Джейфри Рихтер (Jeffrey Richter)
*******/

typedef struct {
    // информация о регионе
    PVOID  pvRgnBaseAddress;
    DWORD   dwRgnProtection;    // PAGE_*
    SIZE_T  RgnSize;
    DWORD   dwRgnStorage;       // MEM_*: Free, Image, Mapped, Private
    DWORD   dwRgnBlocks;
    DWORD   dwRgnGuardBlks;     // если > 0, регион содержит стек потока
    BOOL    fRgnIsAStack;        // TRUE, если регион содержит стек потока
    // информация о блоке
    PVOID  pvBlkBaseAddress;
    DWORD   dwBlkProtection;    // PAGE_*
    SIZE_T  BlkSize;
    DWORD   dwBlkStorage;       // MEM_*: Free, Reserve, Image, Mapped, Private
} VMQUERY, *PVMQUERY;

//////////////////////////// Конец файла //////////////////////////////

BOOL VMQuery(HANDLE hProcess, LPCVOID pvAddress, PVMQUERY pVMQ);

//////////////////////////// Конец файла //////////////////////////////

```

Программа-пример VMMap

Эта программа, «14 VMMap.exe» (см. листинг на рис. 14-4), просматривает свое адресное пространство и показывает содержащиеся в нем регионы и блоки, присутствующие в регионах. Файлы исходного кода и ресурсов этой программы находятся в каталоге 14-VMMap на компакт-диске, прилагаемом к книге. После запуска VMMap на экране появляется следующее окно.

Virtual Memory Map (PID=1508 "14 VMMap.exe")					
00000000	Free	65536			
00010000	Private	4096	1	-RW-	
00011000	Free	61440			
00020000	Private	4096	1	-RW-	
00021000	Free	61440			
00030000	Private	1048576	3	-RW-	Thread Stack
00130000	Private	1048576	2	-RW-	
00230000	Mapped	65536	2	-RW-	
00240000	Mapped	90112	1	-R--	\Device\HarddiskVolume1\WINNT\system32\unicode.n
00256000	Free	40960			
00260000	Mapped	192512	1	-R--	\Device\HarddiskVolume1\WINNT\system32\locale.nl
0028F000	Free	4096			
00290000	Mapped	266240	1	-R--	\Device\HarddiskVolume1\WINNT\system32\sortkey.n
002D1000	Free	61440			
002E0000	Mapped	16384	1	-R--	\Device\HarddiskVolume1\WINNT\system32\sorttbls.
002E4000	Free	49152			
002F0000	Mapped	819200	4	ER--	
003B8000	Free	294912			
00400000	Image	106496	5	ERWC	C:\Documents and Settings\Jeffrey Richter\My Doc
0041A000	Free	24576			
00420000	Mapped	274432	1	-R--	
00463000	Free	53248			
00470000	Mapped	3145728	2	ER--	
00770000	Private	4096	1	-RW-	
00771000	Free	61440			
00780000	Private	4096	1	-RW-	
00781000	Free	61440			
00790000	Private	65536	2	-RW-	
007A0000	Mapped	8192	1	-R--	\Device\HarddiskVolume1\WINNT\system32\ctype.nls
007A2000	Free	1759174656			
69550000	Image	45056	4	ERWC	C:\WINNT\System32\PSAPI.dll
6955B000	Free	243159040			
77D40000	Image	458752	5	ERWC	C:\WINNT\system32\RPCRT4.DLL
77DB0000	Image	352256	5	ERWC	C:\WINNT\system32\ADVAPI32.dll
77E06000	Free	40960			
77E10000	Image	401408	4	ERWC	C:\WINNT\system32\USER32.dll
77E72000	Free	57344			
77E80000	Image	733184	4	ERWC	C:\WINNT\system32\KERNEL32.dll
77F33000	Free	53248			
77F40000	Image	241664	4	ERWC	C:\WINNT\system32\GDI32.DLL
77F7B000	Free	20480			
77F80000	Image	491520	5	ERWC	C:\WINNT\System32\ntdll.dll

Карты виртуальной памяти, представленные в главе 13 в таблицах 13-2, 13-3 и 13-4, созданы с помощью именно этой программы.

Каждый элемент в списке — результат вызова моей функции *VMQuery*. Основной цикл программы VMMap (в функции *Refresh*) выглядит так:

```

BOOL fOk = TRUE;
PVOID pvAddress = NULL;
:
while (fOk) {

    VMQUERY vmq;
    fOk = VMQuery(hProcess, pvAddress, &vmq);

    if (fOk) {
        // формируем строку для вывода на экран
        // и добавляем ее в окно списка
        TCHAR szLine[1024];
        ConstructRgnInfoLine(hProcess, &vmq, szLine, sizeof(szLine));
        ListBox_AddString(hwndLB, szLine);

        if (fExpandRegions) {

```

```

        for (DWORD dwBlock = 0; fOk && (dwBlock < vmq.dwRgnBlocks);
             dwBlock++) {

            ConstructBlkInfoLine(&vmq, szLine, sizeof(szLine));
            ListBox_AddString(hwndLB, szLine);

            // получаем адрес следующего региона
            pvAddress = ((PBYTE) pvAddress + vmq.BlkSize);
            if (dwBlock < vmq.dwRgnBlocks - 1) {
                // нельзя запрашивать информацию о памяти за последним блоком
                fOk = VMQuery(hProcess, pvAddress, &vmq);
            }
        }
    }
    // получаем адрес следующего региона
    pvAddress = ((PBYTE) vmq.pvRgnBaseAddress + vmq.RgnSize);
}
}

```

Этот цикл начинает работу с виртуального адреса NULL и заканчивается, когда *VMQuery* возвращает FALSE, что указывает на невозможность дальнейшего просмотра адресного пространства процесса. На каждой итерации цикла вызывается функция *ConstructRgnInfoLine*; она заполняет символьный буфер информацией о регионе. Потом эти данные вносятся в список.

В основной цикл вложен еще один цикл — он позволяет получать информацию о каждом блоке текущего региона. На каждой итерации из данного цикла вызывается функция *ConstructBlkInfoLine*, заполняющая символьный буфер информацией о блоках региона. Эти данные тоже добавляются к списку. В общем, с помощью функции *VMQuery* просматривать адресное пространство процесса очень легко.



VMMap.cpp

```

 ****
Модуль: VMMap.cpp
Автор: Copyright (c) 2000, Джейфри Рихтер (Jeffrey Richter)
****

#include "..\CmnHdr.h"      /* см. приложение A */
#include <psapi.h>
#include <windowsx.h>
#include <tchar.h>
#include <stdio.h>           // для доступа к sprintf
#include "..\04-ProcessInfo\Toolhelp.h"
#include "Resource.h"
#include "VMQuery.h"

///////////

DWORD g_dwProcessId = 0; // какой процесс надо пройти?
BOOL g_fExpandRegions = FALSE;
CToolhelp g_toolhelp;

```

Рис. 14-4. Программа-пример VMMap

см. след. стр.

Рис. 14-4. продолжение

```

// GetMappedFileName имеется только в Windows 2000 (в PSAPI.DLL);
// если эта функция в системе есть, используем именно ее
typedef DWORD (WINAPI* PFNGETMAPPEDFILENAME)(HANDLE, PVOID, PTSTR, DWORD);
static PFNGETMAPPEDFILENAME g_pfnGetMappedFileName = NULL;

///////////////////////////////



// я использовал эту функцию, чтобы получить карты памяти, приведенные в книге
void CopyControlToClipboard(HWND hwnd) {
    TCHAR szClipData[128 * 1024] = { 0 };

    int nCount = ListBox_GetCount(hwnd);
    for (int nNum = 0; nNum < nCount; nNum++) {
        TCHAR szLine[1000];
        ListBox_GetText(hwnd, nNum, szLine);
        _tcscat(szClipData, szLine);
        _tcscat(szClipData, TEXT("\r\n"));
    }

    OpenClipboard(NULL);
    EmptyClipboard();

    // буфер обмена принимает только данные, находящиеся в блоке, выделенном
    // функцией GlobalAlloc с флагами GMEM_MOVEABLE и GMEM_DDESHARE
    HGLOBAL hClipData = GlobalAlloc(GMEM_MOVEABLE | GMEM_DDESHARE,
        sizeof(TCHAR) * (_tcslen(szClipData) + 1));
    PTSTR pClipData = (PTSTR) GlobalLock(hClipData);

    _tcscpy(pClipData, szClipData);

#ifdef UNICODE
    BOOL fOk = (SetClipboardData(CF_UNICODETEXT, hClipData) == hClipData);
#else
    BOOL fOk = (SetClipboardData(CF_TEXT, hClipData) == hClipData);
#endif
    CloseClipboard();

    if (!fOk) {
        GlobalFree(hClipData);
        chMB("Error putting text on the clipboard");
    }
}

///////////////////////////////



PCTSTR GetMemStorageText(DWORD dwStorage) {

    PCTSTR p = TEXT("Unknown");
    switch (dwStorage) {
        case MEM_FREE:   p = TEXT("Free    "); break;
        case MEM_RESERVE: p = TEXT("Reserve"); break;
        case MEM_IMAGE:  p = TEXT("Image   "); break;
    }
}

```

Рис. 14-4. продолжение

```

case MEM_MAPPED: p = TEXT("Mapped "); break;
case MEM_PRIVATE: p = TEXT("Private"); break;
}
return(p);
}

//////////////////////////////



PTSTR GetProtectText(DWORD dwProtect, PTSTR szBuf, BOOL fShowFlags) {

PCTSTR p = TEXT("Unknown");
switch (dwProtect & ~(PAGE_GUARD | PAGE_NOCACHE | PAGE_WRITECOMBINE)) {
case PAGE_READONLY: p = TEXT("-R--"); break;
case PAGE_READWRITE: p = TEXT("-RW-"); break;
case PAGE_WRITECOPY: p = TEXT("-RWC"); break;
case PAGE_EXECUTE: p = TEXT("E---"); break;
case PAGE_EXECUTE_READ: p = TEXT("ER--"); break;
case PAGE_EXECUTE_READWRITE: p = TEXT("ERW-"); break;
case PAGE_EXECUTE_WRITECOPY: p = TEXT("ERWC"); break;
case PAGE_NOACCESS: p = TEXT("----"); break;
}
_tcscpy(szBuf, p);
if (fShowFlags) {
    _tcscat(szBuf, TEXT(" "));
    _tcscat(szBuf, (dwProtect & PAGE_GUARD) ? TEXT("G") : TEXT("-"));
    _tcscat(szBuf, (dwProtect & PAGE_NOCACHE) ? TEXT("N") : TEXT("-"));
    _tcscat(szBuf, (dwProtect & PAGE_WRITECOMBINE) ? TEXT("W") : TEXT("-"));
}
return(szBuf);
}

//////////////////////////////



void ConstructRgnInfoLine(HANDLE hProcess, PVMQUERY pVMQ, PTSTR szLine, int nMaxLen) {

_stprintf(szLine, TEXT("%p %s %16u "), pVMQ->pvRgnBaseAddress,
    GetMemStorageText(pVMQ->dwRgnStorage), pVMQ->RgnSize);
if (pVMQ->dwRgnStorage != MEM_FREE) {
    wsprintf(_tcschr(szLine, 0), TEXT("%5u "), pVMQ->dwRgnBlocks);
    GetProtectText(pVMQ->dwRgnProtection, _tcschr(szLine, 0), FALSE);
}

_tcscat(szLine, TEXT(" "));

// пытаемся получить полное имя модуля для этого региона
int nLen = _tcslen(szLine);
if (pVMQ->pvRgnBaseAddress != NULL) {
    MODULEENTRY32 me = { sizeof(me) };
    if (g_toolhelp.ModuleFind(pVMQ->pvRgnBaseAddress, &me)) {
        lstrcat(&szLine[nLen], me.szExePath);
    } else {
}
}
}

```

см. след. стр.

Рис. 14-4. продолжение

```

// это не модуль; проверяем, не является ли он файлом, проецируемым в память
if (g_pfnGetMappedFileName != NULL) {
    DWORD d = g_pfnGetMappedFileName(hProcess,
        pVMQ->pvRgnBaseAddress, szLine + nLen, nMaxLen - nLen);
    if (d == 0) {
        // Примечание: GetMappedFileName модифицирует строку при неудаче
        szLine[nLen] = 0;
    }
}
}

if (pVMQ->fRgnIsAStack) {
    _tcscat(szLine, TEXT("Thread Stack"));
}
}

///////////////////////////////
void ConstructBlkInfoLine(PVMQUERY pVMQ, PTSTR szLine, int nMaxLen) {

    _stprintf(szLine, TEXT(" %p %s %16u "),
        pVMQ->pvBlkBaseAddress, GetMemStorageText(pVMQ->dwBlkStorage),
        pVMQ->BlkSize);

    if (pVMQ->dwBlkStorage != MEM_FREE) {
        GetProtectText(pVMQ->dwBlkProtection, _tcschr(szLine, 0), TRUE);
    }
}

///////////////////////////////
void Refresh(HWND hwndLB, DWORD dwProcessId, BOOL fExpandRegions) {

    // очищаем окно списка и создаем в нем горизонтальную полосу прокрутки
    ListBox_ResetContent(hwndLB);
    ListBox_SetHorizontalExtent(hwndLB, 300 * LOWORD(GetDialogBaseUnits()));

    // выполняется ли еще процесс?
    HANDLE hProcess = OpenProcess(PROCESS_QUERY_INFORMATION, FALSE, dwProcessId);

    if (hProcess == NULL) {
        ListBox_AddString(hwndLB, TEXT("")); // пустая строка (так лучше на вид)
        ListBox_AddString(hwndLB,
            TEXT("    The process ID identifies a process that is not running"));
        return;
    }

    // получаем новый снимок процесса
    g_toolhelp.CreateSnapshot(TH32CS_SNAPALL, dwProcessId);

    // просматриваем виртуальное адресное пространство, добавляя элементы в окно списка
}

```

Рис. 14-4. продолжение

```

BOOL fOk = TRUE;
PVOID pvAddress = NULL;

SetWindowRedraw(hwndLB, FALSE);
while (fOk) {

    VMQUERY vmq;
    fOk = VMQuery(hProcess, pvAddress, &vmq);

    if (fOk) {
        // формируем строку для вывода на экран и добавляем ее в окно списка
        TCHAR szLine[1024];
        ConstructRgnInfoLine(hProcess, &vmq, szLine, sizeof(szLine));
        ListBox_AddString(hwndLB, szLine);

        if (fExpandRegions) {
            for (DWORD dwBlock = 0; fOk && (dwBlock < vmq.dwRgnBlocks);
                 dwBlock++) {

                ConstructBlkInfoLine(&vmq, szLine, sizeof(szLine));
                ListBox_AddString(hwndLB, szLine);

                // получаем адрес следующего региона
                pvAddress = ((PBYTE) pvAddress + vmq.BlkSize);
                if (dwBlock < vmq.dwRgnBlocks - 1) {
                    // нельзя запрашивать информацию о памяти за последним блоком
                    fOk = VMQuery(hProcess, pvAddress, &vmq);
                }
            }
        }
        // получаем адрес следующего региона
        pvAddress = ((PBYTE) vmq.pvRgnBaseAddress + vmq.RgnSize);
    }
}
SetWindowRedraw(hwndLB, TRUE);
CloseHandle(hProcess);

///////////////////////////////
BOOL Dlg_OnInitDialog(HWND hwnd, HWND hwndFocus, LPARAM lParam) {
    chSETDLGICONS(hwnd, IDI_VMMAP);

    // показываем в заголовке окна, какой процесс мы просматриваем
    TCHAR szCaption[MAX_PATH * 2];
    GetWindowText(hwnd, szCaption, chDIMOF(szCaption));
    g_toolhelp.CreateSnapshot(TH32CS_SNAPALL, g_dwProcessId);
    PROCESSENTRY32 pe = { sizeof(pe) };
    wsprintf(&szCaption[lstrlen(szCaption)], TEXT(" (PID=%u \\"%s\\")"),
             g_dwProcessId, g_toolhelp.ProcessFind(g_dwProcessId, &pe)
             ? pe.szExeFile : TEXT("unknown"));
}

```

см. след. стр.

Рис. 14-4. продолжение

```
SetWindowText(hwnd, szCaption);

// VMMMap сообщает столько информации, что
// окно лучше сразу отмасштабировать по максимуму
ShowWindow(hwnd, SW_MAXIMIZE);

// принудительно обновляем окно списка
Refresh(GetDlgItem(hwnd, IDC_LISTBOX), g_dwProcessId, g_fExpandRegions);
return(TRUE);
}

//////////void Dlg_OnSize(HWND hwnd, UINT state, int cx, int cy) {

// окно списка всегда занимает всю клиентскую область
SetWindowPos(GetDlgItem(hwnd, IDC_LISTBOX), NULL, 0, 0, cx, cy,
SWP_NOZORDER);
}

//////////void Dlg_OnCommand(HWND hwnd, int id, HWND hwndCtl, UINT codeNotify) {

switch (id) {
    case IDCANCEL:
        EndDialog(hwnd, id);
        break;

    case ID_REFRESH:
        Refresh(GetDlgItem(hwnd, IDC_LISTBOX),
            g_dwProcessId, g_fExpandRegions);
        break;

    case ID_EXPANDREGIONS:
        g_fExpandRegions = g_fExpandRegions ? FALSE: TRUE;
        Refresh(GetDlgItem(hwnd, IDC_LISTBOX),
            g_dwProcessId, g_fExpandRegions);
        break;
    case ID_COPYTOCLIPBOARD:
        CopyControlToClipboard(GetDlgItem(hwnd, IDC_LISTBOX));
        break;
}
}

//////////INT_PTR WINAPI Dlg_Proc(HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam) {

switch (uMsg) {
    chHANDLE_DLGMMSG(hwnd, WM_INITDIALOG, Dlg_OnInitDialog);
    chHANDLE_DLGMMSG(hwnd, WM_COMMAND,      Dlg_OnCommand);
}
```

Рис. 14-4. продолжение

```

    chHANDLE_DLMSG(hwnd, WM_SIZE, Dlg_OnSize);
}
return(FALSE);
}

/////////////////////////////// Конец файла //////////////////////////////

int WINAPI _tWinMain(HINSTANCE hinstExe, HINSTANCE, PTSTR pszCmdLine, int) {
    CToolhelp::EnableDebugPrivilege();

    // пытаемся загрузить PSAPI.DLL и получить адрес GetMappedFileName
    HMODULE hmodPSAPI = LoadLibrary(TEXT("PSAPI"));
    if (hmodPSAPI != NULL) {
#ifdef UNICODE
        g_pfnGetMappedFileName = (PFNGETMAPPEDFILENAME)
            GetProcAddress(hmodPSAPI, "GetMappedFileNameW");
#else
        g_pfnGetMappedFileName = (PFNGETMAPPEDFILENAME)
            GetProcAddress(hmodPSAPI, "GetMappedFileNameA");
#endif
    }

    g_dwProcessId = _ttoi(pszCmdLine);
    if (g_dwProcessId == 0) {
        g_dwProcessId = GetCurrentProcessId();
    }

    DialogBox(hinstExe, MAKEINTRESOURCE(IDD_VMMAP), NULL, Dlg_Proc);

    if (hmodPSAPI != NULL)
        FreeLibrary(hmodPSAPI); // выгружаем PSAPI.DLL, если мы ее загружали

    return(0);
}

/////////////////////////////// Конец файла //////////////////////////////

```

Использование виртуальной памяти в приложениях

В Windows три механизма работы с памятью:

- виртуальная память — наиболее подходящая для операций с большими массивами объектов или структур;
- проецируемые в память файлы — наиболее подходящие для операций с большими потоками данных (обычно из файлов) и для совместного использования данных несколькими процессами на одном компьютере;
- кучи — наиболее подходящие для работы с множеством малых объектов.

В этой главе мы обсудим первый метод — виртуальную память. Остальные два метода (проецируемые в память файлы и кучи) рассматриваются соответственно в главах 17 и 18.

Функции, работающие с виртуальной памятью, позволяют напрямую резервировать регион адресного пространства, передавать ему физическую память (из страниценного файла) и присваивать любые допустимые атрибуты защиты.

Резервирование региона в адресном пространстве

Для этого предназначена функция *VirtualAlloc*:

```
PVOID VirtualAlloc(
    PVOID pvAddress,
    SIZE_T dwSize,
    DWORD fdwAllocationType,
    DWORD fdwProtect);
```

В первом параметре, *pvAddress*, содержится адрес памяти, указывающий, где именно система должна зарезервировать адресное пространство. Обычно в этом параметре передают NULL, тем самым сообщая функции *VirtualAlloc*, что система, ведущая учет свободных областей, должна зарезервировать регион там, где, по ее мнению, будет лучше. Поэтому нет никаких гарантий, что система станет резервировать регионы, начиная с нижних адресов или, наоборот, с верхних. Однако с помощью флага *MEM_TOP_DOWN* (о нем речь впереди) Вы можете сказать свое веское слово.

Для большинства программистов возможность выбора конкретного адреса резервируемого региона — нечто совершенно новое. Вспомните, как это делалось раньше: операционная система просто находила подходящий по размеру блок памяти, выделяла этот блок и возвращала его адрес. Но поскольку каждый процесс владеет собственным адресным пространством, у Вас появляется возможность указывать операционной системе желательный базовый адрес резервируемого региона.

Допустим, нужно выделить регион, начиная с «отметки» 50 Мб в адресном пространстве процесса. Тогда параметр *pvAddress* должен быть равен 52 428 800 ($50 \times 1024 \times 1024$). Если по этому адресу можно разместить регион требуемого размера, система зарезервирует его и вернет соответствующий адрес. Если же по этому адресу свободного пространства недостаточно или просто нет, система не удовлетворит запрос, и функция *VirtualAlloc* вернет NULL. Адрес, передаваемый в *pvAddress*, должен укладываться в границы раздела пользовательского режима Вашего процесса, так как иначе *VirtualAlloc* потерпит неудачу и вернет NULL.

Как я уже говорил в главе 13, регионы всегда резервируются с учетом гранулярности выделения памяти (64 Кб для существующих реализаций Windows). Поэтому, если Вы попытаетесь зарезервировать регион по адресу 19 668 992 ($300 \times 65 536 + 8192$), система округлит этот адрес до ближайшего меньшего числа, кратного 64 Кб, и на самом деле зарезервирует регион по адресу 19 660 800 ($300 \times 65 536$).

Если *VirtualAlloc* в состоянии удовлетворить запрос, она возвращает базовый адрес зарезервированного региона. Если параметр *pvAddress* содержал конкретный адрес, функция возвращает этот адрес, округленный при необходимости до меньшей величины, кратной 64 Кб.

Второй параметр функции *VirtualAlloc* — *dwSize* — указывает размер резервируемого региона в байтах. Поскольку система резервирует регионы только порциями, кратными размеру страницы, используемой данным процессором, то попытка зарезервировать, скажем, 62 Кб даст регион размером 64 Кб (если размер страницы составляет 4, 8 или 16 Кб).

Третий параметр, *fdwAllocationType*, сообщает системе, что именно Вы хотите сделать: зарезервировать регион или передать физическую память. (Такое разграничение необходимо, поскольку *VirtualAlloc* позволяет не только резервировать регионы, но и передавать им физическую память.) Поэтому, чтобы зарезервировать регион адресного пространства, в этом параметре нужно передать идентификатор **MEM_RESERVE**.

Если Вы хотите зарезервировать регион и не собираетесь освобождать его в ближайшее время, попробуйте выделить его в диапазоне самых старших — насколько это возможно — адресов. Тогда регион не окажется где-нибудь в середине адресного пространства процесса, что позволит не допустить вполне вероятной фрагментации этого пространства. Чтобы зарезервировать регион по самым старшим адресам, при вызове функции *VirtualAlloc* в параметре *pvAddress* передайте NULL, а в параметре *fdwAllocationType* — флаг **MEM_TOP_DOWN**.



В Windows 98 флаг **MEM_TOP_DOWN** игнорируется.

Последний параметр, *fdwProtect*, указывает атрибут защиты, присваиваемый региону. Заметьте, что атрибут защиты, связанный с регионом, не влияет на переданную память, отображаемую на этот регион. Но если ему не передана физическая память, то — какой бы атрибут защиты у него ни был — любая попытка обращения по одному из адресов в этом диапазоне приведет к нарушению доступа для данного потока.

Резервируя регион, присваивайте ему тот атрибут защиты, который будет чаще всего использоваться с памятью, передаваемой региону. Скажем, если Вы собираетесь передать региону физическую память с атрибутом защиты **PAGE_READWRITE** (этот атрибут самый распространенный), то и резервировать его следует с тем же атрибутом. Система работает эффективнее, когда атрибут защиты региона совпадает с атрибутом защиты передаваемой памяти.

Вы можете использовать любой из следующих атрибутов защиты: PAGE_NOACCESS, PAGE_READWRITE, PAGE_READONLY, PAGE_EXECUTE, PAGE_EXECUTE_READ или PAGE_EXECUTE_READWRITE. Но указывать атрибуты PAGE_WRITECOPY или PAGE_EXECUTE_WRITECOPY нельзя: иначе функция *VirtualAlloc* не зарезервирует регион и вернет NULL. Кроме того, при резервировании региона флаги PAGE_GUARD, PAGE_WRITECOMBINE или PAGE_NOCACHE применять тоже нельзя — они присваиваются только передаваемой памяти.



Windows 98 поддерживает лишь атрибуты защиты PAGE_NOACCESS, PAGE_READONLY и PAGE_READWRITE. Попытка резервирования региона с атрибутом PAGE_EXECUTE или PAGE_EXECUTE_READ дает регион с атрибутом PAGE_READONLY. А указав PAGE_EXECUTE_READWRITE, Вы получите регион с атрибутом PAGE_READWRITE.

Передача памяти зарезервированному региону

Зарезервировав регион, Вы должны, прежде чем обращаться по содержащимся в нем адресам, передать ему физическую память. Система выделяет региону физическую память из страничного файла на жестком диске. При этом она, разумеется, учитывает свойственный данному процессору размер страниц и передает ресурсы постранично.

Для передачи физической памяти вызовите *VirtualAlloc* еще раз, указав в параметре *fdwAllocationType* не MEM_RESERVE, а MEM_COMMIT. Обычно указывают тот же атрибут защиты, что и при резервировании региона, хотя можно задать и другой.

Затем сообщите функции *VirtualAlloc*, по какому адресу и сколько физической памяти следует передать. Для этого в параметр *pvAddress* запишите желательный адрес, а в параметр *dwSize* — размер физической памяти в байтах. Передавать физическую память сразу всему региону необязательно.

Посмотрим, как это делается на практике. Допустим, программа работает на процессоре x86 и резервирует регион размером 512 Кб, начиная с адреса 5 242 880. Затем Вы передаете физическую память блоку размером 6 Кб, отстоящему от начала зарезервированного региона на 2 Кб. Тогда вызовите *VirtualAlloc* с флагом MEM_COMMIT так:

```
VirtualAlloc((PVOID) (5242880 + (2 * 1024)), 6 * 1024,  
    MEM_COMMIT, PAGE_READWRITE);
```

В этом случае система передаст 8 Кб физической памяти в диапазоне адресов от 5 242 880 до 5 251 071 (т. е. 5 242 880 + 8 Кб – 1 байт), и обе переданные страницы получат атрибут защиты PAGE_READWRITE. Страница является минимальной единицей памяти, которой можно присвоить собственные атрибуты защиты. Следовательно, в регионе могут быть страницы с разными атрибутами защиты (скажем, одна — с атрибутом PAGE_READWRITE, другая — с атрибутом PAGE_READONLY).

Резервирование региона с одновременной передачей физической памяти

Иногда нужно одновременно зарезервировать регион и передать ему физическую память. В таком случае *VirtualAlloc* можно вызвать следующим образом:

```
PVOID pvMem = VirtualAlloc(NULL, 99 * 1024,  
    MEM_RESERVE | MEM_COMMIT, PAGE_READWRITE);
```

Этот вызов содержит запрос на выделение региона размером 99 Кб и передачу ему 99 Кб физической памяти. Обрабатывая этот запрос, система сначала просматривает адресное пространство Вашего процесса, пытаясь найти непрерывную незарезервированную область размером не менее 100 Кб (на машинах с 4-килобайтовыми страницами) или 104 Кб (на машинах с 8-килобайтовыми страницами).

Система просматривает адресное пространство потому, что в *pvAddress* указан NULL. Если бы он содержал конкретный адрес памяти, система проверила бы только его — подходит ли по размеру расположение за ним адресное пространство. Оказалось он недостаточным, функция *VirtualAlloc* вернула бы NULL.

Если системе удастся зарезервировать подходящий регион, она передает ему физическую память. И регион, и переданная память получают один атрибут защиты — в данном случае PAGE_READWRITE.

Наконец, функция *VirtualAlloc* возвращает виртуальный адрес этого региона, который потом записывается в переменную *pvMem*. Если же система не найдет в адресном пространстве подходящую область или не сумеет передать ей физическую память, *VirtualAlloc* вернет NULL.

Конечно, при резервировании региона с одновременной передачей ему памяти можно указать в параметре *pvAddress* конкретный адрес или запросить систему подобрать свободное место в верхней части адресного пространства процесса. Последнее реализуют так: в параметр *pvAddress* заносят NULL, а значение параметра *fdwAllocationType* комбинируют с флагом MEM_TOP_DOWN.

В какой момент региону передают физическую память

Допустим, Вы разрабатываете программу — электронную таблицу, которая поддерживает до 200 строк при 256 колонках. Для каждой ячейки необходима своя структура CELldata, описывающая ее (ячейки) содержимое. Простейший способ работы с двухмерной матрицей ячеек, казалось бы, — взять и объявить в программе такую переменную:

```
CELLDATA CellData[200][256];
```

Но если размер структуры CELldata будет хотя бы 128 байтов, матрица потребует 6 553 600 ($200 \times 256 \times 128$) байтов физической памяти. Не многовато ли? Тем более что большинство пользователей заполняет данными всего несколько ячеек. Выходит, матрицы здесь крайне неэффективны.

Поэтому электронные таблицы реализуют на основе других методов управления структурами данных, используя, например, связанные списки. В этом случае структуры CELldata создаются только для ячеек, содержащих какие-то данные. И поскольку большая часть ячеек в таблице остается незадействованной, Вы экономите колоссальные объемы памяти. Но это значительно усложняет доступ к содержимому ячеек. Чтобы, допустим, выяснить содержимое ячейки на пересечении строки 5 и колонки 10, придется пройти по всей цепочке связанных списков. В итоге метод связанных списков работает медленнее, чем метод, основанный на объявлении матрицы.

К счастью, виртуальная память позволяет найти компромисс между «лобовым» объявлением двухмерной матрицы и реализацией связанных списков. Тем самым можно совместить простоту и высокую скорость доступа к ячейкам, предлагаемую «матричным» методом, с экономным расходованием памяти, заложенным в метод связанных списков.

Вот что надо сделать в своей программе.

1. Зарезервировать достаточно большой регион, чтобы при необходимости в него мог поместиться весь массив структур CELldata. Для резервирования региона физическая память не нужна.
2. Когда пользователь вводит данные в ячейку, вычислить адрес в зарезервированном регионе, по которому должна быть записана соответствующая структура CELldata. Естественно, физическая память на этот регион пока не отображается, и поэтому любое обращение к памяти по данному адресу вызовет нарушение доступа.
3. Передать по адресу, полученному в п. 2, физическую память, необходимую для размещения одной структуры CELldata. (Так как система допускает передачу памяти отдельным частям зарезервированного региона, в нем могут находиться и отображеные, и не отображеные на физическую память участки.)
4. Инициализировать элементы новой структуры CELldata.

Теперь, спроектировав физическую память на нужный участок зарезервированного региона, программа может обратиться к нему, не вызывая при этом нарушения доступа. Таким образом, метод, основанный на использовании виртуальной памяти, самый оптимальный, поскольку позволяет передавать физическую память только по мере ввода данных в ячейки электронной таблицы. И ввиду того, что большая часть ячеек в электронной таблице обычно пуста, то и большая часть зарезервированного региона физическую память не получает.

Но при использовании виртуальной памяти все же возникает одна проблема: приходится определять, когда именно зарезервированному региону надо передавать физическую память. Если пользователь всего лишь редактирует данные, уже содержащиеся в ячейке, в передаче физической памяти необходимости нет — это было сделано в момент первого заполнения ячейки.

Нельзя забывать и о размерности страниц памяти. Попытка передать физическую память для единственной структуры CELldata (как в п. 2 предыдущего списка) приведет к передаче полной страницы памяти. Но в этом, как ни странно, есть свое преимущество: передав физическую память под одну структуру CELldata, Вы одновременно выделите ее и следующим структурам CELldata. Когда пользователь начнет заполнять следующую ячейку (а так обычно и бывает), Вам, может, и не придется передавать дополнительную физическую память.

Определить, надо ли передавать физическую память части региона, можно четырьмя способами.

- Всегда пытаться передавать физическую память. Вместо того чтобы проверять, отображен данный участок региона на физическую память или нет, заставьте программу передавать память при каждом вызове функции *VirtualAlloc*. Ведь система сама делает такую проверку и, если физическая память спроектирована на данный участок, повторной передачи не допускает. Это простейший путь, но при каждом изменении структуры CELldata придется вызывать функцию *VirtualAlloc*, что, естественно, скажется на скорости работы программы.
- Определять (с помощью *VirtualQuery*), передана ли уже физическая память адресному пространству, содержащему структуру CELldata. Если да, больше ничего не делать; нет — вызвать *VirtualAlloc* для передачи памяти. Этот метод на деле еще хуже: он не только замедляет выполнение, но и увеличивает размер программы из-за дополнительных вызовов *VirtualQuery*.
- Вести учет, каким страницам передана физическая память, а каким — нет. Это повысит скорость работы программы: Вы избежите лишних вызовов *Virtual-*

Alloc, а программа сможет — быстрее, чем система — определять, передана ли память. Недостаток этого метода в том, что придется отслеживать передачу страниц; иногда это просто, но может быть и очень сложно — все зависит от конкретной задачи.

- Самое лучшее — использовать структурную обработку исключений (SEH). SEH — одно из средств операционной системы, с помощью которого она уведомляет приложения о возникновении определенных событий. В общем и целом, Вы добавляете в программу обработчик исключений, после чего любая попытка обращения к участку, которому не передана физическая память, заставляет систему уведомлять программу о возникшей проблеме. Далее программа передает память нужному участку и сообщает системе, что та должна повторить операцию, вызвавшую исключение. На этот раз доступ к памяти пройдет успешно, и программа, как ни в чем не бывало, продолжит работу. Таким образом, Ваша задача заметно упрощается (а значит, упрощается и код); кроме того, программа, не делая больше лишних вызовов, выполняется быстрее. Но подробное рассмотрение механизма структурной обработки исключений мы отложим до глав 23, 24 и 25. Программа-пример *Spreadsheet* в главе 25 продемонстрирует именно этот способ использования виртуальной памяти.

Возврат физической памяти и освобождение региона

Для возврата физической памяти, отображенной на регион, или освобождения всего региона адресного пространства используется функция *VirtualFree*:

```
BOOL VirtualFree(
    LPVOID pvAddress,
    SIZE_T dwSize,
    DWORD fdwFreeType);
```

Рассмотрим простейший случай вызова этой функции — для освобождения зарезервированного региона. Когда процессу больше не нужна физическая память, переданная региону, зарезервированный регион и всю связанную с ним физическую память можно освободить единственным вызовом *VirtualFree*.

В этом случае в параметр *pvAddress* надо поместить базовый адрес региона, т. е. значение, возвращенное функцией *VirtualAlloc* после резервирования данного региона. Системе известен размер региона, расположенного по указанному адресу, поэтому в параметре *dwSize* можно передать 0. Фактически Вы даже обязаны это сделать, иначе вызов *VirtualFree* не даст результата. В третьем параметре (*fdwFreeType*) передайте идентификатор *MEM_RELEASE*; это приведет к возврату системе всей физической памяти, отображенной на регион, и к освобождению самого региона. Освобождая регион, Вы должны освободить и зарезервированное под него адресное пространство. Нельзя выделить регион размером, допустим, 128 Кб, а потом освободить только 64 Кб: надо освобождать все 128 Кб.

Если Вам нужно, не освобождая регион, вернуть в систему часть физической памяти, переданной региону, для этого тоже следует вызвать *VirtualFree*. При этом ее параметр *pvAddress* должен содержать адрес, указывающий на первую возвращаемую страницу. Кроме того, в параметре *dwSize* задайте количество освобождаемых байтов, а в параметре *fdwFreeType* — идентификатор *MEM_DECOMMIT*.

Как и передача, возврат памяти осуществляется с учетом размерности страниц. Иначе говоря, задание адреса, указывающего на середину страницы, приведет к возврату всей страницы. Разумеется, то же самое произойдет, если суммарное значение

параметров *pvAddress* и *dwSize* выпадет на середину страницы. Так что системе возвращаются все страницы, попадающие в диапазон от *pvAddress* до *pvAddress* + *dwSize*.

Если же *dwSize* равен 0, а *pvAddress* указывает на базовый адрес выделенного региона, *VirtualFree* вернет системе весь диапазон выделенных страниц. После возврата физической памяти освобожденные страницы доступны любому другому процессу, а попытка обращения к адресам, уже не связанным с физической памятью, приведет к нарушению доступа.

В какой момент физическую память возвращают системе

На практике уловить момент, подходящий для возврата памяти, — штука непростая. Вернемся к примеру с электронной таблицей. Если программа работает на машине с процессором x86, размер каждой страницы памяти — 4 Кб, т. е. на одной странице умещается 32 (4096 / 128) структуры CELLDATA. Если пользователь удаляет содержимое элемента *CellData[0][1]*, Вы можете вернуть страницу памяти, но только при условии, что ячейки в диапазоне от *CellData[0][0]* до *CellData[0][31]* тоже не используются. Как об этом узнать? Проблема решается несколькими способами.

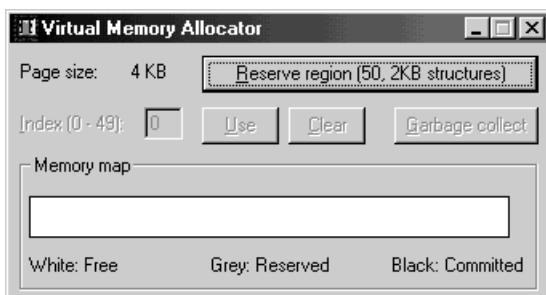
- Несомненно, простейший выход — сделать структуру CELLDATA такой, чтобы она занимала ровно одну страницу. Тогда, как только данные в какой-либо из этих структур больше не нужны, Вы могли бы просто возвращать системе соответствующую страницу. Даже если бы структура данных занимала не одну, а несколько страниц, возврат памяти все равно был бы делом несложным. Но кто же пишет программы, подгоняя размер структур под размер страниц памяти — у разных процессоров они разные.
 - Гораздо практичнее вести учет используемых структур данных. Для экономии памяти можно применить битовую карту. Так, имея массив из 100 структур, Вы создаете дополнительный массив из 100 битов. Изначально все биты сброшены (обнулены), указывая тем самым, что ни одна структура не используется. По мере заполнения структур Вы устанавливаете соответствующие биты (т. е. приравниваете их единице). Отпада необходимость в какой-то структуре — сбросьте ее бит и проверьте биты соседних структур, расположенных в пределах той же страницы памяти. Если и они не используются, страницу можно вернуть системе.
 - В последнем варианте реализуется функция сбора мусора. Как известно, система при первой передаче физической памяти обнуляет все байты на переданной странице. Чтобы воспользоваться этим обстоятельством, предусмотрите в своей структуре элемент типа BOOL (назовем его, скажем, *fInUse*) и всякий раз, когда структура записывается в переданную память, устанавливайте его в TRUE. При выполнении программы Вы будете периодически вызывать функцию сбора мусора, которая должна просматривать все структуры. Для каждой структуры (и существующей, и той, которая может быть создана) функция сначала определяет, передана ли под нее память; если да, то проверяет значение *fInUse*. Если он равен 0, структура не используется; TRUE — структура занята. Проверив все структуры, расположенные в пределах заданной страницы, функция сбора мусора вызывает *VirtualFree*, чтобы освободить память, — если, конечно, на этой странице нет используемых структур.
- Функцию сбора мусора можно вызывать сразу после того, как необходимость в одной из структур отпадет, но делать так не стоит, поскольку функция каждый раз просматривает все структуры — и существующие, и те, которые могут быть созданы. Оптимальный путь — реализовать эту функцию как поток с бо-

лее низким уровнем приоритета. Это позволит не отнимать время у потока, выполняющего основную программу. А когда основная программа будет простоять или ее поток займется файловым вводом-выводом, вот тогда система и выделит время функции сборки мусора.

Лично я предпочитаю первые два способа. Однако, если Ваши структуры компактны (меньше одной страницы памяти), советую применять последний метод.

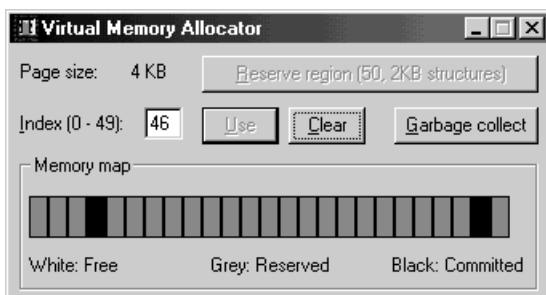
Программа-пример VMAlloc

Эта программа, «15 VMAlloc.exe» (см. листинг на рис. 15-1), демонстрирует применение механизма виртуальной памяти для управления массивом структур. Файлы исходного кода и ресурсов этой программы находятся в каталоге 15-VMAlloc на компакт-dиске, прилагаемом к книге. После запуска VMAlloc на экране появится диалоговое окно, показанное ниже.



Изначально для массива не резервируется никакого региона, и все адресное пространство, предназначенное для него, свободно, что и отражено на карте памяти. Если щелкнуть кнопку Reserve Region (50,2KB Structures), программа VMAlloc вызовет *VirtualAlloc* для резервирования региона, что сразу отразится на карте памяти. После этого станут активными и остальные кнопки в диалоговом окне.

Теперь в поле можно ввести индекс и щелкнуть кнопку Use. При этом по адресу, где должен располагаться указанный элемент массива, передается физическая память. Далее карта памяти вновь перерисовывается и уже отражает состояние региона, зарезервированного под весь массив. Когда Вы, зарезервировав регион, вновь щелкните кнопку Use, чтобы пометить элементы 7 и 46 как занятые, окно (при выполнении программы на процессоре с размером страниц по 4 Кб) будет выглядеть так:

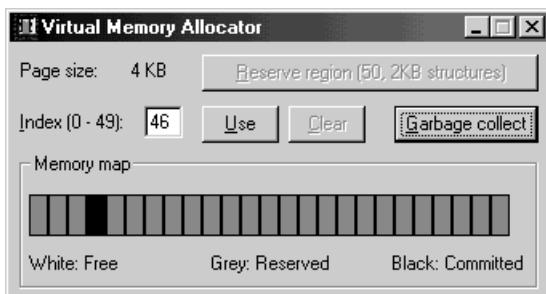


Любой элемент массива, помеченный как занятый, можно освободить щелчком кнопки Clear. Но это не приведет к возврату физической памяти, переданной под элемент массива. Дело в том, что каждая страница содержит несколько структур и освобождение одной структуры не влечет за собой освобождения других. Если бы память была возвращена, то пропали бы и данные, содержащиеся в остальных струк-

турах. И поскольку выбор кнопки Clear никак не сказывается на физической памяти региона, карта памяти после освобождения элемента не меняется.

Однако освобождение структуры приводит к тому, что ее элемент *fInUse* устанавливается в FALSE. Это нужно для того, чтобы функция сбора мусора могла вернуть не используемую больше физическую память. Кнопка Garbage Collect, если Вы еще не догадались, заставляет программу VMAlloc выполнить функцию сбора мусора. Для упрощения программы я не стал выделять эту функцию в отдельный поток.

Чтобы посмотреть, как работает функция сбора мусора, очистите элемент массива с индексом 46. Заметьте, что карта памяти пока не изменилась. Теперь щелкните кнопку Garbage Collect. Программа освободит страницу, содержащую 46-й элемент, и карта памяти сразу же обновится, как показано ниже. Заметьте, что функцию *GarbageCollect* можно легко использовать в любых других приложениях. Я реализовал ее так, чтобы она работала с массивами структур данных любого размера; при этом структура не обязательно должна полностью умещаться на странице памяти. Единственное требование заключается в том, что первый элемент структуры должен быть значением типа BOOL, которое указывает, задействована ли данная структура.



И, наконец, хоть это и не видно на экране, закрытие окна приводит к возврату всей переданной памяти и освобождению зарезервированного региона.

Но есть в этой программе еще одна особенность, о которой я пока не упоминал. Программе приходится определять состояние памяти в адресном пространстве региона в трех случаях.

- После изменения индекса. Программе нужно включить кнопку Use и отключить кнопку Clear (или наоборот).
- В функции сбора мусора. Программа, прежде чем проверять значение флага *fInUse*, должна определить, была ли передана память.
- При обновлении карты памяти. Программа должна выяснить, какие страницы свободны, какие — зарезервированы, а какие — переданы.

Все эти проверки VMAlloc осуществляет через функцию *VirtualQuery*, рассмотренную в предыдущей главе.

```


VMAloc.cpp

/*
Modуль: VMAloc.cpp
Автор: Copyright (c) 2000, Джейфри Рихтер (Jeffrey Richter)
*/

```

Рис. 15-1. Программа-пример VMAloc

Рис. 15-1. продолжение

```

#include "..\CmnHdr.h"      /* см. приложение A */
#include <WindowsX.h>
#include <tchar.h>
#include "Resource.h"

///////////////////////////////



// переменная для хранения размера страниц на данном процессоре
UINT g_uPageSize = 0;

// пример структуры данных, используемой для массива
typedef struct {
    BOOL fInUse;
    BYTE bOtherData[2048 - sizeof(BOOL)];
} SOMEDATA, *PSOMEDATA;

// число структур в массиве
#define MAX_SOMEDATA     (50)

// указатель на массив структур данных
PSOMEDATA g_pSomeData = NULL;

// прямоугольная область в окне, занимаемая картой памяти
RECT g_rcMemMap;

///////////////////////////////



BOOL Dlg_OnInitDialog(HWND hwnd, HWND hwndFocus, LPARAM lParam) {

    chSETDLGICONS(hwnd, IDI_VMALLOC);

    // инициализируем диалоговое окно с отключением
    // пока недоступных элементов управления
    EnableWindow(GetDlgItem(hwnd, IDC_INDEXTEXT), FALSE);
    EnableWindow(GetDlgItem(hwnd, IDC_INDEX), FALSE);
    EnableWindow(GetDlgItem(hwnd, IDC_USE), FALSE);
    EnableWindow(GetDlgItem(hwnd, IDC_CLEAR), FALSE);
    EnableWindow(GetDlgItem(hwnd, IDC_GARBAGECOLLECT), FALSE);

    // получаем координаты поля вывода карты памяти
    GetWindowRect(GetDlgItem(hwnd, IDC_MEMMAP), &g_rcMemMap);
    MapWindowPoints(NULL, hwnd, (LPPPOINT) &g_rcMemMap, 2);

    // уничтожаем временное окно, которое определяет позицию
    // поля вывода для карты памяти
    DestroyWindow(GetDlgItem(hwnd, IDC_MEMMAP));

    // выводим в диалоговое окно размер страницы (просто для сведения)
    TCHAR szBuf[10];
    wsprintf(szBuf, TEXT("(%d KB)", g_uPageSize / 1024));
    SetDlgItemText(hwnd, IDC_PAGESIZE, szBuf);
}

```

см. след. стр.

Рис. 15-1. продолжение

```

// инициализируем поле ввода
SetDlgItemInt(hwnd, IDC_INDEX, 0, FALSE);

return(TRUE);
}

///////////////////////////////
void Dlg_OnDestroy(HWND hwnd) {

if (g_pSomeData != NULL)
    VirtualFree(g_pSomeData, 0, MEM_RELEASE);
}

///////////////////////////////

VOID GarbageCollect(PVOID pvBase, DWORD dwNum, DWORD dwStructSize) {

static DWORD s_uPageSize = 0;

if (s_uPageSize == 0) {
    // получаем размер страниц на данном процессоре
    SYSTEM_INFO si;
    GetSystemInfo(&si);
    s_uPageSize = si.dwPageSize;
}

UINT uMaxPages = dwNum * dwStructSize / g_uPageSize;

for (UINT uPage = 0; uPage < uMaxPages; uPage++) {
    BOOL fAnyAllocsInThisPage = FALSE;
    UINT uIndex      = uPage * g_uPageSize / dwStructSize;
    UINT uIndexLast = uIndex + g_uPageSize / dwStructSize;

    for (; uIndex < uIndexLast; uIndex++) {
        MEMORY_BASIC_INFORMATION mbi;
        VirtualQuery(&g_pSomeData[uIndex], &mbi, sizeof(mbi));
        fAnyAllocsInThisPage = ((mbi.State == MEM_COMMIT) &&
                               * (PBOOL) ((PBYTE) pvBase + dwStructSize * uIndex));

        // прекращаем проверку этой страницы, так как ее нельзя вернуть
        if (fAnyAllocsInThisPage) break;
    }

    if (!fAnyAllocsInThisPage) {
        // на этой странице нет структур, возвращаем ее
        VirtualFree(&g_pSomeData[uIndexLast - 1], dwStructSize, MEM_DECOMMIT);
    }
}

/////////////////////////////

```

Рис. 15-1. продолжение

```

void Dlg_OnCommand(HWND hwnd, int id, HWND hwndCtl, UINT codeNotify) {

    UINT uIndex = 0;

    switch (id) {
        case IDCANCEL:
            EndDialog(hwnd, id);
            break;

        case IDC_reserve:
            // резервируем адресное пространство,
            // достаточное для размещения массива структур
            g_pSomeData = (PSOMEDATA) VirtualAlloc(NULL,
                MAX_SOMEDATA * sizeof(SOMEDATA), MEM_RESERVE, PAGE_READWRITE);

            // отключаем кнопку Reserve и включаем остальные
            // элементы управления
            EnableWindow(GetDlgItem(hwnd, IDC_reserve), FALSE);
            EnableWindow(GetDlgItem(hwnd, IDC_INDEXTEXT), TRUE);
            EnableWindow(GetDlgItem(hwnd, IDC_INDEX), TRUE);
            EnableWindow(GetDlgItem(hwnd, IDC_USE), TRUE);
            EnableWindow(GetDlgItem(hwnd, IDC_GARBAGECOLLECT), TRUE);

            // переводим фокус в поле ввода индекса
            SetFocus(GetDlgItem(hwnd, IDC_INDEX));

            // объявляем поле вывода карты памяти недействительным (для его перерисовки)
            InvalidateRect(hwnd, &g_rcMemMap, FALSE);
            break;

        case IDC_INDEX:
            if (codeNotify != EN_CHANGE)
                break;

            uIndex = GetDlgItemInt(hwnd, id, NULL, FALSE);
            if ((g_pSomeData != NULL) && chINRANGE(0, uIndex, MAX_SOMEDATA - 1)) {
                MEMORY_BASIC_INFORMATION mbi;
                VirtualQuery(&g_pSomeData[uIndex], &mbi, sizeof(mbi));
                BOOL fOk = (mbi.State == MEM_COMMIT);
                if (fOk)
                    fOk = g_pSomeData[uIndex].fInUse;

                EnableWindow(GetDlgItem(hwnd, IDC_USE), !fOk);
                EnableWindow(GetDlgItem(hwnd, IDC_CLEAR), fOk);

            } else {
                EnableWindow(GetDlgItem(hwnd, IDC_USE), FALSE);
                EnableWindow(GetDlgItem(hwnd, IDC_CLEAR), FALSE);
            }
            break;
    }
}

```

см. след. стр.

Рис. 15-1. продолжение

```

case IDC_USE:
    uIndex = GetDlgItemInt(hwnd, IDC_INDEX, NULL, FALSE);
    if (chINRANGE(0, uIndex, MAX_SOMEDATA - 1)) {

        // Примечание: новые страницы всегда обнуляются системой
        VirtualAlloc(&g_pSomeData[uIndex], sizeof(SOMEDATA),
                     MEM_COMMIT, PAGE_READWRITE);

        g_pSomeData[uIndex].fInUse = TRUE;

        EnableWindow(GetDlgItem(hwnd, IDC_USE), FALSE);
        EnableWindow(GetDlgItem(hwnd, IDC_CLEAR), TRUE);

        // переводим фокус на кнопку Clear
        SetFocus(GetDlgItem(hwnd, IDC_CLEAR));

        // объявляем поле вывода карты памяти недействительным
        InvalidateRect(hwnd, &g_rcMemMap, FALSE);
    }
    break;

case IDC_CLEAR:
    uIndex = GetDlgItemInt(hwnd, IDC_INDEX, NULL, FALSE);
    if (chINRANGE(0, uIndex, MAX_SOMEDATA - 1)) {
        g_pSomeData[uIndex].fInUse = FALSE;
        EnableWindow(GetDlgItem(hwnd, IDC_USE), TRUE);
        EnableWindow(GetDlgItem(hwnd, IDC_CLEAR), FALSE);

        // переводим фокус на кнопку Use
        SetFocus(GetDlgItem(hwnd, IDC_USE));
    }
    break;

case IDC_GARBAGECOLLECT:
    GarbageCollect(g_pSomeData, MAX_SOMEDATA, sizeof(SOMEDATA));

    // объявляем поле вывода карты памяти недействительным
    InvalidateRect(hwnd, &g_rcMemMap, FALSE);
    break;
}

///////////////////////////////
void Dlg_OnPaint(HWND hwnd) { // перерисовывает карту памяти

    PAINTSTRUCT ps;
    BeginPaint(hwnd, &ps);

    UINT uMaxPages = MAX_SOMEDATA * sizeof(SOMEDATA) / g_uPageSize;
    UINT uMemMapWidth = g_rcMemMap.right - g_rcMemMap.left;
}

```

Рис. 15-1. продолжение

```

if (g_pSomeData == NULL) {

    // память еще предстоит зарезервировать
    Rectangle(ps.hdc, g_rcMemMap.left, g_rcMemMap.top,
              g_rcMemMap.right - uMemMapWidth % uMaxPages, g_rcMemMap.bottom);

} else {

    // проходим виртуальное адресное пространство, создавая карту памяти
    for (UINT uPage = 0; uPage < uMaxPages; uPage++) {

        UINT uIndex = uPage * g_uPageSize / sizeof(SOMEDATA);
        UINT uIndexLast = uIndex + g_uPageSize / sizeof(SOMEDATA);
        for (; uIndex < uIndexLast; uIndex++) {

            MEMORY_BASIC_INFORMATION mbi;
            VirtualQuery(&g_pSomeData[uIndex], &mbi, sizeof(mbi));

            int nBrush = 0;
            switch (mbi.State) {
                case MEM_FREE:   nBrush = WHITE_BRUSH; break;
                case MEM_RESERVE: nBrush = GRAY_BRUSH; break;
                case MEM_COMMIT:  nBrush = BLACK_BRUSH; break;
            }

            SelectObject(ps.hdc, GetStockObject(nBrush));
            Rectangle(ps.hdc,
                      g_rcMemMap.left + uMemMapWidth / uMaxPages * uPage,
                      g_rcMemMap.top,
                      g_rcMemMap.left + uMemMapWidth / uMaxPages * (uPage + 1),
                      g_rcMemMap.bottom);

        }
    }
}

EndPaint(hwnd, &ps);
}

///////////////////////////////
INT_PTR WINAPI Dlg_Proc(HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam) {

    switch (uMsg) {
        chHANDLE_DLGMMSG(hwnd, WM_INITDIALOG, Dlg_OnInitDialog);
        chHANDLE_DLGMMSG(hwnd, WM_COMMAND,      Dlg_OnCommand);
        chHANDLE_DLGMMSG(hwnd, WM_PAINT,        Dlg_OnPaint);
        chHANDLE_DLGMMSG(hwnd, WM_DESTROY,      Dlg_OnDestroy);
    }
    return(FALSE);
}

```

см. след. стр.

Рис. 15-1. продолжение

```
///////////  
int WINAPI _tWinMain(HINSTANCE hinstExe, HINSTANCE, LPTSTR pszCmdLine, int) {  
  
    // получаем размер страниц для данного процессора  
    SYSTEM_INFO si;  
    GetSystemInfo(&si);  
    g_uPageSize = si.dwPageSize;  
  
    DialogBox(hinstExe, MAKEINTRESOURCE(IDD_VMALLOC), NULL, Dlg_Proc);  
    return(0);  
}  
  
/////////// Конец файла ///////////
```

Изменение атрибутов защиты

Хоть это и не принято, но атрибуты защиты, присвоенные странице или страницам переданной физической памяти, можно изменять. Допустим, Вы разработали код для управления связанным списком, узлы (вершины) которого хранятся в зарезервированном регионе. При желании можно написать функции, которые обрабатывали бы связанные списки и изменяли бы атрибуты защиты переданной памяти при старте на PAGE_READWRITE, а при завершении — обратно на PAGE_NOACCESS.

Сделав так, Вы защитите данные в связанном списке от возможных «жучков», скрытых в программе. Например, если какой-то блок кода в Вашей программе из-за наличия «блуждающего» указателя обратится к данным в связанном списке, возникнет нарушение доступа. Поэтому такой подход иногда очень полезен — особенно когда пытаешься найти трудноуловимую ошибку в своей программе.

Атрибуты защиты страницы памяти можно изменить вызовом *VirtualProtect*:

```
BOOL VirtualProtect(  
    PVOID pvAddress,  
    SIZE_T dwSize,  
    DWORD f1NewProtect,  
    PDWORD pf1OldProtect);
```

Здесь *pvAddress* указывает на базовый адрес памяти (который должен находиться в пользовательском разделе Вашего процесса), *dwSize* определяет число байтов, для которых Вы изменяете атрибут защиты, а *f1NewProtect* содержит один из идентификаторов PAGE_*, кроме PAGE_WRITECOPY и PAGE_EXECUTE_WRITECOPY.

Последний параметр, *pf1OldProtect*, содержит адрес переменной типа DWORD, в которую *VirtualProtect* заносит старое значение атрибута защиты для данной области памяти. В этом параметре (даже если Вас не интересует такая информация) нужно передать корректный адрес, иначе функция приведет к нарушению доступа.

Естественно, атрибуты защиты связаны с целыми страницами памяти и не могут присваиваться отдельным байтам. Поэтому, если на процессоре с четырехкилобайтными страницами вызвать *VirtualProtect*, например, так:

```
VirtualProtect(pvRgnBase + (3 * 1024), 2 * 1024,  
    PAGE_NOACCESS, &f1OldProtect);
```

то атрибут защиты PAGE_NOACCESS будет присвоен двум страницам памяти.

WINDOWS 98 Windows 98 поддерживает лишь атрибуты защиты PAGE_NOACCESS, PAGE_READONLY и PAGE_READWRITE. Попытка изменить атрибут защиты страницы на PAGE_EXECUTE или PAGE_EXECUTE_READ приведет к тому, что эта область памяти получит атрибут PAGE_READONLY. А указав атрибут PAGE_EXECUTE_READWRITE, Вы получите страницу с атрибутом PAGE_READWRITE.

Функцию *VirtualProtect* нельзя использовать для изменения атрибутов защиты страниц, диапазон которых охватывает разные зарезервированные регионы. В таких случаях *VirtualProtect* надо вызывать для каждого региона отдельно.

Сброс содержимого физической памяти

WINDOWS 98 Windows 98 не поддерживает сброс физической памяти.

Когда Вы модифицируете содержимое страниц физической памяти, система пытается как можно дольше хранить эти изменения в оперативной памяти. Однако, выполняя приложения, система постоянно получает запросы на загрузку в оперативную память страниц из EXE-файлов, DLL и/или страничного файла. Любой такой запрос заставляет систему просматривать оперативную память и выгружать модифицированные страницы в страничный файл.

Windows 2000 позволяет программам увеличить свою производительность за счет сброса физической памяти. Вы сообщаете системе, что данные на одной или нескольких страницах памяти не изменились. Если система в процессе поиска свободной страницы в оперативной памяти выбирает измененную страницу, то должна сначала записать ее в страничный файл. Эта операция отнимает довольно много времени и отрицательно сказывается на производительности. Поэтому в большинстве приложений желательно, чтобы система как можно дольше хранила модифицированные страницы в страничном файле.

Однако некоторые программы занимают блоки памяти на очень малое время, а потом им уже не требуется их содержимое. Для большего быстродействия программа может попросить систему не записывать определенные страницы в страничный файл. И тогда, если одна из этих страниц понадобится для других целей, системе не придется сохранять ее в страничном файле, что, естественно, повысит скорость работы программы. Такой отказ от страницы (или страниц) памяти называется *сбросом физической памяти* (*resetting of physical storage*) и инициируется вызовом функции *VirtualAlloc* с передачей ей в третьем параметре флага *MEM_RESET*.

Если страницы, на которые Вы ссылаетесь при вызове *VirtualAlloc*, находятся в страничном файле, система их удалит. Когда в следующий раз программа обратится к памяти, она получит новые страницы, инициализированные нулями. Если же Вы сбрасываете страницу, находящуюся в оперативной памяти, система помечает ее как неизменявшуюся, и она не записывается в страничный файл. Но, хотя ее содержимое не обнуляется, читать такую страницу памяти уже нельзя. Если системе не понадобится эта страница оперативной памяти, ее содержимое останется прежним. В ином случае система может забрать ее в свое распоряжение, и тогда обращение к этой странице приведет к тому, что система предоставит программе новую страницу, заполненную нулями. А поскольку этот процесс нам не подвластен, лучше считать, что после сброса страница содержит только мусор.

При сбросе физической памяти надо учитывать и несколько других моментов. Во-первых, когда Вы вызываете *VirtualAlloc*, базовый адрес обычно округляется до ближайшего меньшего значения, кратного размеру страниц, а количество байтов — до ближайшего большего значения, кратного той же величине. Такой механизм округления базового адреса и количества байтов был бы очень опасен при сбросе физической памяти; поэтому *VirtualAlloc* при передаче ей флага *MEM_RESET* округляет эти значения прямо наоборот. Допустим, в Вашей программе есть следующий исходный код:

```
PINT pnData = (PINT) VirtualAlloc(NULL, 1024,  
    MEM_RESERVE | MEM_COMMIT, PAGE_READWRITE);  
pn[0] = 100;  
pn[1] = 200;  
VirtualAlloc((PVOID) pnData, sizeof(int), MEM_RESET, PAGE_READWRITE);
```

Этот код передает одну страницу памяти, а затем сообщает, что первые четыре байта (*sizeof(int)*) больше не нужны и их можно сбросить. Однако, как и при любых других действиях с памятью, эта операция выполняется только над блоками памяти, размер которых кратен размеру страниц. В данном случае вызов завершится неудачно (*VirtualAlloc* вернет NULL). Почему? Дело в том, что при вызове *VirtualAlloc* Вы указали флаг *MEM_RESET* и базовый адрес, переданный функции, теперь округляется до ближайшего большего значения, кратного размеру страниц, а количество байтов — до ближайшего меньшего значения, кратного той же величине. Так делается, чтобы исключить случайную потерю важных данных. В предыдущем примере округление количества байтов до ближайшего меньшего значения дает 0, а эта величина недопустима.

Второе, о чем следует помнить при сбросе памяти, — флаг *MEM_RESET* нельзя комбинировать (логической операцией OR) ни с какими другими флагами. Следующий вызов всегда будет заканчиваться неудачно:

```
PVOID pvMem = VirtualAlloc(NULL, 1024,  
    MEM_RESERVE | MEM_COMMIT | MEM_RESET, PAGE_READWRITE);
```

Впрочем, комбинировать флаг *MEM_RESET* с другими флагами все равно бессмысленно.

И, наконец, последнее. Вызов *VirtualAlloc* с флагом *MEM_RESET* требует передачи корректного атрибута защиты страницы, даже несмотря на то что он не будет использоваться данной функцией.

Программа-пример MemReset

Эта программа, «15 MemReset.exe» (см. листинг на рис. 15-2), демонстрирует, как работает флаг *MEM_RESET*. Файлы исходного кода и ресурсов этой программы находятся в каталоге 15-MemReset на компакт-диске, прилагаемом к книге.

Первое, что делает код этой программы, — резервирует регион и передает ему физическую память. Поскольку размер региона, переданный в *VirtualAlloc*, равен 1024 байтам, система автоматически округляет это значение до размера страницы. Затем функция *lstrcpy* копирует в этот буфер строку, и содержимое страницы оказывается измененным. Если система впоследствии считает, что ей нужна страница, содержащая наши данные, она запишет эту страницу в страницочный файл. Когда наша программа попытается считать эти данные, система автоматически загрузит страницу из страницочного файла в оперативную память.

После записи строки в страницу памяти наша программа спрашивает у пользователя, понадобятся ли еще эти данные. Если пользователь выбирает отрицательный

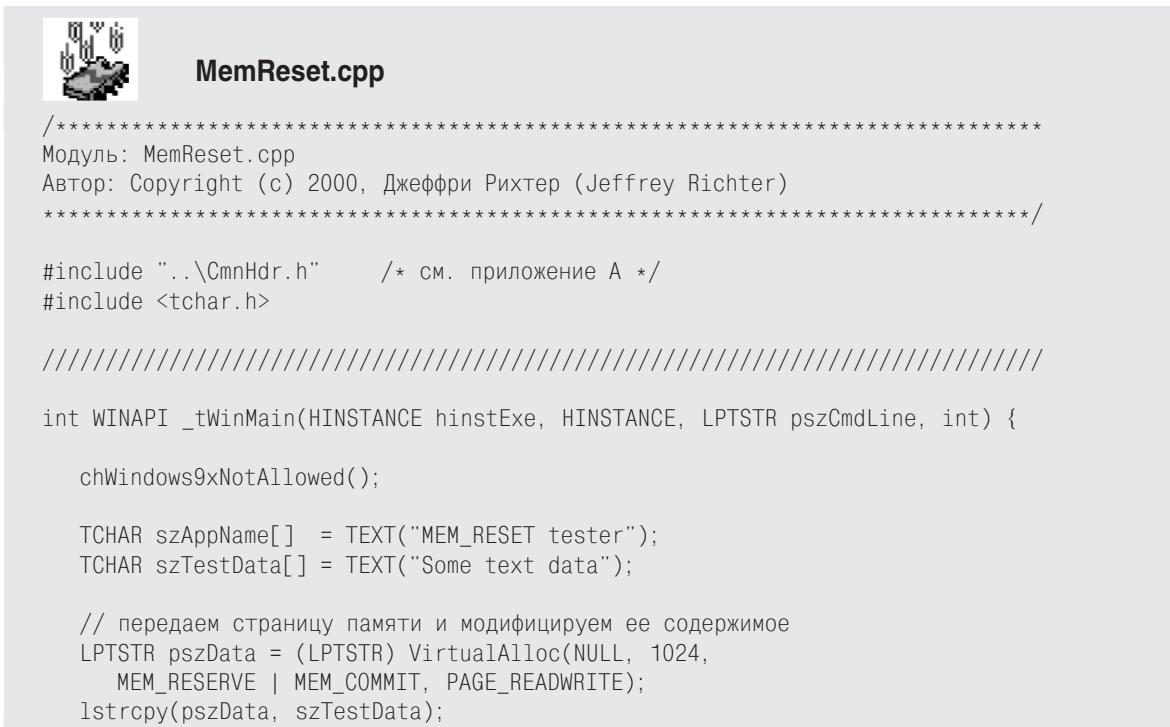
ответ (щелчком кнопки No), программа сообщает системе, что страница не изменилась, для чего вызывает *VirtualAlloc* с флагом *MEM_RESET*.

Для демонстрации того факта, что память действительно сброшена, смоделируем высокую нагрузку на оперативную память, для чего:

1. Получим общий размер оперативной памяти на компьютере вызовом *GlobalMemoryStatus*.
2. Передадим эту память вызовом *VirtualAlloc*. Данная операция выполняется очень быстро, поскольку система не выделяет оперативную память до тех пор, пока процесс не изменит какие-нибудь страницы.
3. Изменим содержимое только что переданных страниц через функцию *ZeroMemory*. Это создает высокую нагрузку на оперативную память, и отдельные страницы выгружаются в страничный файл.

Если пользователь захочет оставить данные, сброс не осуществляется, и при первой же попытке доступа к ним соответствующие страницы будут подгружаться в оперативную память из страничного файла. Если же пользователь откажется от этих данных, мы выполняем сброс памяти, система не записывает их в страничный файл, и это ускоряет выполнение программы.

После вызова *ZeroMemory* я сравниваю содержимое страницы данных со строкой, которая была туда записана. Если данные не сбрасывались, содержимое идентично, а если сбрасывались — то ли идентично, то ли нет. В моей программе содержимое никогда не останется прежним, поскольку я заставляю систему выгрузить все страницы оперативной памяти в страничный файл. Но если бы размер выгруженой области был меньше общего объема оперативной памяти, то не исключено, что исходное содержимое все равно осталось бы в памяти. Так что будьте осторожны!



```


MemReset.cpp

/*****
Modуль: MemReset.cpp
Автор: Copyright (c) 2000, Джейфри Рихтер (Jeffrey Richter)
*****/

#include "..\CmnHdr.h"      /* см. приложение A */
#include <tchar.h>

///////////

int WINAPI _tWinMain(HINSTANCE hinstExe, HINSTANCE, LPTSTR pszCmdLine, int) {
    chWindows9xNotAllowed();

    TCHAR szAppName[] = TEXT("MEM_RESET tester");
    TCHAR szTestData[] = TEXT("Some text data");

    // передаем страницу памяти и модифицируем ее содержимое
    LPTSTR pszData = (LPTSTR) VirtualAlloc(NULL, 1024,
        MEM_RESERVE | MEM_COMMIT, PAGE_READWRITE);
    lstrcpy(pszData, szTestData);
}

```

Рис. 15-1. Программа-пример *MemReset*

см. след. стр.

Рис. 15-1. продолжение

```

if (MessageBox(NULL, TEXT("Do you want to access this data later?"),
szAppName, MB_YESNO) == IDNO) {

    // Мы хотим сохранить эту страницу физической памяти в нашем
    // процессе, но ее данные нас больше не интересуют.
    // Скажем системе, что данные на этой странице не изменились.

    // Примечание: поскольку MEM_RESET разрушает данные, VirtualAlloc округляет
    // параметры с базовым адресом и размером до наиболее безопасных значений.
    // Вот пример:
    //     VirtualAlloc(pvData, 5000, MEM_RESET, PAGE_READWRITE)
    // сбросит 0 страниц на процессорах с размером страниц более 4 Кб
    // и 1 страницу на процессорах с четырехкилобайтовыми страницами.
    // Поэтому, чтобы вызов VirtualAlloc всегда был успешным, надо
    // сначала вызвать VirtualQuery и определить точный размер страницы.

    MEMORY_BASIC_INFORMATION mbi;
    VirtualQuery(pszData, &mbi, sizeof(mbi));
    VirtualAlloc(pszData, mbi.RegionSize, MEM_RESET, PAGE_READWRITE);
}

// передаем объем памяти, равный размеру оперативной памяти
MEMORYSTATUS mst;
GlobalMemoryStatus(&mst);
PVOID pvDummy = VirtualAlloc(NULL, mst.dwTotalPhys,
    MEM_RESERVE | MEM_COMMIT, PAGE_READWRITE);

// изменяем содержимое всех страниц в регионе, чтобы все страницы,
// модифицированные в оперативной памяти, записывались в станичный файл
ZeroMemory(pvDummy, mst.dwTotalPhys);

// сравниваем нашу страницу данных с тем,
// что было записано туда изначально
if (lstrcmp(pszData, szTestData) == 0) {

    // Данные на этой странице совпали с тем, что мы туда записывали.
    // Функция ZeroMemory заставила систему записать нашу
    // страницу в станичный файл.
    MessageBox(NULL, TEXT("Modified data page was saved."),
szAppName, MB_OK);
} else {

    // Данные на этой странице не совпадают с тем, что мы туда записывали.
    // ZeroMemory не заставила систему записать измененную страницу
    // в станичный файл.
    MessageBox(NULL, TEXT("Modified data page was NOT saved."),
szAppName, MB_OK);
}

return(0);
}

/////////////////////////////// Конец файла ///////////////////////////////

```

Механизм Address Windowing Extensions (только Windows 2000)

Жизнь идет вперед, и приложения требуют все больше и больше памяти — особенно серверные. Чем выше число клиентов, обращающихся к серверу, тем меньше его производительность. Для увеличения быстродействия серверное приложение должно хранить как можно больше своих данных в оперативной памяти и сбрасывать их на диск как можно реже. Другим классам приложений (базам данных, программам для работы с трехмерной графикой, математическими моделями и др.) тоже нужно манипулировать крупными блоками памяти. И всем этим приложениям уже тесно в 32-разрядном адресном пространстве.

Для таких приложений Windows 2000 предлагает новый механизм — Address Windowing Extensions (AWE). Создавая AWE, Microsoft стремилась к тому, чтобы приложения могли:

- работать с оперативной памятью, никогда не выгружаемой на диск операционной системой;
- обращаться к таким объемам оперативной памяти, которые превышают размеры соответствующих разделов в адресных пространствах их процессов.

AWE дает возможность приложению выделять себе один и более блоков оперативной памяти, невидимых в адресном пространстве процесса. Сделав это, приложение резервирует регион адресного пространства (с помощью *VirtualAlloc*), и он становится адресным окном (address window). Далее программа вызывает функцию, которая связывает адресное окно с одним из выделенных блоков оперативной памяти. Эта операция выполняется чрезвычайно быстро (обычно за пару микросекунд).

Через одно адресное окно единовременно доступен лишь один блок памяти. Это, конечно, усложняет программирование, так как при обращении к другому блоку приходится явно вызывать функции, которые как бы переключают адресное окно на следующий блок.

Вот пример, демонстрирующий использование AWE:

```
// сначала резервируем для адресного окна регион размером 1 Мб
ULONG_PTR ulRAMBytes = 1024 * 1024
PVOID pvWindow = VirtualAlloc(NULL, ulRAMBytes, MEM_RESERVE | MEM_PHYSICAL, PAGE_READWRITE);
// получаем размер страниц на данной процессорной платформе
SYSTEM_INFO sinf;
GetSystemInfo(&sinf);

// вычисляем, сколько страниц памяти нужно для нашего количества байтов
ULONG_PTR ulRAMPages = (ulRAMBytes + sinf.dwPageSize - 1) / sinf.dwPageSize;

// создаем соответствующий массив для номеров фреймов страниц
ULONG_PTR aRAMPages[ulRAMPages];

// выделяем страницы оперативной памяти (в полномочиях пользователя)
// должна быть разрешена блокировка страниц в памяти)
AllocateUserPhysicalPages(
    GetCurrentProcess(), // выделяем память для нашего процесса
    &ulRAMPages, // на входе: количество запрошенных страниц RAM,
                  // на выходе: количество выделенных страниц RAM
```

см. след. стр.

```
aRAMPages);           // на выходе: специфический массив,  
                      // идентифицирующий выделенные страницы  
  
// назначаем страницы оперативной памяти нашему окну  
MapUserPhysicalPages(pvWindow,    // адрес адресного окна  
                     ulRAMPages,          // число элементов в массиве  
                     aRAMPages);         // массив страниц RAM  
  
// обращаемся к этим страницам через виртуальный адрес pvWindow  
:  
  
// освобождаем блок страниц оперативной памяти  
FreeUserPhysicalPages(  
    GetCurrentProcess(), // освобождаем RAM, выделенную нашему процессу  
    &ulRAMPages,          // на входе: количество страниц RAM,  
                        // на выходе: количество освобожденных страниц RAM  
    aRAMPages);          // на входе: массив, идентифицирующий освобождаемые  
                        // страницы RAM  
  
// уничтожаем адресное окно  
VirtualFree(pvWindow, 0, MEM_RELEASE);
```

Как видите, пользоваться AWE несложно. А теперь хочу обратить Ваше внимание на несколько интересных моментов, связанных с этим фрагментом кода.

Вызов *VirtualAlloc* резервирует адресное окно размером 1 Мб. Обычно адресное окно гораздо больше. Вы должны выбрать его размер в соответствии с объемом блоков оперативной памяти, необходимых Вашему приложению. Но, конечно, размер такого окна ограничен размером самого крупного свободного (и непрерывного!) блока в адресном пространстве процесса. Флаг *MEM_RESERVED* указывает, что я просто резервирую диапазон адресов, а флаг *MEM_PHYSICAL* — что в конечном счете этот диапазон адресов будет связан с физической (оперативной) памятью. Механизм AWE требует, чтобы вся память, связываемая с адресным окном, была доступна для чтения и записи; поэтому в данном случае функции *VirtualAlloc* можно передать только один атрибут защиты — *PAGE_READWRITE*. Кроме того, нельзя пользоваться функцией *VirtualProtect* и пытаться изменять тип защиты этого блока памяти.

Для выделения блока в физической памяти надо вызвать функцию *AllocateUserPhysicalPages*:

```
BOOL AllocateUserPhysicalPages(  
    HANDLE hProcess,  
    PULONG_PTR pulRAMPages,  
    PULONG_PTR aRAMPages);
```

Она выделяет количество страниц оперативной памяти, заданное в значении, на которое указывает параметр *pulRAMPages*, и закрепляет эти страницы за процессом, определяемым параметром *hProcess*.

Операционная система назначает каждой странице оперативной памяти *номер фрейма страницы* (page frame number). По мере того как система отбирает страницы памяти, выделяемые приложению, она вносит соответствующие данные (номер фрейма страницы для каждой страницы оперативной памяти) в массив, на который указывает параметр *aRAMPages*. Сами по себе эти номера для приложения совершенно бесполезны; Вам не следует просматривать содержимое этого массива и тем бо-

лее что-либо менять в нем. Вы не узнаете, какие страницы оперативной памяти будут выделены под запрошенный блок, да это и не нужно. Когда эти страницы связываются с адресным окном, они появляются в виде непрерывного блока памяти. А что там система делает для этого, Вас не должно интересовать.

Когда функция *AllocateUserPhysicalPages* возвращает управление, значение в *pulRAMPages* сообщает количество фактически выделенных страниц. Обычно оно совпадает с тем, что Вы передаете функции, но может оказаться и поменьше.

Страницы оперативной памяти выделяются только процессу, из которого была вызвана данная функция; AWE не разрешает проецировать их на адресное пространство другого процесса. Поэтому такие блоки памяти нельзя разделять между процессами.



Конечно, оперативная память — ресурс драгоценный, и приложению может выделить лишь ее незадействованную часть. Не злоупотребляйте механизмом AWE: если Ваш процесс захватит слишком много оперативной памяти, это может привести к интенсивной перекачке страниц на диск и резкому падению производительности всей системы. Кроме того, это ограничит возможности системы в создании новых процессов, потоков и других ресурсов. (Мониторинг степени использования физической памяти можно реализовать через функцию *GlobalMemoryStatusEx*.)

AllocateUserPhysicalPages требует также, чтобы приложению была разрешена блокировка страниц в памяти (т. е. у пользователя должно быть право «Lock Pages in Memory»), а иначе функция потерпит неудачу. По умолчанию таким правом пользователи или их группы не наделяются. Оно назначается учетной записи Local System, которая обычно используется различными службами. Если Вы хотите запускать интерактивное приложение,зывающее *AllocateUserPhysicalPages*, администратор должен предоставить Вам соответствующее право еще до того, как Вы зарегистрируетесь в системе.

Теперь, создав адресное окно и выделив блок памяти, я связываю этот блок с окном вызовом функции *MapUserPhysicalPages*:

```
BOOL MapUserPhysicalPages(
    PVOID pvAddressWindow,
    ULONG_PTR ulRAMPages,
    PULONG_PTR aRAMPages);
```

Ее первый параметр, *pvAddressWindow*, определяет виртуальный адрес адресного окна, а последние два параметра, *ulRAMPages* и *aRAMPages*, сообщают, сколько страниц оперативной памяти должно быть видимо через адресное окно и что это за страницы. Если окно меньше связываемого блока памяти, функция потерпит неудачу.



Функция *MapUserPhysicalPages* отключает текущий блок оперативной памяти от адресного окна, если вместо параметра *aRAMPages* передается NULL. Вот пример:

```
// отключаем текущий блок RAM от адресного окна
MapUserPhysicalPages(pvWindow, ulRAMPages, NULL);
```

WINDOWS 2000 Связав блок оперативной памяти с адресным окном, Вы можете легко обращаться к этой памяти, просто ссылаясь на виртуальные адреса относительно базового адреса адресного окна (в моем примере это *puWindow*).

Когда необходимость в блоке памяти отпадет, освободите его вызовом функции *FreeUserPhysicalPages*:

```
BOOL FreeUserPhysicalPages(  
    HANDLE hProcess,  
    PULONG_PTR pulRAMPages,  
    PULONG_PTR aRAMPages);
```

В Windows 2000 право «Lock Pages in Memory» активизируется так:

1. Запустите консоль Computer Management MMC. Для этого щелкните кнопку Start, выберите команду Run, введите «compmgmt.msc /a» и щелкните кнопку OK.
2. Если в левой секции нет элемента Local Computer Policy, выберите из меню Console команду Add/Remove Snap-in. На вкладке Standalone в списке Snap-ins Added To укажите строку Computer Management (Local). Теперь щелкните кнопку Add, чтобы открыть диалоговое окно Add Standalone Snap-in. В списке Available Standalone Snap-ins укажите Select Group Policy и выберите кнопку Add. В диалоговом окне Select Group Policy Object просто щелкните кнопку Finish. Наконец, в диалоговом окне Add Standalone Snap-in щелкните кнопку Close, а в диалоговом окне Add/Remove Snap-in — кнопку OK. После этого в левой секции консоли Computer Management должен появиться элемент Local Computer Policy.
3. В левой секции консоли последовательно раскройте следующие элементы: Local Computer Policy, Computer Configuration, Windows Settings, Security Settings и Local Policies. Выберите User Rights Assignment.
4. В правой секции выберите атрибут Lock Pages in Memory.
5. Выберите из меню Action команду Select Security, чтобы открыть диалоговое окно Lock Pages in Memory. Щелкните кнопку Add. В диалоговом окне Select Users or Groups добавьте пользователей и/или группы, которым Вы хотите разрешить блокировку страниц в памяти. Затем закройте все диалоговые окна, щелкнув в каждом из них кнопку OK.

Новые права вступят в силу при следующей регистрации в системе. Если Вы только что сами себе предоставили право «Lock Pages in Memory», выйдите из системы и вновь зарегистрируйтесь в ней.

Ее первый параметр, *hProcess*, идентифицирует процесс, владеющий данными страницами памяти, а последние два параметра сообщают, сколько страниц оперативной памяти следует освободить и что это за страницы. Если освобождаемый блок в данный момент связан с адресным окном, он сначала отключается от этого окна.

И, наконец, завершая очистку, я освобождаю адресное окно. Для этого я вызываю *VirtualFree* и передаю ей базовый виртуальный адрес окна, нуль вместо размера региона и флаг *MEM_RELEASE*.

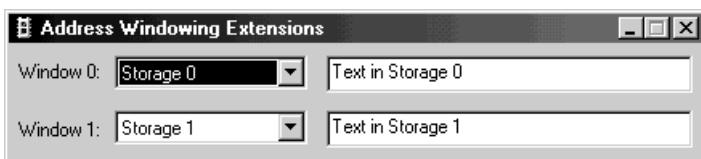
В моем простом примере создается одно адресное окно и единственный блок памяти. Это позволяет моей программе обращаться к оперативной памяти, которая никогда не будет сбрасываться на диск. Однако приложение может создать несколько адресных окон и выделить несколько блоков памяти. Эти блоки разрешается связывать с любым адресным окном, но операционная система не позволит связать один блок сразу с двумя окнами.

64-разрядная Windows 2000 полностью поддерживает AWE, так что перенос 32-разрядных приложений, использующих этот механизм, не вызывает никаких проблем. Однако AWE не столь полезен для 64-разрядных приложений, поскольку размеры их адресных пространств намного больше. Но все равно он дает возможность приложению выделять физическую память, которая никогда не сбрасывается на диск.

Программа-пример AWE

Эта программа, «15 AWE.exe» (см. листинг на рис. 15-3), демонстрирует, как создавать несколько адресных окон и связывать с ними разные блоки памяти. Файлы исходного кода и ресурсов этой программы находятся в каталоге 15-AWE на компакт-диске, прилагаемом к книге. Сразу после запуска программы AWE создается два адресных окна и выделяется два блока памяти.

Изначально первый блок занимает строка «Text in Storage 0», второй — строка «Text in Storage 1». Далее первый блок связывается с первым адресным окном, а второй — со вторым окном. При этом окно программы выглядит так, как показано ниже.



Оно позволяет немного поэкспериментировать. Во-первых, эти блоки можно назначить разным адресным окнам, используя соответствующие поля со списками. В них, кстати, предлагается и вариант No Storage, при выборе которого память отключается от адресного окна. Во-вторых, любое изменение текста немедленно отражается на блоке памяти, связанном с текущим адресным окном.

Если Вы попытаетесь связать один и тот же блок памяти с двумя адресными окнами одновременно, то, поскольку механизм AWE это не разрешает, на экране появится следующее сообщение.



Исходный код этой программы-примера предельно ясен. Чтобы облегчить работу с механизмом AWE, я создал три C++-класса, которые содержатся в файле AddrWindows.h. Первый класс, CSystemInfo, — очень простая оболочка функции *GetSystemInfo*. По одному его экземпляру создают остальные два класса.

Второй C++-класс, CAddrWindow, инкапсулирует адресное окно. Его метод *Create* резервирует адресное окно, метод *Destroy* уничтожает это окно, метод *UnmapStorage* отключает от окна связанный с ним блок памяти, а метод оператора приведения PVOID просто возвращает виртуальный адрес адресного окна.

Третий C++-класс, CAddrWindowStorage, инкапсулирует блок памяти, который можно назначить объекту класса CAddrWindow. Метод *Allocate* разрешает блокировать страницы в памяти, выделяет блок памяти, а затем отменяет право на блокировку. Метод *Free* освобождает блок памяти. Метод *HowManyPagesAllocated* возвращает количество фактически выделенных страниц. Наконец, метод *MapStorage* связывает блок памяти с объектом класса CAddrWindow, а *UnmapStorage* отключает блок от этого объекта.

Применение C++-классов существенно упростило реализацию программы AWE. Она создает по два объекта классов CAddrWindow и CAddrWindowStorage. Остальной код просто вызывает нужные методы в нужное время.



AWE.cpp

```

/*****
Modуль: AWE.cpp
Автор: Copyright (c) 2000, Джейфри Рихтер (Jeffrey Richter)
*****/

#include "..\CmnHdr.h"      /* см. приложение A */
#include <Windowsx.h>
#include <tchar.h>
#include "AddrWindow.h"
#include "Resource.h"

///////////////////////////////



CAddrWindow g_aw[2];           // два адресных окна
CAddrWindowStorage g_aws[2];   // два блока памяти
const ULONG_PTR g_nChars = 1024; // буфер на 1024 символа

///////////////////////////////



BOOL Dlg_OnInitDialog(HWND hwnd, HWND hwndFocus, LPARAM lParam) {
    chSETDLGICONS(hwnd, IDI_AWE);

    // создаем два адресных окна
    chVERIFY(g_aw[0].Create(g_nChars * sizeof(TCHAR)));
    chVERIFY(g_aw[1].Create(g_nChars * sizeof(TCHAR)));

    // создаем два блока памяти
    if (!g_aws[0].Allocate(g_nChars * sizeof(TCHAR))) {
        chFAIL("Failed to allocate RAM.\nMost likely reason: "
               "you are not granted the Lock Pages in Memory user right.");
    }
    chVERIFY(g_aws[1].Allocate(g_nChars * sizeof(TCHAR)));

    // помещаем в первый блок текст по умолчанию
    g_aws[0].MapStorage(g_aw[0]);
    lstrcpy((PSTR) (VOID) g_aw[0], TEXT("Text in Storage 0"));

    // помещаем во второй блок текст по умолчанию
    g_aws[1].MapStorage(g_aw[0]);
    lstrcpy((PSTR) (VOID) g_aw[0], TEXT("Text in Storage 1"));

    // заполняем элементы управления диалогового окна
    for (int n = 0; n <= 1; n++) {
        // настраиваем поле со списком для каждого адресного окна
    }
}

```

Рис. 15-3. Программа-пример AWE

Рис. 15-3. продолжение

```

int id = ((n == 0) ? IDC_WINDOWOSTORAGE : IDC_WINDOW1STORAGE);
HWND hwndCB = GetDlgItem(hwnd, id);
ComboBox_AddString(hwndCB, TEXT("No storage"));
ComboBox_AddString(hwndCB, TEXT("Storage 0"));
ComboBox_AddString(hwndCB, TEXT("Storage 1"));

// окно 0 показывает Storage 0, а окно 1 – Storage 1
ComboBox_SetCurSel(hwndCB, n + 1);
FORWARD_WM_COMMAND(hwnd, id, hwndCB, CBN_SELCHANGE, SendMessage);
Edit_LimitText(GetDlgItem(hwnd,
    (n == 0) ? IDC_WINDOWOTEXT : IDC_WINDOW1TEXT), g_nChars);
}

return(TRUE);
}

///////////////////////////////
void Dlg_OnCommand(HWND hwnd, int id, HWND hwndCtl, UINT codeNotify) {
    switch (id) {

case IDCANCEL:
    EndDialog(hwnd, id);
    break;

case IDC_WINDOWOSTORAGE:
case IDC_WINDOW1STORAGE:
    if (codeNotify == CBN_SELCHANGE) {
        // показываем для каждого окна другой блок
        int nWindow = ((id == IDC_WINDOWOSTORAGE) ? 0 : 1);
        int nStorage = ComboBox_GetCurSel(hwndCtl) - 1;
        if (nStorage == -1) { // с этим окном блок памяти не связан
            chVERIFY(g_aw[nWindow].UnmapStorage());
        } else {
            if (!g_aws[nStorage].MapStorage(g_aw[nWindow])) {
                // не удалось связать блок с окном
                chVERIFY(g_aw[nWindow].UnmapStorage());
                ComboBox_SetCurSel(hwndCtl, 0); // устанавливаем "No storage"
                chMB("This storage can be mapped only once.");
            }
        }
    }

    // обновляем текст в поле, относящемся к адресному окну
    HWND hwndText = GetDlgItem(hwnd,
        ((nWindow == 0) ? IDC_WINDOWOTEXT : IDC_WINDOW1TEXT));
    MEMORY_BASIC_INFORMATION mbi;
    VirtualQuery(g_aw[nWindow], &mbi, sizeof(mbi));
    // Примечание: mbi.State == MEM_RESERVE, если с адресным окном
    // не связан блок памяти
    EnableWindow(hwndText, (mbi.State == MEM_COMMIT));
}

```

см. след. стр.

Рис. 15-3. продолжение

```

        Edit_SetText(hwndText, IsWindowEnabled(hwndText)
            ? (PCSTR) (PVOID) g_aw[nWindow] : TEXT("(No storage")));
    }
    break;
case IDC_WINDOW0TEXT:
case IDC_WINDOW1TEXT:
    if (codeNotify == EN_CHANGE) {
        // обновляем память, связанную с этим адресным окном
        int nWindow = ((id == IDC_WINDOW0TEXT) ? 0 : 1);
        Edit_GetText(hwndCtl, (PSTR) (PVOID) g_aw[nWindow], g_nChars);
    }
    break;
}
}

/////////////////////////////// Конец файла //////////////////////////////

INT_PTR WINAPI Dlg_Proc(HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam) {

switch (uMsg) {
    chHANDLE_DLGMMSG(hwnd, WM_INITDIALOG, Dlg_OnInitDialog);
    chHANDLE_DLGMMSG(hwnd, WM_COMMAND, Dlg_OnCommand);
}

return(FALSE);
}

/////////////////////////////// Конец файла //////////////////////////////

int WINAPI _tWinMain(HINSTANCE hinstExe, HINSTANCE, PTSTR pszCmdLine, int) {

chWindows2000Required();

DialogBox(hinstExe, MAKEINTRESOURCE(IDD_AWE), NULL, Dlg_Proc);
return(0);
}

/////////////////////////////// Конец файла //////////////////////////////

```

AddrWindow.h

```

/*********************************************
Модуль: AddrWindow.h
Автор: Copyright (c) 2000, Джейфри Рихтер (Jeffrey Richter)
*****/
#pragma once

/////////////////////////////// Конец файла //////////////////////////////

```

Рис. 15-3. продолжение

```

#include "..\CmnHdr.h"      /* см. приложение A */
#include <tchar.h>

///////////////////////////////



class CSystemInfo : public SYSTEM_INFO {
public:
    CSystemInfo() { GetSystemInfo(this); }
};

///////////////////////////////



class CAddrWindow {
public:
    CAddrWindow() { m_pvWindow = NULL; }
    ~CAddrWindow() { Destroy(); }

    BOOL Create(SIZE_T dwBytes, PVOID pvPreferredWindowBase = NULL) {
        // резервируем регион для адресного окна
        m_pvWindow = VirtualAlloc(pvPreferredWindowBase, dwBytes,
            MEM_RESERVE | MEM_PHYSICAL, PAGE_READWRITE);
        return(m_pvWindow != NULL);
    }

    BOOL Destroy() {
        BOOL fOk = TRUE;
        if (m_pvWindow != NULL) {
            // удаляем регион, выделенный для адресного окна
            fOk = VirtualFree(m_pvWindow, 0, MEM_RELEASE);
            m_pvWindow = NULL;
        }
        return(fOk);
    }

    BOOL UnmapStorage() {
        // отключаем всю память от адресного окна
        MEMORY_BASIC_INFORMATION mbi;
        VirtualQuery(m_pvWindow, &mbi, sizeof(mbi));
        return(MapUserPhysicalPages(m_pvWindow,
            mbi.RegionSize / sm_sinf.dwPageSize, NULL));
    }

    // возвращаем виртуальный адрес адресного окна
    operator PVOID() { return(m_pvWindow); }

private:
    PVOID m_pvWindow; // виртуальный адрес адресного окна
    static CSystemInfo sm_sinf;
};

///////////////////////////////

```

см. след. стр.

Рис. 15-3. продолжение

```

CSysInfo CAddrWindow::sm_sinf;

//////////////////////////////



class CAddrWindowStorage {
public:
    CAddrWindowStorage() { m_ulPages = 0; m_pulUserPfnArray = NULL; }
    ~CAddrWindowStorage() { Free(); }

    BOOL Allocate(ULONG_PTR ulBytes) {
        // выделяем память, предназначенную для адресного окна

        Free(); // очищаем существующее адресное окно,
        // принадлежащее этому объекту

        // вычисляем количество страниц по числу байтов
        m_ulPages = (ulBytes + sm_sinf.dwPageSize) / sm_sinf.dwPageSize;

        // создаем массив для номеров фреймов страниц
        m_pulUserPfnArray = (PULONG_PTR)
            HeapAlloc(GetProcessHeap(), 0, m_ulPages * sizeof(ULONG_PTR));

        BOOL f0k = (m_pulUserPfnArray != NULL);
        if (f0k) {
            // должно быть предоставлено право "Lock Pages in Memory"
            EnablePrivilege(SE_LOCK_MEMORY_NAME, TRUE);
            f0k = AllocateUserPhysicalPages(GetCurrentProcess(),
                &m_ulPages, m_pulUserPfnArray);
            EnablePrivilege(SE_LOCK_MEMORY_NAME, FALSE);
        }
        return(f0k);
    }

    BOOL Free() {
        BOOL f0k = TRUE;
        if (m_pulUserPfnArray != NULL) {
            f0k = FreeUserPhysicalPages(GetCurrentProcess(),
                &m_ulPages, m_pulUserPfnArray);
            if (f0k) {
                // освобождаем массив для номеров фреймов страниц
                HeapFree(GetProcessHeap(), 0, m_pulUserPfnArray);
                m_ulPages = 0;
                m_pulUserPfnArray = NULL;
            }
        }
        return(f0k);
    }

    ULONG_PTR HowManyPagesAllocated() { return(m_ulPages); }
}

```

Рис. 15-3. продолжение

```

BOOL MapStorage(CAddrWindow& aw) {
    return(MapUserPhysicalPages(aw,
        HowManyPagesAllocated(), m_pulUserPfnArray));
}

BOOL UnmapStorage(CAddrWindow& aw) {
    return(MapUserPhysicalPages(aw,
        HowManyPagesAllocated(), NULL));
}

private:
    static BOOL EnablePrivilege(PCTSTR pszPrivName, BOOL fEnable = TRUE) {

        BOOL fOk = FALSE; // считаем, что функция потерпит неудачу
        HANDLE hToken;

        // пытаемся открыть маркер доступа у этого процесса
        if (OpenProcessToken(GetCurrentProcess(),
            TOKEN_ADJUST_PRIVILEGES, &hToken)) {

            // пытаемся разрешить блокировать страницы в памяти
            TOKEN_PRIVILEGES tp = { 1 };
            LookupPrivilegeValue(NULL, pszPrivName, &tp.Privileges[0].Luid);
            tp.Privileges[0].Attributes = fEnable ? SE_PRIVILEGE_ENABLED : 0;
            AdjustTokenPrivileges(hToken, FALSE, &tp, sizeof(tp), NULL, NULL);
            fOk = (GetLastError() == ERROR_SUCCESS);
            CloseHandle(hToken);
        }
        return(fOk);
    }

private:
    ULONG_PTR m_ulPages;           // число страниц памяти
    PULONG_PTR m_pulUserPfnArray;  // массив для номеров фреймов страниц

private:
    static CSysInfo sm_sinf;
};

/////////////////////////////// Конец файла ///////////////////////////////
CSysInfo CAddrWindowStorage::sm_sinf;
/////////////////////////////// Конец файла ///////////////////////////////

```

Стек потока

Иногда система сама резервирует какие-то регионы в адресном пространстве Вашего процесса. Я уже упоминал в главе 13, что это делается для размещения блоков переменных окружения процесса и его потоков. Еще один случай резервирования региона самой системой — создание стека потока.

Всякий раз, когда в процессе создается поток, система резервирует регион адресного пространства для стека потока (у каждого потока свой стек) и передает этому региону какой-то объем физической памяти. По умолчанию система резервирует 1 Мб адресного пространства и передает ему всего две страницы памяти. Но стандартные значения можно изменить, указав при сборке программы параметр компоновщика /STACK:

```
/STACK:reserve[, commit]
```

Тогда при создании стека потока система зарезервирует регион адресного пространства, размер которого указан в параметре /STACK компоновщика. Кроме того, объем изначально передаваемой памяти можно переопределить вызовом *CreateThread* или *_beginthreadex*. У обеих функций есть параметр, который позволяет изменять объем памяти, изначально передаваемой региону стека. Если в нем передать 0, система будет использовать значение, указанное в параметре /STACK. Далее я исхожу из того, что стек создается со стандартными параметрами.

На рис. 16-1 показано, как может выглядеть регион стека (зарезервированный по адресу 0x08000000) в системе с размером страниц по 4 Кб. Регион стека и вся переданная ему память имеют атрибут защиты PAGE_READWRITE.

Зарезервировав регион, система передает физическую память двум верхним его страницам. Непосредственно перед тем, как приступить к выполнению потока, система устанавливает регистр указателя стека на конец верхней страницы региона стека (адрес, очень близкий к 0x08100000). Это та страница, с которой поток начнет использовать свой стек. Вторая страница сверху называется *сторожевой* (guard page).

По мере разрастания дерева вызовов (одновременного обращения ко все большему числу функций) потоку, естественно, требуется и больший объем стека. Как только поток обращается к следующей странице (а она сторожевая), система уведомляется об этой попытке. Тогда система передает память еще одной странице, расположенной как раз за сторожевой. После чего флаг PAGE_GUARD, как эстафетная палочка, переходит от текущей сторожевой к той странице, которой только что передана память. Благодаря такому механизму объем памяти, занимаемой стеком, увеличивается только по необходимости. Если дерево вызовов у потока будет расти и дальше, регион стека будет выглядеть примерно так, как показано на рис. 16-2.

Допустим, стек потока практически заполнен (как на рис. 16-2) и регистр указателя стека указывает на адрес 0x08003004. Тогда, как только поток вызовет еще одну функцию, система, по идеи, должна передать дополнительную физическую память. Но

когда система передает память странице по адресу 0x08001000, она делает это уже по-другому. Регион стека теперь выглядит, как на рис. 16-3.

Адрес	Состояние страницы
0x080FF000	Верхняя часть стека (переданная страница)
0x080FE000	Переданная страница с флагом PAGE_GUARD
0x080FD000	Зарезервированная страница
0x08003000	Зарезервированная страница
0x08002000	Зарезервированная страница
0x08001000	Зарезервированная страница
0x08000000	Нижняя часть стека (зарезервированная страница)

Рис. 16-1. Так выглядит регион стека потока сразу после его создания

Адрес	Состояние страницы
0x080FF000	Верхняя часть стека (переданная страница)
0x080FE000	Переданная страница
0x080FD000	Переданная страница
0x08003000	Переданная страница
0x08002000	Переданная страница с флагом PAGE_GUARD
0x08001000	Зарезервированная страница
0x08000000	Нижняя часть стека (зарезервированная страница)

Рис. 16-2. Почти заполненный регион стека потока

Адрес	Состояние страницы
0x080FF000	Верхняя часть стека (переданная страница)
0x080FE000	Переданная страница
0x080FD000	Переданная страница
0x08003000	Переданная страница
0x08002000	Переданная страница
0x08001000	Переданная страница
0x08000000	Нижняя часть стека (зарезервированная страница)

Рис. 16-3. Целиком заполненный регион стека потока

Как и можно было предполагать, флаг PAGE_GUARD со страницы по адресу 0x08002000 удаляется, а странице по адресу 0x08001000 передается физическая память. Но этой странице не присваивается флаг PAGE_GUARD. Это значит, что региону адресного пространства, зарезервированному под стек потока, теперь передана вся физическая память, которая могла быть ему передана. Самая нижняя страница остается зарезервированной, физическая память ей никогда не передается. Чуть позже я поясню, зачем это сделано.

Передавая физическую память странице по адресу 0x08001000, система выполняет еще одну операцию: генерирует исключение EXCEPTION_STACK_OVERFLOW (в файле WinNT.h оно определено как 0xC00000FD). При использовании структурной обработки исключений (SEH) Ваша программа получит уведомление об этой ситуации и сможет корректно обработать ее. Подробнее о SEH см. главы 23, 24 и 25, а также листинг программы Summation, приведенный в конце этой главы.

Если поток продолжит использовать стек даже после исключения, связанного с переполнением стека, будет задействована вся память на странице по адресу 0x08001000, и поток попытается получить доступ к странице по адресу 0x08000000. Поскольку эта страница лишь зарезервирована (но не передана), возникнет исключение — нарушение доступа. Если это произойдет в момент обращения потока к стеку, Вас ждут крупные неприятности. Система возьмет управление на себя и завершит не только данный поток, но и весь процесс. И даже не сообщит об этом пользователю; процесс просто исчезнет!

Теперь объясню, почему нижняя страница стека всегда остается зарезервированной. Это позволяет защищать другие данные процесса от случайной перезаписи. Видите ли, по адресу 0x07FFF000 (на 1 страницу ниже, чем 0x08000000) может быть передана физическая память для другого региона адресного пространства. Если бы странице по адресу 0x08000000 была передана физическая память, система не суме-

ла бы перехватить попытку потока расширить стек за пределы зарезервированного региона. А если бы стек расползся за пределы этого региона, поток мог бы перезаписать другие данные в адресном пространстве своего процесса — такого «жучка» выловить очень сложно.

Стек потока в Windows 98

В Windows 98 стеки ведут себя почти так же, как и в Windows 2000. Но отличия все же есть.

На рис. 16-4 показано, как в Windows 98 может выглядеть регион стека (зарезервированный с адреса 0x00530000) размером 1 Мб.

Адрес	Размер	Состояние страницы
0x00640000	16 страниц (65 536 байтов)	Верхняя часть стека (зарезервирована для перехвата обращений к несуществующей области стека)
0x0063F000	1 страница (4096 байтов)	Переданная страница с атрибутом PAGE_READWRITE (задействованная область стека)
0x0063E000	1 страница (4096 байтов)	Страница с атрибутом PAGE_NOACCESS (заменяет флаг PAGE_GUARD)
0x00638000	6 страниц (24 576 байтов)	Страницы, зарезервированные для перехвата переполнения стека
0x00637000	1 страница (4096 байтов)	Переданная страница с атрибутом PAGE_READWRITE (для совместимости с 16-разрядными компонентами)
0x00540000	247 страниц (1 011 712 байтов)	Страницы, зарезервированные для дальнейшего расширения стека
0x00530000	16 страниц (65 536 байтов)	Нижняя часть стека (зарезервирована для перехвата переполнения стека)

Рис. 16-4. Так выглядит регион стека сразу после его создания под управлением Windows 98

Во-первых, размер региона на самом деле 1 Мб плюс 128 Кб, хотя мы хотели создать стек объемом всего 1 Мб. В Windows 98 система резервирует под стек на 128 Кб больше, чем было запрошено. Собственно стек располагается в середине этого региона, а по обеим его границам размещаются блоки по 64 Кб каждый.

Блок перед стеком предназначен для перехвата его переполнения, а блок после стека — для перехвата обращений к несуществующим областям стека. Чтобы понять, какая польза от последнего блока, рассмотрим такой фрагмент кода:

```
int WINAPI WinMain(HINSTANCE hinstExe, HINSTANCE,
    PSTR pszCmdLine, int nCmdShow) {

    char szBuf[100];
    szBuf[10000] = 0; // обращение к несуществующей области стека

    return(0);
}
```

Когда выполняется оператор присвоения, происходит попытка обращения за конец стека потока. Разумеется, ни компилятор, ни компоновщик не уловят эту ошибку в приведенном фрагменте кода, но, если приложение работает под управлением Win-

dows 98, выполнение этого оператора вызовет нарушение доступа. Это одна из приятных особенностей Windows 98, отсутствующих в Windows 2000, в которой сразу за стеком потока может быть расположен другой регион. И если Вы случайно обратитесь за пределы стека, Вы можете испортить содержимое области памяти, принадлежащей другой части Вашего процесса, — система ничего не заметит.

Второе отличие: в стеке нет страниц с флагом атрибутов защиты PAGE_GUARD. Поскольку Windows 98 такой флаг не поддерживает, при расширении стека потока она действует несколько иначе. Она помечает страницу переданной памяти, располагаемой под стеком, атрибутом PAGE_NOACCESS (на рис. 16-4 — по адресу 0x0063E000). Когда поток обращается к этой странице, происходит нарушение доступа. Система перехватывает это исключение, меняет атрибут защиты страницы с PAGE_NOACCESS на PAGE_READWRITE и передает память новой «сторожевой» странице, размещаемой сразу за предыдущей.

Третье: обратите внимание на единственную страницу с атрибутом PAGE_READWRITE по адресу 0x00637000. Она создается для совместимости с 16-разрядной Windows. Хотя Microsoft нигде не говорит об этом, разработчики обнаружили, что первые 16 байтов сегмента стека 16-разрядной программы содержат информацию о ее стеке, локальной куче и локальной таблице атомарного доступа. Поскольку Win32-приложения в Windows 98 часто обращаются к 16-разрядным DLL и некоторые из этих DLL предполагают наличие тех самых 16 байтов в начале сегмента стека, Microsoft пришлось эмулировать подобные данные и в Windows 98. Когда 32-разрядный код обращается к 16-разрядному, Windows 98 отображает 16-битный селектор процессора на 32-разрядный стек и записывает в регистр сегмента стека (SS) такое значение, чтобы он указывал на страницу по адресу 0x00637000. И тогда 16-разрядный код, получив доступ к своим 16 байтам в начале сегмента стека, продолжает выполнение без всяких проблем.

По мере роста стека потока, выполняемого под управлением Windows 98, блок памяти по адресу 0x0063F000 постепенно увеличивается, а сторожевая страница смешается вниз до тех пор, пока не будет достигнут предел в 1 Мб, после чего она исчезает так же, как и в Windows 2000. Одновременно система смешает позицию страницы, предназначенной для совместимости с компонентами 16-разрядной Windows, и она, в конце концов, попадает в 64-килобайтовый блок, расположенный в начале региона стека. Поэтому целиком заполненный стек в Windows 98 выглядит так, как показано на рис. 16-5.

Адрес	Размер	Состояние страницы
0x00640000	16 страниц (65 536 байтов)	Верхняя часть стека (зарезервирована для перехвата обращений к несуществующей области стека)
0x00540000	256 страниц (1 Мб)	Переданная страница с атрибутом PAGE_READWRITE (задействованная область стека)
0x00539000	7 страниц (28 672 байта)	Страницы, зарезервированные для перехвата переполнения стека
0x00538000	1 страница (4096 байтов)	Переданная страница с атрибутом PAGE_READWRITE (для совместимости с 16-разрядными компонентами)
0x00530000	8 страниц (32 768 байтов)	Нижняя часть стека (зарезервирована для перехвата переполнения стека)

Рис. 16-5. Целиком заполненный регион стека потока в Windows 98

Функция из библиотеки C/C++ для контроля стека

Библиотека C/C++ содержит функцию, позволяющую контролировать стек. Транслируя исходный код программы, компилятор при необходимости генерирует вызовы этой функции. Она обеспечивает корректную передачу страниц физической памяти стеку потока.

Возьмем, к примеру, небольшую функцию, требующую массу памяти под свои локальные переменные:

```
void SomeFunction() {
    int nValues[4000];

    // здесь что-то делаем с массивом

    nValues[0] = 0; // а тут что-то присваиваем
}
```

Для размещения целочисленного массива функция потребует минимум 16 000 байтов стекового пространства, так как каждое целое значение занимает 4 байта. Код, генерируемый компилятором, обычно выделяет такое пространство в стеке простым уменьшением указателя стека процессора на 16 000 байтов. Однако система не передаст физическую память этой нижней области стека, пока не произойдет обращения по данному адресу.

В системе с размером страниц по 4 или 8 Кб это могло бы создать проблему. Если первое обращение к стеку проходит по адресу, расположенному ниже сторожевой страницы (как в показанном выше фрагменте кода), поток обратится к зарезервированной памяти, и возникнет нарушение доступа. Поэтому, чтобы можно было спокойно писать функции вроде приведенной выше, компилятор и вставляет в код вызовы библиотечной функции для контроля стека.

При трансляции программы компилятору известен размер страниц памяти, используемых целевым процессором (4 Кб для x86 и 8 Кб для Alpha). Встречая в программе ту или иную функцию, компилятор определяет требуемый для нее объем стека и, если он превышает размер одной страницы, вставляет вызов функции, контролирующей стек.

Ниже показан псевдокод, который иллюстрирует, что именно делает функция, контролирующая стек. (Я говорю «псевдокод» потому, что обычно эта функция реализуется поставщиками компиляторов на языке ассемблера.)

```
// стандартной библиотеке С "известен" размер страницы в целевой системе
#ifndef _M_ALPHA
#define PAGESIZE (8 * 1024) // страницы по 8 Кб
#else
#define PAGESIZE (4 * 1024) // страницы по 4 Кб
#endif

void StackCheck(int nBytesNeededFromStack) {
    // Получим значение указателя стека. В этом месте указатель стека
    // еще НЕ был уменьшен для учета локальных переменных функции.
    PBYTE pbStackPtr = (указатель стека процессора);

    while (nBytesNeededFromStack >= PAGESIZE) {
        // смещаем страницу вниз по стеку - должна быть сторожевой
        pbStackPtr -= PAGESIZE;
```

см. след. стр.

```
// обращаемся к какому-нибудь байту на сторожевой странице, вызывая
// тем самым передачу новой страницы и сдвиг сторожевой страницы вниз
pbStackPtr[0] = 0;

// уменьшаем требуемое количество байтов в стеке
nBytesNeededFromStack -= PAGESIZE;
}

// перед возвратом управления функция StackCheck устанавливает регистр
// указателя стека на адрес, следующий за локальными переменными функции
}
```

В компиляторе Microsoft Visual C++ предусмотрен параметр, позволяющий контролировать пороговый предел числа страниц, начиная с которого компилятор автоматически вставляет в программу вызов функции *StackCheck*. Используйте этот параметр, только если Вы точно знаете, что делаете, и если это действительно нужно. В 99,99999 процентах из ста приложения и DLL не требуют применения упомянутого параметра.

Программа-пример Summation

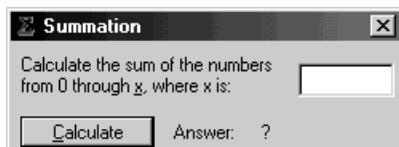
Эта программа, «16 Summation.exe» (см. листинг на рис. 16-6), демонстрирует использование фильтров и обработчиков исключений для корректного восстановления после переполнения стека. Файлы исходного кода и ресурсов этой программы находятся в каталоге 16-Summation на компакт-диске, прилагаемом к книге. Возможно, Вам придется сначала прочесть главы по SEH, чтобы понять, как работает эта программа.

Она суммирует числа от 0 до x , где x — число, введенное пользователем. Конечно, проще было бы написать функцию с именем *Sum*, которая вычисляла бы по формуле:

$$\text{Sum} = (x * (x + 1)) / 2;$$

Но для этого примера я сделал функцию *Sum* рекурсивной, чтобы она использовала большое стековое пространство.

При запуске программы появляется диалоговое окно, показанное ниже.



В этом окне Вы вводите число и щелкаете кнопку *Calculate*. Программа создает поток, единственная обязанность которого — сложить все числа от 0 до x . Пока он выполняется, первичный поток программы, вызвав *WaitForSingleObject*, просит систему не выделять ему процессорное время. Когда новый поток завершается, система вновь выделяет процессорное время первичному потоку. Тот выясняет сумму, получая код завершения нового потока вызовом *GetExitCodeThread*, и — это очень важно — закрывает свой описатель нового потока, так что система может уничтожить объект ядра «поток», и утечки ресурсов не произойдет.

Далее первичный поток проверяет код завершения суммирующего потока. Если он равен *UINT_MAX*, значит, произошла ошибка: суммирующий поток переполнил стек при подсчете суммы; тогда первичный поток выведет окно с соответствующим сообщением. Если же код завершения отличен от *UINT_MAX*, суммирующий поток

отработал успешно; код завершения и есть искомая сумма. В этом случае первичный поток просто отображает результат суммирования в диалоговом окне.

Теперь обратимся к суммирующему потоку. Его функция — *SumThreadFunc*. При создании этого потока первичный поток передает ему в единственном параметре *pvParam* количество целых чисел, которые следует просуммировать. Затем его функция инициализирует переменную *iSum* значением *UINT_MAX*, т. е. изначально предполагается, что работа функции не завершится успехом. Далее *SumThreadFunc* активизирует SEH так, чтобы перехватывать любое исключение, возникающее при выполнении потока. После чего для вычисления суммы вызывается рекурсивная функция *Sum*.

Если сумма успешно вычислена, *SumThreadFunc* просто возвращает значение переменной *iSum*; оно и будет кодом завершения потока. Но, если при выполнении *Sum* возникает исключение, система сразу оценивает выражение в фильтре исключений. Иначе говоря, система вызывает *FilterFunc*, передавая ей код исключения. В случае переполнения стека этим кодом будет *EXCEPTION_STACK_OVERFLOW*. Чтобы увидеть, как программа обрабатывает исключение, вызванное переполнением стека, дайте ей просуммировать числа от 0 до 44000.

Моя функция *FilterFunc* очень проста. Сначала она проверяет, произошло ли исключение, связанное с переполнением стека. Если нет, возвращает *EXCEPTION_CONTINUE_SEARCH*, а если да — *EXCEPTION_EXECUTE_HANDLER*. Это подсказывает системе, что фильтр готов к обработке этого исключения и что надо выполнить код в блоке *except*. В данном случае обработчик исключения ничего особенного не делает, просто закрывая поток с кодом завершения *UINT_MAX*. Родительский поток, получив это специальное значение, выводит пользователю сообщение с предупреждением.

И последнее, что хотелось бы обсудить: почему я выделил функцию *Sum* в отдельный поток вместо того, чтобы просто создать SEH-фрейм в первичном потоке и вызывать *Sum* из его блока *try*. На то есть три причины.

Во-первых, всякий раз, когда создается поток, он получает стек размером 1 Мб. Если бы я вызывал *Sum* из первичного потока, часть стекового пространства уже была бы занята, и функция не смогла бы использовать весь объем стека. Согласен, моя программа очень проста и, может быть, не займет слишком большое стековое пространство. А если программа посложнее? Легко представить ситуацию, когда *Sum* подсчитывает сумму целых чисел от 0 до 1000 и стек вдруг оказывается чем-то занят, — тогда его переполнение произойдет, скажем, еще при вычислении суммы от 0 до 750. Таким образом, работа функции *Sum* будет надежнее, если предоставить ей полный стек, не используемый другим кодом.

Вторая причина в том, что поток уведомляется об исключении «переполнение стека» лишь однажды. Если бы я вызывал *Sum* из первичного потока и произошло бы переполнение стека, то это исключение было бы перехвачено и корректно обработано. Но к тому моменту физическая память была бы передана под все зарезервированное адресное пространство стека, и в нем уже не осталось бы страниц с флагом защиты. Начни пользователь новое суммирование, и функция *Sum* переполнила бы стек, а соответствующее исключение не было бы возбуждено. Вместо этого возникло бы исключение «нарушение доступа», и корректно обработать эту ситуацию уже не удалось бы.

И последнее, почему я использую отдельный поток: физическую память, отведенную под его стек, можно освободить. Рассмотрим такой сценарий: пользователь просит функцию *Sum* вычислить сумму целых чисел от 0 до 30 000. Это требует передачи региону стека весьма ощутимого объема памяти. Затем пользователь проводит не-

сколько операций суммирования — максимум до 5000. И окажется, что стеку передан порядочный объем памяти, который больше не используется. А ведь эта физическая память выделяется из страничного файла. Так что лучше бы освободить ее и вернуть системе. И поскольку программа завершает поток *SumThreadFunc*, система автоматически освобождает физическую память, переданную региону стека.



Summation.cpp

```
*****  
Модуль: Summation.cpp  
Автор: Copyright (c) 2000, Джейфри Рихтер (Jeffrey Richter)  
*****  
  
#include "..\CmnHdr.h"      /* см. приложение A */  
#include <windowsx.h>  
#include <limits.h>  
#include <process.h>        // для доступа к _beginthreadex  
#include <tchar.h>  
#include "Resource.h"  
  
//////////  
  
// Пример вызова Sum для uNum от 0 до 9  
// uNum: 0 1 2 3 4 5 6 7 8 9 ...  
// Sum: 0 1 3 6 10 15 21 28 36 45 ...  
UINT Sum(UINT uNum) {  
  
    // рекурсивный вызов Sum  
    return((uNum == 0) ? 0 : (uNum + Sum(uNum - 1)));  
}  
  
//////////  
  
LONG WINAPI FilterFunc(DWORD dwExceptionCode) {  
  
    return((dwExceptionCode == STATUS_STACK_OVERFLOW)  
        ? EXCEPTION_EXECUTE_HANDLER : EXCEPTION_CONTINUE_SEARCH);  
}  
  
//////////  
// Отдельный поток, отвечающий за вычисление суммы.  
// Я использую его по следующим причинам:  
// 1. Отдельный поток получает собственный мегабайт стекового пространства.  
// 2. Поток уведомляется о переполнении стека лишь однажды.  
// 3. Память, выделенная для стека, освобождается по завершении потока.  
DWORD WINAPI SumThreadFunc(PVOID pvParam) {  
  
    // параметр pvParam определяет количество суммируемых чисел  
    UINT uSumNum = PtrToUlong(pvParam);
```

Рис. 16-6. Программа-пример *Summation*

Рис. 16-6. продолжение

```

// uSum содержит сумму чисел от 0 до uSumNum; если сумму вычислить
// не удалось, возвращается значение UINT_MAX
UINT uSum = UINT_MAX;

__try {
    // для перехвата исключения "переполнение стека"
    // функцию Sum надо выполнять в SEH-фрейме
    uSum = Sum(uSumNum);
}
__except (FilterFunc(GetExceptionCode())) {
    // Если мы попали сюда, то это потому, что перехватили переполнение
    // стека. Здесь можно сделать все, что надо для корректного
    // возобновления работы. Но, так как от этого примера больше ничего
    // не требуется, кода в блоке обработчика нет.
}

// кодом завершения потока является либо сумма первых uSumNum
// чисел, либо UINT_MAX в случае переполнения стека
return(uSum);
}

///////////
BOOL Dlg_OnInitDialog(HWND hwnd, HWND hwndFocus, LPARAM lParam) {
    chSETDLGICONS(hwnd, IDI_SUMMATION);

    // мы принимаем не более чем девятизначные целые числа
    Edit_LimitText(GetDlgItem(hwnd, IDC_SUMNUM), 9);

    return(TRUE);
}

///////////

void Dlg_OnCommand(HWND hwnd, int id, HWND hwndCtl, UINT codeNotify) {
    switch (id) {
        case IDCANCEL:
            EndDialog(hwnd, id);
            break;

        case IDC_CALC:
            // получаем количество целых чисел, которые
            // пользователь хочет просуммировать
            UINT uSum = GetDlgItemInt(hwnd, IDC_SUMNUM, NULL, FALSE);

            // создаем поток (с собственным стеком), отвечающий за суммирование
            DWORD dwThreadId;
            HANDLE hThread = chBEGINTHREADEX(NULL, 0,
                SumThreadFunc, (PVOID) (UINT_PTR) uSum, 0, &dwThreadId);
    }
}

```

см. след. стр.

Рис. 16-6. продолжение

```
// ждем завершения потока
WaitForSingleObject(hThread, INFINITE);

// код завершения – результат суммирования
GetExitCodeThread(hThread, (PDWORD) &uSum);

// закончив, закрываем описатель потока,
// чтобы система могла разрушить объект ядра "поток"
CloseHandle(hThread);

// обновляем содержимое диалогового окна
if (uSum == UINT_MAX) {
    // если код завершения равен UINT_MAX,
    // произошло переполнение стека
    SetDlgItemText(hwnd, IDC_ANSWER, TEXT("Error"));
    chMB("The number is too big, please enter a smaller number");
} else {
    // сумма вычислена успешно
    SetDlgItemInt(hwnd, IDC_ANSWER, uSum, FALSE);
}
break;
}

/////////////////////////////// INT_PTR WINAPI Dlg_Proc(HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam) {

switch (uMsg) {
    chHANDLE_DLGMMSG(hwnd, WM_INITDIALOG, Dlg_OnInitDialog);
    chHANDLE_DLGMMSG(hwnd, WM_COMMAND, Dlg_OnCommand);
}
return(FALSE);
}

/////////////////////////////// int WINAPI _tWinMain(HINSTANCE hinstExe, HINSTANCE, PTSTR pszCmdLine, int) {

DialogBox(hinstExe, MAKEINTRESOURCE(IDD_SUMMATION), NULL, Dlg_Proc);
return(0);
}

/////////////////////////////// Конец файла ///////////////////////////////
```

Проектируемые в память файлы

Операции с файлами — это то, что рано или поздно приходится делать практически во всех программах, и всегда это вызывает массу проблем. Должно ли приложение просто открыть файл, считать и закрыть его, или открыть, считать фрагмент в буфер и перезаписать его в другую часть файла? В Windows многие из этих проблем решаются очень изящно — с помощью проектируемых в память файлов (*memory-mapped files*).

Как и виртуальная память, проектируемые файлы позволяют резервировать регион адресного пространства и передавать ему физическую память. Различие между этими механизмами состоит в том, что в последнем случае физическая память не выделяется из страничного файла, а берется из файла, уже находящегося на диске. Как только файл спроектирован в память, к нему можно обращаться так, будто он целиком в нее загружен.

Проектируемые файлы применяются для:

- загрузки и выполнения EXE- и DLL-файлов. Это позволяет существенно экономить как на размере страничного файла, так и на времени, необходимом для подготовки приложения к выполнению;
- доступа к файлу данных, размещенному на диске. Это позволяет обойтись без операций файлового ввода-вывода и буферизации его содержимого;
- разделения данных между несколькими процессами, выполняемыми на одной машине. (В Windows есть и другие методы для совместного доступа разных процессов к одним данным — но все они так или иначе реализованы на основе проектируемых в память файлов.)

Эти области применения проектируемых файлов мы и рассмотрим в данной главе.

Проектирование в память EXE- и DLL-файлов

При вызове из потока функции *CreateProcess* система действует так:

1. Отыскивает EXE-файл, указанный при вызове *CreateProcess*. Если файл не найден, новый процесс не создается, а функция возвращает FALSE.
2. Создает новый объект ядра «процесс».
3. Создает адресное пространство нового процесса.
4. Резервирует регион адресного пространства — такой, чтобы в него поместился данный EXE-файл. Желательное расположение этого региона указывается внутри самого EXE-файла. По умолчанию базовый адрес EXE-файла — 0x00400000 (в 64-разрядном приложении под управлением 64-разрядной Windows 2000 этот адрес может быть другим). При создании исполняемого файла приложения базовый адрес может быть изменен через параметр компоновщика /BASE.

5. Отмечает, что физическая память, связанная с зарезервированным регионом, — EXE-файл на диске, а не страничный файл.

Спроектировав EXE-файл на адресное пространство процесса, система обращается к разделу EXE-файла со списком DLL, содержащих необходимые программе функции. После этого система, вызывая *LoadLibrary*, поочередно загружает указанные (а при необходимости и дополнительные) DLL-модули. Всякий раз, когда для загрузки DLL вызывается *LoadLibrary*, система выполняет действия, аналогичные описанным выше в пп. 4 и 5:

1. Резервирует регион адресного пространства — такой, чтобы в него мог поместиться заданный DLL-файл. Желательное расположение этого региона указывается внутри самого DLL-файла. По умолчанию Microsoft Visual C++ присваивает DLL-модулям базовый адрес 0x10000000 (в 64-разрядной DLL под управлением 64-разрядной Windows 2000 этот адрес может быть другим). При компоновке DLL это значение можно изменить с помощью параметра /BASE. У всех стандартных системных DLL, поставляемых с Windows, разные базовые адреса, чтобы не допустить их перекрытия при загрузке в одно адресное пространство.
2. Если зарезервировать регион по желательному для DLL базовому адресу не удается (из-за того, что он слишком мал либо занят каким-то еще EXE- или DLL-файлом), система пытается найти другой регион. Но по двум причинам такая ситуация весьма неприятна. Во-первых, если в DLL нет информации о возможной переадресации (relocation information), загрузка может вообще не получиться. (Такую информацию можно удалить из DLL при компоновке с параметром /FIXED. Это уменьшит размер DLL-файла, но тогда модуль *должен* грузиться только по указанному базовому адресу.) Во-вторых, системе приходится выполнять модификацию адресов (relocations) внутри DLL. В Windows 98 эта операция осуществляется по мере подкачки страниц в оперативную память. Но в Windows 2000 на это уходит дополнительная физическая память, выделяемая из страничного файла, да и загрузка такой DLL займет больше времени.
3. Отмечает, что физическая память, связанная с зарезервированным регионом, — DLL-файл на диске, а не страничный файл. Если Windows 2000 пришлось выполнять модификацию адресов из-за того, что DLL не удалось загрузить по желательному базовому адресу, она запоминает, что часть физической памяти для DLL связана со страничным файлом.

Если система почему-либо не свяжет EXE-файл с необходимыми ему DLL, на экране появится соответствующее сообщение, а адресное пространство процесса и объект «процесс» будут освобождены. При этом *CreateProcess* вернет FALSE; прояснить причину сбоя поможет функция *GetLastError*.

После увязки EXE- и DLL-файлов с адресным пространством процесса начинает исполняться стартовый код EXE-файла. Подкачку страниц, буферизацию и кэширование система берет на себя. Например, если код в EXE-файле переходит к команде, не загруженной в память, возникает ошибка. Обнаружив ее, система перекачивает нужную страницу кода из образа файла на страницу оперативной памяти. Затем отображает страницу оперативной памяти на должный участок адресного пространства процесса, тем самым позволяя потоку продолжить выполнение кода. Все эти операции скрыты от приложения и периодически повторяются при каждой попытке процесса обратиться к коду или данным, отсутствующим в оперативной памяти.

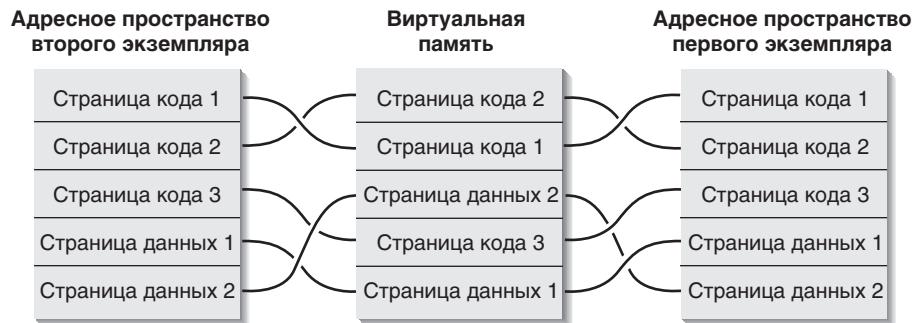
Статические данные не разделяются несколькими экземплярами EXE или DLL

Когда Вы создаете новый процесс для уже выполняемого приложения, система просто открывает другое проецируемое в память представление (view) объекта «проекция файла» (file-mapping object), идентифицирующего образ исполняемого файла, и создает новые объекты «процесс» и «поток» (для первичного потока). Этим объектам присваиваются идентификаторы процесса и потока. С помощью проецируемых в память файлов несколько одновременно выполняемых экземпляров приложения может совместно использовать один и тот же код, загруженный в оперативную память.

Здесь возникает небольшая проблема. Процессы используют линейное (flat) адресное пространство. При компиляции и компоновке программы весь ее код и данные объединяются в нечто, так сказать, большое и цельное. Данные, конечно, отделены от кода, но только в том смысле, что они расположены вслед за кодом в EXE-файле¹. Вот упрощенная иллюстрация того, как код и данные приложения загружаются в виртуальную память, а затем отображаются на адресное пространство процесса:



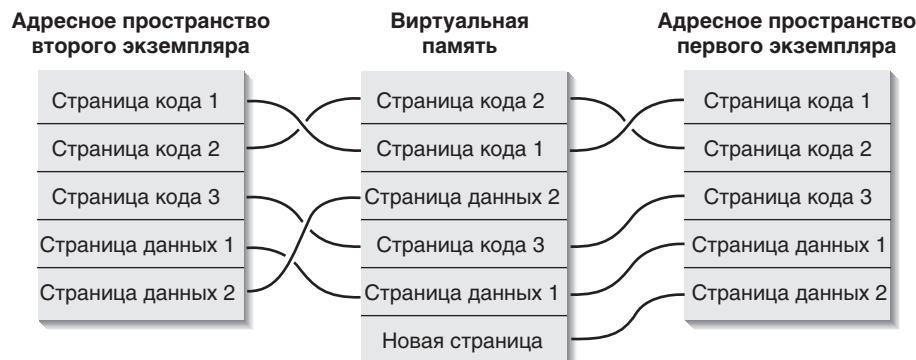
Теперь допустим, что запущен второй экземпляр программы. Система просто-напросто проецирует страницы виртуальной памяти, содержащие код и данные файла, на адресное пространство второго экземпляра приложения:



Если один экземпляр приложения модифицирует какие-либо глобальные переменные, размещенные на странице данных, содержимое памяти изменяется для всех экземпляров этого приложения. Такое изменение могло бы привести к катастрофическим последствиям и поэтому недопустимо.

¹ На самом деле содержимое файла разбито на отдельные разделы (sections). Код находится в одном разделе, а глобальные переменные — в другом. Разделы выравниваются по границам страниц. Приложение определяет размер страницы через функцию `GetSystemInfo`. В EXE- или DLL-файле раздел кода обычно предшествует разделу данных.

Система предотвращает подобные ситуации, применяя механизм копирования при записи. Всякий раз, когда программа пытается записывать что-то в файл, спроецированный в память, система перехватывает эту попытку, выделяет новый блок памяти, копирует в него нужную программе страницу и после этого разрешает запись в новый блок памяти. Благодаря этому работа остальных экземпляров программы не нарушается. Вот что получится, когда первый экземпляр программы попытается изменить какую-нибудь глобальную переменную на второй странице данных:



Система выделяет новую страницу и копирует на нее содержимое страницы данных 2. Адресное пространство первого экземпляра изменяется так, чтобы отобразить новую страницу данных на тот же участок, что и исходную. Теперь процесс может изменить глобальную переменную, не затрагивая данные другого экземпляра.

Аналогичная цепочка событий происходит и при отладке приложения. Например, запустив несколько экземпляров программы, Вы хотите отладить только один из них. Вызвав отладчик, Вы ставите в строке исходного кода точку прерывания. Отладчик модифицирует Ваш код, заменяя одну из команд на языке ассемблера другой — заставляющей активизировать сам отладчик. И здесь Вы сталкиваетесь с той же проблемой. После модификации кода все экземпляры программы, доходя до исполнения измененной команды, приводили бы к его активизации. Чтобы этого избежать, система вновь использует копирование при записи. Обнаружив попытку отладчика изменить код, она выделяет новый блок памяти, копирует туда нужную страницу и позволяет отладчику модифицировать код на этой копии.

-
- WINDOWS 98** При загрузке процесса система просматривает все страницы образа файла. Физическая память из страничного файла передается сразу только тем страницам, которые должны быть защищены атрибутом копирования при записи. При обращении к такому участку образа файла в память загружается соответствующая страница. Если ее модификации не происходят, она может быть выгружена из памяти и при необходимости загружена вновь. Если же страница файла модифицируется, система перекачивает ее на одну из ранее переданных страниц в страничном файле.

Поведение Windows 2000 и Windows 98 в подобных случаях одинаково, кроме ситуации, когда в память загружено два экземпляра одного модуля и никаких данных не изменено. Тогда процессы под управлением Windows 2000 могут совместно использовать данные, а в Windows 98 каждый процесс получает свою копию этих данных. Но если в память загружен лишь один экземпляр модуля или же данные были модифицированы (что чаще всего и бывает), Windows 2000 и Windows 98 ведут себя одинаково.

Статические данные разделяются несколькими экземплярами EXE или DLL

По умолчанию для большей безопасности глобальные и статические данные не разделяются несколькими проекциями одного и того же EXE или DLL. Но иногда удобнее, чтобы несколько проекций EXE разделяли единственный экземпляр переменной. Например, в Windows не так-то просто определить, запущено ли несколько экземпляров приложения. Если бы у Вас была переменная, доступная всем экземплярам приложения, она могла бы отражать число этих экземпляров. Тогда при запуске нового экземпляра приложения его поток просто проверил бы значение глобальной переменной (обновленное другим экземпляром приложения) и, будь оно больше 1, сообщил бы пользователю, что запустить можно лишь один экземпляр; после чего эта копия приложения была бы завершена.

В этом разделе мы рассмотрим метод, обеспечивающий совместное использование переменных всеми экземплярами EXE или DLL. Но сначала Вам понадобятся некоторые базовые сведения.

Любой образ EXE- или DLL-файла состоит из группы разделов. По соглашению имени каждого стандартного раздела начинается с точки. Например, при компиляции программы весь код помещается в раздел *.text*, неинициализированные данные — в раздел *.bss*, а инициализированные — в раздел *.data*.

С каждым разделом связана одна из комбинаций атрибутов, перечисленных в следующей таблице.

Атрибут	Описание
READ	Разрешает чтение из раздела
WRITE	Разрешает запись в раздел
EXECUTE	Содержимое раздела можно исполнять
SHARED	Раздел доступен нескольким экземплярам приложения (этот атрибут отключает механизм копирования при записи)

Запустив утилиту DumpBin из Microsoft Visual Studio (с ключом /Headers), Вы увидите список разделов в файле образа EXE или DLL. Пример такого списка, показанный ниже, относится к EXE-файлу.

```
SECTION HEADER #1
.text name
11A70 virtual size
1000 virtual address
12000 size of raw data
1000 file pointer to raw data
0 file pointer to relocation table
0 file pointer to line numbers
0 number of relocations
0 number of line numbers
60000020 flags
Code
Execute Read
```

```
SECTION HEADER #2
.rdata name
1F6 virtual size
```

см. след. стр.

```
13000 virtual address
 1000 size of raw data
13000 file pointer to raw data
  0 file pointer to relocation table
  0 file pointer to line numbers
  0 number of relocations
  0 number of line numbers
40000040 flags
  Initialized Data
  Read Only
```

SECTION HEADER #3

```
.data name
  560 virtual size
14000 virtual address
 1000 size of raw data
14000 file pointer to raw data
  0 file pointer to relocation table
  0 file pointer to line numbers
  0 number of relocations
  0 number of line numbers
C0000040 flags
  Initialized Data
  Read Write
```

SECTION HEADER #4

```
.idata name
  580 virtual size
15000 virtual address
 1000 size of raw data
15000 file pointer to raw data
  0 file pointer to relocation table
  0 file pointer to line numbers
  0 number of relocations
  0 number of line numbers
C0000040 flags
  Initialized Data
  Read Write
```

SECTION HEADER #5

```
.didat name
  7A2 virtual size
16000 virtual address
 1000 size of raw data
16000 file pointer to raw data
  0 file pointer to relocation table
  0 file pointer to line numbers
  0 number of relocations
  0 number of line numbers
C0000040 flags
  Initialized Data
  Read Write
```

```

SECTION HEADER #6
  .reloc name
    26D virtual size
  17000 virtual address
    1000 size of raw data
  17000 file pointer to raw data
    0 file pointer to relocation table
    0 file pointer to line numbers
    0 number of relocations
    0 number of line numbers
42000040 flags
  Initialized Data
  Discardable
  Read Only

Summary
  1000 .data
  1000 .didat
  1000 .idata
  1000 .rdata
  1000 .reloc
  12000 .text

```

Некоторые из часто встречающихся разделов перечислены в таблице ниже.

Имя раздела	Описание
.bss	Неинициализированные данные
.CRT	Неизменяемые данные библиотеки С
.data	Инициализированные данные
.debug	Отладочная информация
.didat	Таблица имен для отложенного импорта (delay imported names table)
.edata	Таблица экспортимых имен
.idata	Таблица импортируемых имен
.rdata	Неизменяемые данные периода выполнения
.reloc	Настроечная информация — таблица переадресации (relocation table)
.rsrc	Ресурсы
.text	Код EXE или DLL
.tls	Локальная память потока
.xdata	Таблица для обработки исключений

Кроме стандартных разделов, генерируемых компилятором и компоновщиком, можно создавать свои разделы в EXE- или DLL-файле, используя директиву компилятора:

```
#pragma data_seg("имя_раздела")
```

Например, можно создать раздел Shared, в котором содержится единственная переменная типа LONG:

```
#pragma data_seg("Shared")
LONG g_lInstanceCount = 0;
#pragma data_seg()
```

Обрабатывая этот код, компилятор создаст раздел Shared и поместит в него все *инициализированные* переменные, встретившиеся после директивы *#pragma*. В нашем примере в этом разделе находится переменная *g_InstanceCount*. Директива *#pragma data_seg()* сообщает компилятору, что следующие за ней переменные нужно вновь помещать в стандартный раздел данных, а не в Shared. Важно помнить, что компилятор помещает в новый раздел только инициализированные переменные. Если из предыдущего фрагмента кода исключить инициализацию переменной, она будет включена в другой раздел:

```
#pragma data_seg("Shared")
LONG g_lInstanceCount;
#pragma data_seg()
```

Однако в компиляторе Microsoft Visual C++ 6.0 предусмотрен спецификатор *allocate*, который позволяет помещать неинициализированные данные в любой раздел. Взгляните на этот код:

```
// создаем раздел Shared и заставляем компилятор
// поместить в него инициализированные данные
#pragma data_seg("Shared")

// инициализированная переменная, по умолчанию помещается в раздел Shared
int a = 0;

// неинициализированная переменная, по умолчанию помещается в другой раздел
int b;

// сообщаем компилятору прекратить включение инициализированных данных
// в раздел Shared
#pragma data_seg()

// инициализированная переменная, принудительно помещается в раздел Shared
__declspec(allocate("Shared")) int c = 0;

// неинициализированная переменная, принудительно помещается в раздел Shared
__declspec(allocate("Shared")) int d;

// инициализированная переменная, по умолчанию помещается в другой раздел
int e = 0;

// неинициализированная переменная, по умолчанию помещается в другой раздел
int f;
```

Чтобы спецификатор *allocate* работал корректно, сначала должен быть создан соответствующий раздел. Так что, убрав из предыдущего фрагмента кода первую строку *#pragma data_seg*, Вы не смогли бы его скомпилировать.

Чаще всего переменные помещают в собственные разделы, намереваясь сделать их разделяемыми между несколькими проекциями EXE или DLL. По умолчанию каждая проекция получает свой набор переменных. Но можно сгруппировать в отдельном разделе переменные, которые должны быть доступны всем проекциям EXE или DLL; тогда система не станет создавать новые экземпляры этих переменных для каждой проекции EXE или DLL.

Чтобы переменные стали разделяемыми, одного указания компилятору выделить их в какой-то раздел мало. Надо также сообщить компоновщику, что переменные в

в этом разделе должны быть общими. Для этого предназначен ключ `/SECTION` компоновщика:

```
/SECTION:имя, атрибуты
```

За двоеточием укажите имя раздела, атрибуты которого Вы хотите изменить. В нашем примере нужно изменить атрибуты раздела `Shared`, поэтому ключ должен выглядеть так:

```
/SECTION:Shared, RWS
```

После запятой мы задаем требуемые атрибуты. При этом используются такие сокращения: `R` (READ), `W` (WRITE), `E` (EXECUTE) и `S` (SHARED). В данном случае мы указали, что раздел `Shared` должен быть «читаемым», «записываемым» и «разделяемым». Если Вы хотите изменить атрибуты более чем у одного раздела, указывайте ключ `/SECTION` для каждого такого раздела.

Соответствующие директивы для компоновщика можно вставлять прямо в исходный код:

```
#pragma comment(linker, "/SECTION:Shared, RWS")
```

Эта строка заставляет компилятор включить строку `</SECTION: Shared,RWS>` в особый раздел *directive*. Компоновщик, собирая OBJ-модули, проверяет этот раздел в каждом OBJ-модуле и действует так, словно все эти строки переданы ему как аргументы в командной строке. Я всегда применяю этот очень удобный метод: перемещая файл исходного кода в новый проект, не надо изменять никаких параметров в диалоговом окне Project Settings в Visual C++.

Хотя создавать общие разделы можно, Microsoft не рекомендует это делать. Во-первых, разделение памяти таким способом может нарушить защиту. Во-вторых, наличие общих переменных означает, что ошибка в одном приложении повлияет на другое, так как этот блок данных не удастся защитить от случайной записи.

Представьте, Вы написали два приложения, каждое из которых требует от пользователя вводить пароль. При этом Вы решили чуть-чуть облегчить жизнь пользователю: если одна из программ уже выполняется на момент запуска другой, то вторая считывает пароль из общей памяти. Так что пользователю не нужно повторно вводить пароль, если одно из приложений уже запущено.

Все выглядит вполне невинно. В конце концов только Ваши приложения загружают данную DLL, и только они знают, где искать пароль, содержащийся в общем разделе памяти. Но хакеры не дремлют, и если им захочется узнать Ваш пароль, то максимум, что им понадобится, — написать небольшую программу, загружающую Вашу DLL, и понаблюдать за общим блоком памяти. Когда пользователь введет пароль, хакерская программа тут же его узнает.

Трудолюбивая хакерская программа может также предпринять серию попыток угадать пароль, записывая его варианты в общую память. А угадав, сможет посыпать любые команды этим двум приложениям. Данную проблему можно было бы решить, если бы существовал какой-нибудь способ разрешать загрузку DLL только определенным программам. Но пока это невозможно — любая программа, вызвав `LoadLibrary`, способна явно загрузить любую DLL.

Программа-пример AppInst

Эта программа, «17 AppInst.exe» (см. листинг на рис. 17-1), демонстрирует, как выяснить, сколько экземпляров приложения уже выполняется в системе. Файлы исходного кода и ресурсов этой программы находятся в каталоге 17-AppInst на компакт-диске.

ке, прилагаемом к книге. После запуска AppInst на экране появляется диалоговое окно, в котором сообщается, что сейчас выполняется только один ее экземпляр.



Если Вы запустите второй экземпляр, оба диалоговых окна сообщат, что теперь выполняется два экземпляра.



Вы можете запускать и закрывать сколько угодно экземпляров этой программы — окно любого из них всегда будет отражать точное количество выполняемых экземпляров.

Где-то в начале файла AppInst.cpp Вы заметите следующие строки:

```
// указываем компилятору поместить эту инициализированную переменную  
// в раздел Shared, чтобы она стала доступной всем экземплярам программы  
#pragma data_seg("Shared")  
volatile LONG g_lApplicationInstances = 0;  
#pragma data_seg()  
  
// указываем компоновщику, что раздел Shared должен быть  
// читаемым, записываемым и разделяемым  
#pragma comment(linker, "/Section:Shared,RWS")
```

В этих строках кода создается раздел Shared с атрибутами защиты, которые разрешают его чтение, запись и разделение. Внутри него находится одна переменная, *g_lApplicationInstances*, доступная всем экземплярам программы. Заметьте, что для этой переменной указан спецификатор *volatile*, чтобы оптимизатор не слишком с ней умничал.

При выполнении функции *_tWinMain* каждого экземпляра значение переменной *g_lApplicationInstances* увеличивается на 1, а перед выходом из *_tWinMain* — уменьшается на 1. Я изменяю ее значение с помощью функции *InterlockedExchangeAdd*, так как эта переменная является общим ресурсом для нескольких потоков.

Когда на экране появляется диалоговое окно каждого экземпляра программы, вызывается функция *Dlg_OnInitDialog*. Она рассыпает всем окнам верхнего уровня зарегистрированное оконное сообщение (идентификатор которого содержится в переменной *g_aMsgAppInstCountUpdate*):

```
PostMessage(HWND_BROADCAST, g_aMsgAppInstCountUpdate, 0, 0);
```

Это сообщение игнорируется всеми окнами в системе, кроме окон AppInst. Когда его принимает одно из окон нашей программы, код в *Dlg_Proc* просто обновляет в диалоговом окне значение, отражающее текущее количество экземпляров (а эта величина хранится в переменной *g_lApplicationInstances*).

**AppInst.cpp**

```
*****
Modуль: AppInst.cpp
Автор: Copyright (c) 2000, Джейфри Рихтер (Jeffrey Richter)
*****
```

```
#include "..\CmnHdr.h"      /* см. приложение A */
#include <windowsx.h>
#include <tchar.h>
#include "Resource.h"

//////////
```

```
// общесистемное оконное сообщение с уникальным идентификатором
UINT g_uMsgAppInstCountUpdate = INVALID_ATOM;

//////////
```

```
// указываем компилятору поместить эту инициализированную переменную
// в раздел Shared, чтобы она стала доступной всем экземплярам программы
#pragma data_seg("Shared")
volatile LONG g_lApplicationInstances = 0;
#pragma data_seg()
```

```
// указываем компоновщику, что раздел Shared должен быть
// читаемым, записываемым и разделяемым
#pragma comment(linker, "/Section:Shared,RWS")
```

```
//////////
```

```
BOOL Dlg_OnInitDialog(HWND hwnd, HWND hwndFocus, LPARAM lParam) {
    chSETDLGICONS(hwnd, IDI_APPINST);

    // инициализируем статический элемент управления
    PostMessage(HWND_BROADCAST, g_uMsgAppInstCountUpdate, 0, 0);
    return(TRUE);
}
```

```
//////////
```

```
void Dlg_OnCommand(HWND hwnd, int id, HWND hwndCtl, UINT codeNotify) {
    switch (id) {
        case IDCANCEL:
            EndDialog(hwnd, id);
            break;
    }
}
```

```
//////////
```

Рис. 17-1. Программа-пример *AppInst*

см. след. стр

Рис. 17-1. продолжение

```

INT_PTR WINAPI Dlg_Proc(HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam) {

    if (uMsg == g_uMsgAppInstCountUpdate) {
        SetDlgItemInt(hwnd, IDC_COUNT, g_lApplicationInstances, FALSE);
    }

    switch (uMsg) {
        chHANDLE_DLGMMSG(hwnd, WM_INITDIALOG, Dlg_OnInitDialog);
        chHANDLE_DLGMMSG(hwnd, WM_COMMAND, Dlg_OnCommand);
    }
    return(FALSE);
}

/////////////////////////////// Конец файла //////////////////////

int WINAPI _tWinMain(HINSTANCE hinstExe, HINSTANCE, LPTSTR pszCmdLine, int) {

    // получаем числовое значение из общесистемного оконного сообщения,
    // которое применяется для уведомления всех окон верхнего уровня
    // об изменении счетчика числа пользователей данного модуля
    g_uMsgAppInstCountUpdate =
        RegisterWindowMessage(TEXT("MsgAppInstCountUpdate"));

    // запущен еще один экземпляр этой программы
    InterlockedExchangeAdd((PLONG) &g_lApplicationInstances, 1);

    DialogBox(hinstExe, MAKEINTRESOURCE(IDD_APPINST), NULL, Dlg_Proc);

    // данный экземпляр закрывается
    InterlockedExchangeAdd((PLONG) &g_lApplicationInstances, -1);

    // сообщаем об этом остальным экземплярам программы
    PostMessage(HWND_BROADCAST, g_uMsgAppInstCountUpdate, 0, 0);

    return(0);
}

/////////////////////////////// Конец файла /////////////////////

```

Файлы данных, проецируемые в память

Операционная система позволяет проецировать на адресное пространство процесса и файл данных. Это очень удобно при манипуляциях с большими потоками данных.

Чтобы представить всю мощь такого применения механизма проецирования файлов, рассмотрим четыре возможных метода реализации программы, меняющей порядок следования всех байтов в файле на обратный.

Метод 1: один файл, один буфер

Первый (и теоретически простейший) метод — выделение блока памяти, достаточного для размещения всего файла. Открываем файл, считываем его содержимое в блок памяти, закрываем. Располагая в памяти содержимым файла, можно поменять первый

байт с последним, второй — с предпоследним и т. д. Этот процесс будет продолжаться, пока мы не поменяем местами два смежных байта, находящихся в середине файла. Закончив эту операцию, вновь открываем файл и перезаписываем его содержимое.

Этот довольно простой в реализации метод имеет два существенных недостатка. Во-первых, придется выделить блок памяти такого же размера, что и файл. Это терпимо, если файл небольшой. А если он занимает 2 ГБ? Система просто не позволит приложению передать такой объем физической памяти. Значит, к большим файлам нужен совершенно иной подход.

Во-вторых, если перезапись вдруг прервется, содержимое файла будет испорчено. Простейшая мера предосторожности — создать копию исходного файла (потом ее можно удалить), но это потребует дополнительного дискового пространства.

Метод 2: два файла, один буфер

Открываем существующий файл и создаем на диске новый — нулевой длины. Затем выделяем небольшой внутренний буфер размером, скажем, 8 КБ. Устанавливаем указатель файла в позицию 8 КБ от конца, считываем в буфер последние 8 КБ содержимого файла, меняем в нем порядок следования байтов на обратный и переписываем буфер в только что созданный файл. Повторяем эти операции, пока не дойдем до начала исходного файла. Конечно, если длина файла не будет кратна 8 КБ, операции придется немного усложнить, но это не страшно. Закончив обработку, закрываем оба файла и удаляем исходный файл.

Этот метод посложнее первого, зато позволяет гораздо эффективнее использовать память, так как требует выделения лишь 8 КБ. Но и здесь не без проблем, и вот две главных. Во-первых, обработка идет медленнее, чем при первом методе: на каждой итерации перед считыванием приходится находить нужный фрагмент исходного файла. Во-вторых, может понадобиться огромное пространство на жестком диске. Если длина исходного файла 400 МБ, новый файл постепенно вырастет до этой величины, и перед самым удалением исходного файла будет занято 800 МБ, т. е. на 400 МБ больше, чем следовало бы. Так что все пути ведут... к третьему методу.

Метод 3: один файл, два буфера

Программа инициализирует два раздельных буфера, допустим, по 8 КБ и считывает первые 8 КБ файла в один буфер, а последние 8 КБ — в другой. Далее содержимое обоих буферов обменивается в обратном порядке и первый буфер записывается в конец, а второй — в начало того же файла. На каждой итерации программа перемещает восемькилобайтовые блоки из одной половины файла в другую. Разумеется, нужно предусмотреть какую-то обработку на случай, если длина файла не кратна 16 КБ, и эта обработка будет куда сложнее, чем в предыдущем методе. Но разве это испугает опытного программиста?

По сравнению с первыми двумя этот метод позволяет экономить пространство на жестком диске, так как все операции чтения и записи протекают в рамках одного файла. Что же касается памяти, то и здесь данный метод довольно эффективен, используя всего 16 КБ. Однако он, по-видимому, самый сложный в реализации. И, кроме того, как и первый метод, он может испортить файл данных, если процесс вдруг прервется.

Ну а теперь посмотрим, как тот же процесс реализуется, если применить файлы, проецируемые в память.

Метод 4: один файл и никаких буферов

Вы открываете файл, указывая системе зарезервировать регион виртуального адресного пространства. Затем сообщаете, что первый байт файла следует спроектировать на первый байт этого региона, и обращаетесь к региону так, будто он на самом деле содержит файл. Если в конце файла есть отдельный нулевой байт, можно вызвать библиотечную функцию `_strrev` и поменять порядок следования байтов на обратный.

Огромный плюс этого метода в том, что всю работу по кэшированию файла выполняет сама система: не надо выделять память, загружать данные из файла в память, переписывать их обратно в файл и т. д. и т. п. Но, увы, вероятность прерывания процесса, например из-за сбоя в электросети, по-прежнему сохраняется, и от порчи данных Вы не застрахованы.

Использование проецируемых в память файлов

Для этого нужно выполнить три операции:

1. Создать или открыть объект ядра «файл», идентифицирующий дисковый файл, который Вы хотите использовать как проецируемый в память.
2. Создать объект ядра «проекция файла», чтобы сообщить системе размер файла и способ доступа к нему.
3. Указать системе, как спроектировать в адресное пространство Вашего процесса объект «проекция файла» — целиком или частично.

Закончив работу с проецируемым в память файлом, следует выполнить тоже три операции:

1. Сообщить системе об отмене проектирования на адресное пространство процесса объекта ядра «проекция файла».
2. Закрыть этот объект.
3. Закрыть объект ядра «файл».

Детальное рассмотрение этих операций — в следующих пяти разделах.

Этап 1: создание или открытие объекта ядра «файл»

Для этого Вы должны применять только функцию `CreateFile`:

```
HANDLE CreateFile(
    PCSTR pszFileName,
    DWORD dwDesiredAccess,
    DWORD dwShareMode,
    PSECURITY_ATTRIBUTES psa,
    DWORD dwCreationDisposition,
    DWORD dwFlagsAndAttributes,
    HANDLE hTemplateFile);
```

Как видите, у функции `CreateFile` довольно много параметров. Здесь я сосредоточусь только на первых трех: `pszFileName`, `dwDesiredAccess` и `dwShareMode`.

Как Вы, наверное, догадались, первый параметр, `pszFileName`, идентифицирует имя создаваемого или открываемого файла (при необходимости вместе с путем). Второй параметр, `dwDesiredAccess`, указывает способ доступа к содержимому файла. Здесь задается одно из четырех значений, показанных в таблице ниже.

Значение	Описание
0	Содержимое файла нельзя считывать или записывать; указывайте это значение, если Вы хотите всего лишь получить атрибуты файла
GENERIC_READ	Чтение файла разрешено
GENERIC_WRITE	Запись в файл разрешена
GENERIC_READ	
GENERIC_WRITE	Разрешено и то и другое

Создавая или открывая файл данных с намерением использовать его в качестве проецируемого в память, можно установить либо флаг GENERIC_READ (только для чтения), либо комбинированный флаг GENERIC_READ | GENERIC_WRITE (чтение/запись).

Третий параметр, *dwShareMode*, указывает тип совместного доступа к данному файлу (см. следующую таблицу).

Значение	Описание
0	Другие попытки открыть файл закончатся неудачно
FILE_SHARE_READ	Попытка постороннего процесса открыть файл с флагом GENERIC_WRITE не удастся
FILE_SHARE_WRITE	Попытка постороннего процесса открыть файл с флагом GENERIC_READ не удастся
FILE_SHARE_READ FILE_SHARE_WRITE	Посторонний процесс может открывать файл без ограничений

Создав или открыв указанный файл, *CreateFile* возвращает его описатель, в ином случае — идентификатор INVALID_HANDLE_VALUE.



Большинство функций Windows, возвращающих те или иные описатели, при неудачном вызове дает NULL. Но *CreateFile* — исключение и в таких случаях возвращает идентификатор INVALID_HANDLE_VALUE, определенный как ((HANDLE) -1).

Этап 2: создание объекта ядра «проекция файла»

Вызвав *CreateFile*, Вы указали операционной системе, где находится физическая память для проекции файла: на жестком диске, в сети, на CD-ROM или в другом месте. Теперь сообщите системе, какой объем физической памяти нужен проекции файла. Для этого вызовите функцию *CreateFileMapping*:

```
HANDLE CreateFileMapping(
    HANDLE hFile,
    PSECURITY_ATTRIBUTES psa,
    DWORD fdwProtect,
    DWORD dwMaximumSizeHigh,
    DWORD dwMaximumSizeLow,
    PCSTR pszName);
```

Первый параметр, *hFile*, идентифицирует описатель файла, проецируемого на адресное пространство процесса. Этот описатель Вы получили после вызова *CreateFile*. Параметр *psa* — указатель на структуру SECURITY_ATTRIBUTES, которая относится к объекту ядра «проекция файла»; для установки защиты по умолчанию ему присваивается NULL.

Как я уже говорил в начале этой главы, создание файла, проецируемого в память, аналогично резервированию региона адресного пространства с последующей передачей ему физической памяти. Разница лишь в том, что физическая память для проецируемого файла — сам файл на диске, и для него не нужно выделять пространство в страничном файле. При создании объекта «проекция файла» система не резервирует регион адресного пространства и не увязывает его с физической памятью из файла (как это сделать, я расскажу в следующем разделе). Но, как только дело дойдет до отображения физической памяти на адресное пространство процесса, системе понадобится точно знать атрибут защиты, присваиваемый страницам физической памяти. Поэтому в *fdwProtect* надо указать желательные атрибуты защиты. Обычно используется один из перечисленных в следующей таблице.

Атрибут защиты	Описание
PAGE_READONLY	Отобразив объект «проекция файла» на адресное пространство, можно считывать данные из файла. При этом Вы должны были передать в <i>CreateFile</i> флаг GENERIC_READ.
PAGE_READWRITE	Отобразив объект «проекция файла» на адресное пространство, можно считывать данные из файла и записывать их. При этом Вы должны были передать в <i>CreateFile</i> комбинацию флагов GENERIC_READ GENERIC_WRITE.
PAGE_WRITECOPY	Отобразив объект «проекция файла» на адресное пространство, можно считывать данные из файла и записывать их. Запись приведет к созданию закрытой копии страницы. При этом Вы должны были передать в <i>CreateFile</i> либо GENERIC_READ, либо GENERIC_READ GENERIC_WRITE.

WINDOWS 98 В Windows 98 функции *CreateFileMapping* можно передать флаг PAGE_WRITECOPY; тем самым Вы скажете системе передать физическую память из страничного файла. Эта память резервируется для копии информации из файла данных, и лишь модифицированные страницы действительно записываются в страничный файл. Изменения не распространяются на исходный файл данных. Результат применения флага PAGE_WRITECOPY одинаков в Windows 2000 и в Windows 98.

Кроме рассмотренных выше атрибутов защиты страницы, существует еще и четыре атрибута раздела; их можно ввести в параметр *fdwProtect* функции *CreateFileMapping* побитовой операцией OR. Раздел (section) — всего лишь еще одно название проекции памяти.

Первый из этих атрибутов, SEC_NOCACHE, сообщает системе, что никакие страницы файла, проецируемого в память, кэшировать не надо. В результате при записи данных в файл система будет обновлять данные на диске чаще обычного. Этот флаг, как и атрибут защиты PAGE_NOCACHE, предназначен для разработчиков драйверов устройств и обычно в приложениях не используется.

WINDOWS 98 Windows 98 игнорирует флаг SEC_NOCACHE.

Второй атрибут, SEC_IMAGE, указывает системе, что данный файл является переносимым исполняемым файлом (portable executable, PE). Отображая его на адресное пространство процесса, система просматривает содержимое файла, чтобы определить, какие атрибуты защиты следует присвоить различным страницам проецируе-

мого образа (mapped image). Например, раздел кода PE-файла (*.text*) обычно проецируется с атрибутом PAGE_EXECUTE_READ, тогда как раздел данных этого же файла (*.data*) — с атрибутом PAGE_READWRITE. Атрибут SEC_IMAGE заставляет систему спроектировать образ файла и автоматически подобрать подходящие атрибуты защиты страниц.

WINDOWS 98 Windows 98 игнорирует флаг SEC_IMAGE.

Последние два атрибута (SEC_RESERVE и SEC_COMMIT) взаимоисключают друг друга и неприменимы для проецирования в память файла данных. Эти флаги мы рассмотрим ближе к концу главы. *CreateFileMapping* их игнорирует.

Следующие два параметра этой функции (*dwMaximumSizeHigh* и *dwMaximumSizeLow*) самые важные. Основное назначение *CreateFileMapping* — гарантировать, что объекту «проекция файла» доступен нужный объем физической памяти. Через эти параметры мы сообщаем системе максимальный размер файла в байтах. Так как Windows позволяет работать с файлами, размеры которых выражаются 64-разрядными числами, в параметре *dwMaximumSizeHigh* указываются старшие 32 бита, а в *dwMaximumSizeLow* — младшие 32 бита этого значения. Для файлов размером менее 4 Гб *dwMaximumSizeHigh* всегда равен 0. Наличие 64-разрядного значения подразумевает, что Windows способна обрабатывать файлы длиной до 16 экзабайтов.

Для создания объекта «проекция файла» таким, чтобы он отражал текущий размер файла, передайте в обоих параметрах нули. Так же следует поступить, если Вы собираетесь ограничиться считыванием или как-то обработать файл, не меняя его размер. Для дозаписи данных в файл выбирайте его размер максимальным, чтобы оставить пространство «для маневра». Если в данный момент файл на диске имеет нулевую длину, в параметрах *dwMaximumSizeHigh* и *dwMaximumSizeLow* нельзя передавать нули. Иначе система решит, что Вам нужна проекция файла с объемом памяти, равным 0. А это ошибка, и *CreateFileMapping* вернет NULL.

Если Вы еще следите за моими рассуждениями, то, должно быть, подумали: что-то тут не все ладно. Очень, конечно, мило, что Windows поддерживает файлы и их проекции размером вплоть до 16 экзабайтов, но как, интересно, спроектировать такой файл на адресное пространство 32-разрядного процесса, ограниченное 4 Гб, из которых и использовать-то можно только 2 Гб? На этот вопрос я отвечу в следующем разделе. (Конечно, адресное пространство 64-разрядного процесса, размер которого составляет 16 экзабайтов, позволяет работать с еще большими проекциями файлов, но аналогичное ограничение существует и там.)

Чтобы досконально разобраться, как работают функции *CreateFile* и *CreateFileMapping*, предлагаю один эксперимент. Возьмите код, приведенный ниже, соберите его и запустите под отладчиком. Пошагово выполняя операторы, переключитесь в окно командного процессора и запросите содержимое каталога «C:\» командой dir. Обратите внимание на изменения, происходящие в каталоге при выполнении каждого оператора.

```
int WINAPI _tWinMain(HINSTANCE hinstExe, HINSTANCE,
PTSTR pszCmdLine, int nCmdShow) {

// перед выполнением этого оператора, в каталоге C:\  

// еще нет файла "MMFTest.dat"
HANDLE hfile = CreateFile("C:\\\\MMFTest.dat",
```

см. след. стр.

```
    GENERIC_READ | GENERIC_WRITE, FILE_SHARE_READ | FILE_SHARE_WRITE, NULL,
    CREATE_ALWAYS, FILE_ATTRIBUTE_NORMAL, NULL);

    // перед выполнением этого оператора файл MMFTTest.dat существует,
    // но имеет нулевую длину
    HANDLE hfilemap = CreateFileMapping(hfile, NULL, PAGE_READWRITE, 0, 100, NULL);

    // после выполнения предыдущего оператора размер файла MMFTTest.dat
    // возрастает до 100 байтов

    // очистка
    CloseHandle(hfilemap);
    CloseHandle(hfile);

    // по завершении процесса файл MMFTTest.dat останется
    // на диске и будет иметь длину 100 байтов
    return(0);
}
```

Вызов *CreateFileMapping* с флагом *PAGE_READWRITE* заставляет систему проверять, чтобы размер соответствующего файла данных на диске был не меньше, чем указано в параметрах *dwMaximumSizeHigh* и *dwMaximumSizeLow*. Если файл окажется меньше заданного, *CreateFileMapping* увеличит его размер до указанной величины. Это делается специально, чтобы выделить физическую память перед использованием файла в качестве проецируемого в память. Если объект «проекция файла» создан с флагом *PAGE_READONLY* или *PAGE_WRITECOPY*, то размер, переданный функции *CreateFileMapping*, не должен превышать физический размер файла на диске (так как Вы не сможете что-то дописать в файл).

Последний параметр функции *CreateFileMapping* — *pszName* — строка с нулевым байтом в конце; в ней указывается имя объекта «проекция файла», которое используется для доступа к данному объекту из другого процесса (пример см. в главе 3). Но обычно совместное использование проецируемого в память файла не требуется, и поэтому в данном параметре передают NULL.

Система создает объект «проекция файла» и возвращает его описатель в вызвавший функцию поток. Если объект создать не удалось, возвращается нулевой описатель (NULL). И здесь еще раз обратите внимание на отличительную особенность функции *CreateFile* — при ошибке она возвращает не NULL, а идентификатор INVALID_HANDLE_VALUE (определенный как -1).

Этап 3: проецирование файловых данных на адресное пространство процесса

Когда объект «проекция файла» создан, нужно, чтобы система, зарезервировав регион на адресного пространства под данные файла, передала их как физическую память, отображенную на регион. Это делает функция *MapViewOfFile*:

```
PVOID MapViewOfFile(
    HANDLE hFileMappingObject,
    DWORD dwDesiredAccess,
    DWORD dwFileOffsetHigh,
    DWORD dwFileOffsetLow,
    SIZE_T dwNumberOfBytesToMap);
```

Параметр *bFileMappingObject* идентифицирует описатель объекта «проекция файла», возвращаемый предшествующим вызовом либо *CreateFileMapping*, либо *OpenFileMapping* (ее мы рассмотрим чуть позже). Параметр *dwDesiredAccess* идентифицирует вид доступа к данным. Все правильно: придется опять указывать, как именно мы хотим обращаться к файловым данным. Можно задать одно из четырех значений, описанных в следующей таблице.

Значение	Описание
FILE_MAP_WRITE	Файловые данные можно считывать и записывать; Вы должны были передать функции <i>CreateFileMapping</i> атрибут PAGE_READWRITE
FILE_MAP_READ	Файловые данные можно только считывать; Вы должны были вызвать <i>CreateFileMapping</i> с любым из следующих атрибутов: PAGE_READONLY, PAGE_READWRITE или PAGE_WRITECOPY
FILE_MAP_ALL_ACCESS	То же, что и FILE_MAP_WRITE
FILE_MAP_COPY	Файловые данные можно считывать и записывать, но запись приводит к созданию закрытой копии страницы; Вы должны были вызвать <i>CreateFileMapping</i> с любым из следующих атрибутов: PAGE_READONLY, PAGE_READWRITE или PAGE_WRITECOPY (Windows 98 требует вызывать <i>CreateFileMapping</i> с атрибутом PAGE_WRITECOPY)

Кажется странным и немного раздражает, что Windows требует бесконечно указывать все эти атрибуты защиты. Могу лишь предположить, что это сделано для того, чтобы приложение максимально полно контролировало защиту данных.

Остальные три параметра относятся к резервированию региона адресного пространства и к отображению на него физической памяти. При этом необязательно проецировать на адресное пространство весь файл сразу. Напротив, можно спроектировать лишь малую его часть, которая в таком случае называется представлением (view) — теперь-то Вам, наверное, понятно, откуда произошло название функции *MapViewOfFile*.

Проектируя на адресное пространство процесса представление файла, нужно сделать две вещи. Во-первых, сообщить системе, какой байт файла данных считать в представлении первым. Для этого предназначены параметры *dwFileOffsetHigh* и *dwFileOffsetLow*. Поскольку Windows поддерживает файлы длиной до 16 экзабайтов, приходится определять смещение в файле как 64-разрядное число: старшие 32 бита передаются в параметре *dwFileOffsetHigh*, а младшие 32 бита — в параметре *dwFileOffsetLow*. Заметьте, что смещение в файле должно быть кратно гранулярности выделения памяти в данной системе. (В настоящее время во всех реализациях Windows она составляет 64 Кб.) О гранулярности выделения памяти см. раздел «Системная информация» в главе 14.

Во-вторых, от Вас потребуется указать размер представления, т. е. сколько байтов файла данных должно быть спроектировано на адресное пространство. Это равносильно тому, как если бы Вы задали размер региона, резервируемого в адресном пространстве. Размер указывается в параметре *dwNumberOfBytesToMap*. Если этот параметр равен 0, система попытается спроектировать представление, начиная с указанного смещения и до конца файла.

WINDOWS 98 В Windows 98, если *MapViewOfFile* не найдет регион, достаточно большой для размещения всего объекта «проекция файла», возвращается NULL — независимо от того, какой размер представления был запрошен.

WINDOWS 2000 В Windows 2000 функция *MapViewOfFile* ищет регион, достаточно большой для размещения запрошенного представления, не обращая внимания на размер самого объекта «проекция файла».

Если при вызове *MapViewOfFile* указан флаг FILE_MAP_COPY, система передаст физическую память из страничного файла. Размер передаваемого пространства определяется параметром *dwNumberOfBytesToMap*. Пока Вы лишь считываете данные из представления файла, страницы, переданные из страничного файла, не используются. Но стоит какому-нибудь потоку в Вашем процессе совершить попытку записи по адресу, попадающему в границы представления файла, как система тут же берет из страничного файла одну из переданных страниц, копирует на нее исходные данные и проецирует ее на адресное пространство процесса. Так что с этого момента потоки Вашего процесса начинают обращаться к локальной копии данных и теряют доступ к исходным данным.

Создав копию исходной страницы, система меняет ее атрибут защиты с PAGE_WRITECOPY на PAGE_READWRITE. Рассмотрим пример:

```
// открываем файл, который мы собираемся спроектировать
HANDLE hFile = CreateFile(pszFileName, GENERIC_READ | GENERIC_WRITE, 0, NULL,
    OPEN_ALWAYS, FILE_ATTRIBUTE_NORMAL, NULL);

// создаем для файла объект "проекция файла"
HANDLE hFileMapping = CreateFileMapping(hFile, NULL, PAGE_WRITECOPY, 0, 0, NULL);

// Проецируем представление файла с атрибутом "копирование при записи";
// система передаст столько физической памяти из страничного файла,
// сколько нужно для размещения всего файла. Первоначально все страницы
// в представлении получат атрибут PAGE_WRITECOPY.
PBYTE pbFile = (PBYTE) MapViewOfFile(hFileMapping, FILE_MAP_COPY, 0, 0, 0);

// считываем байт из представления файла
BYTE bSomeByte = pbFile[0];
// при чтении система не трогает страницы, переданные из страничного файла;
// страница сохраняет свой атрибут PAGE_WRITECOPY

// записываем байт в представление файла
pbFile[0] = 0;
// При первой записи система берет страницу, переданную из страничного файла,
// копирует исходное содержимое страницы, расположенной по запрашиваемому адресу
// в памяти, и проецирует новую страницу (копию) на адресное пространство процесса.
// Новая страница получает атрибут PAGE_READWRITE.

// записываем еще один байт в представление файла
pbFile[1] = 0;
// поскольку теперь байт располагается на странице с атрибутом PAGE_READWRITE,
// система просто записывает его на эту страницу (она связана со страничным файлом)

// закончив работу с представлением проецируемого файла, прекращаем проецирование;
// функция UnMapViewOfFile обсуждается в следующем разделе
UnMapViewOfFile(pbFile);
// вся физическая память, взятая из страничного файла, возвращается системе;
// все, что было записано на эти страницы, теряется
```

```
// "уходя, гасите свет"
CloseHandle(hFileMapping);
CloseHandle(hFile);
```

WINDOWS 98 Как уже упоминалось, Windows 98 сначала передает проецируемому файлу физическую память из страничного файла. Однако запись модифицированных страниц в страничный файл происходит только при необходимости.

Этап 4: отключение файла данных от адресного пространства процесса

Когда необходимость в данных файла (спроектированного на регион адресного пространства процесса) отпадет, освободите регион вызовом:

```
BOOL UnmapViewOfFile(PVOID pvBaseAddress);
```

Ее единственный параметр, *pvBaseAddress*, указывает базовый адрес возвращаемого системе региона. Он должен совпадать со значением, полученным после вызова *MapViewOfFile*. Вы обязаны вызывать функцию *UnmapViewOfFile*. Если Вы не сделаете этого, регион не освободится до завершения Вашего процесса. И еще: повторный вызов *MapViewOfFile* приводит к резервированию нового региона в пределах адресного пространства процесса, но ранее выделенные регионы *не освобождаются*.

Для повышения производительности при работе с представлением файла система буферизует страницы данных в файле и не обновляет немедленно дисковый образ файла. При необходимости можно заставить систему записать измененные данные (все или частично) в дисковый образ файла, вызвав функцию *FlushViewOfFile*:

```
BOOL FlushViewOfFile(
    PVOID pvAddress,
    SIZE_T dwNumberofBytesToFlush);
```

Ее первый параметр принимает адрес байта, который содержится в границах представления файла, проецируемого в память. Переданный адрес округляется до значения, кратного размеру страниц. Второй параметр определяет количество байтов, которые надо записать в дисковый образ файла. Если *FlushViewOfFile* вызывается в отсутствие измененных данных, она просто возвращает управление.

В случае проецируемых файлов, физическая память которых расположена на сетевом диске, *FlushViewOfFile* гарантирует, что файловые данные будут перекачаны с рабочей станции. Но она не гарантирует, что сервер, обеспечивающий доступ к этому файлу, запишет данные на удаленный диск, так как он может просто кэшировать их. Для подстраховки при создании объекта «проекция файла» и последующем проектировании его представления используйте флаг *FILE_FLAG_WRITE_THROUGH*. При открытии файла с этим флагом функция *FlushViewOfFile* вернет управление только после сохранения на диске сервера всех файловых данных.

У функции *UnmapViewOfFile* есть одна особенность. Если первоначально представление было спроектировано с флагом *FILE_MAP_COPY*, любые изменения, внесенные Вами в файловые данные, на самом деле производятся над копией этих данных, хранящихся в страничном файле. Вызванной в этом случае функции *UnmapViewOfFile* нечего обновлять в дисковом файле, и она просто инициирует возврат системе страниц физической памяти, выделенных из страничного файла. Все изменения в данных на этих страницах теряются.

Поэтому о сохранении измененных данных придется заботиться самостоятельно. Например, для уже спроектированного файла можно создать еще один объект «про-

екция файла» с атрибутом PAGE_READWRITE и спроектировать его представление на адресное пространство процесса с флагом FILE_MAP_WRITE. Затем просмотреть первое представление, отыскивая страницы с атрибутом PAGE_READWRITE. Найдя страницу с таким атрибутом, Вы анализируете ее содержимое и решаете: записывать ее или нет. Если обновлять файл не нужно, Вы продолжаете просмотр страниц. А для сохранения страницы с измененными данными достаточно вызвать *MoveMemory* и скопировать страницу из первого представления файла во второе. Поскольку второе представление создано с атрибутом PAGE_READWRITE, функция *MoveMemory* обновит содержимое дискового файла. Так что этот метод вполне пригоден для анализа изменений и сохранения их в файле.

WINDOWS 98 Windows 98 не поддерживает атрибут защиты «копирование при записи», поэтому при просмотре первого представления файла, проецируемого в память, Вы не сможете проверить страницы по флагу PAGE_READWRITE. Вам придется разработать свой метод.

Этапы 5 и 6: закрытие объектов «проекция файла» и «файл»

Закончив работу с любым открытым Вами объектом ядра, Вы должны его закрыть, иначе в процессе начнется утечка ресурсов. Конечно, по завершении процесса система автоматически закроет объекты, оставленные открытыми. Но, если процесс поработает еще какое-то время, может накопиться слишком много незакрытых описателей. Поэтому старайтесь придерживаться правил хорошего тона и пишите код так, чтобы открытые объекты всегда закрывались, как только они станут не нужны. Для закрытия объектов «проекция файла» и «файл» дважды вызовите функцию *CloseHandle*.

Рассмотрим это подробнее на фрагменте псевдокода:

```
HANDLE hFile = CreateFile(...);
HANDLE hFileMapping = CreateFileMapping(hFile, ...);
PVOID pvFile = MapViewOfFile(hFileMapping, ...);

// работаем с файлом, спроектированным в память
UnmapViewOfFile(pvFile);
CloseHandle(hFileMapping);
CloseHandle(hFile);
```

Этот фрагмент иллюстрирует стандартный метод управления проецируемыми файлами. Но он не отражает того факта, что при вызове *MapViewOfFile* система увеличивает счетчики числа пользователей объектов «файл» и «проекция файла». Этот побочный эффект весьма важен, так как позволяет переписать показанный выше фрагмент кода следующим образом:

```
HANDLE hFile = CreateFile(...);
HANDLE hFileMapping = CreateFileMapping(hFile, ...);
CloseHandle(hFile);
PVOID pvFile = MapViewOfFile(hFileMapping, ...);
CloseHandle(hFileMapping);

// работаем с файлом, спроектированным в память
UnmapViewOfFile(pvFile);
```

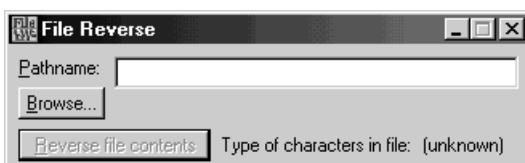
При операциях с проецируемыми файлами обычно открывают файл, создают объект «проекция файла» и с его помощью проецируют представление файловых

данных на адресное пространство процесса. Поскольку система увеличивает внутренние счетчики объектов «файл» и «проекция файла», их можно закрыть в начале кода, тем самым исключив возможную утечку ресурсов.

Если Вы будете создавать из одного файла несколько объектов «проекция файла» или проецировать несколько представлений этого объекта, применить функцию *CloseHandle* в начале кода не удастся — описатели еще понадобятся Вам для дополнительных вызовов *CreateFileMapping* и *MapViewOfFile*.

Программа-пример FileRev

Эта программа, «17 FileRev.exe» (см. листинг на рис. 17-2), демонстрирует, как с помощью механизма проецирования записать в обратном порядке содержимое текстового ANSI- или Unicode-файла. Файлы исходного кода и ресурсов этой программы находятся в каталоге 17-FileRev на компакт-диске, прилагаемом к книге. После запуска FileRev на экране появляется диалоговое окно, показанное ниже.



Выбрав имя файла и щелкнув кнопку Reverse File Contents, Вы активизируете функцию, которая меняет порядок символов в файле на обратный. Программа корректно работает только с текстовыми файлами. В какой кодировке создан текстовый файл (ANSI или Unicode), FileRev определяет вызовом *IsTextUnicode* (см. главу 2).

WINDOWS 98 В Windows 98 функция *IsTextUnicode* определена, но не реализована; она просто возвращает FALSE, а последующий вызов *GetLastError* дает ERROR_CALL_NOT_IMPLEMENTED. Это значит, что программа FileRev, выполняемая в Windows 98, всегда считает, что файл содержит текст в ANSI-кодировке.

После щелчка кнопки Reverse File Contents программа создает копию файла с именем FileRev.dat. Делается это для того, чтобы не испортить исходный файл, изменив порядок следования байтов на обратный. Далее программа вызывает функцию *FileReverse* — она меняет порядок байтов на обратный и после этого вызывает *CreateFile*, открывая FileRev.dat для чтения и записи.

Как я уже говорил, простейший способ «перевернуть» содержимое файла — вызвать функцию *_strrev* из библиотеки С. Но для этого последний символ в строке должен быть нулевой. И поскольку текстовые файлы не заканчиваются нулевым символом, программа FileRev подписывает его в конец файла. Для этого сначала вызывает функция *GetFileSize*:

```
dwFileSize = GetFileSize(hFile, NULL);
```

Теперь, вооружившись знанием длины файла, можно создать объект «проекция файла», вызвав *CreateFileMapping*. При этом размер объекта равен *dwFileSize* плюс размер «широкого» символа, чтобы учесть дополнительный нулевой символ в конце файла. Создав объект «проекция файла», программа проецирует на свое адресное пространство представление этого объекта. Переменная *pvFile* содержит значение, возвращенное функцией *MapViewOfFile*, и указывает на первый байт текстового файла.

Следующий шаг — запись нулевого символа в конец файла и реверсия строки:

```
PSTR pchANSI = (PSTR) pvFile;
pchANSI[dwFileSize / sizeof(CHAR)] = 0;

// "переворачиваем" содержимое файла
_strrev(pchANSI);
```

В текстовом файле каждая строка завершается символами возврата каретки ('\r') и перевода строки ('\n'). К сожалению, после вызова функции `_strrev` эти символы тоже меняются местами. Поэтому для загрузки преобразованного файла в текстовый редактор придется заменить все пары «\n\r» на исходные «\r\n». В программе этим занимается следующий цикл:

```
while (pchANSI != NULL) {
    // вхождение найдено...
    *pchANSI++ = '\r'; // заменяем '\n' на '\r'
    *pchANSI++ = '\n'; // заменяем '\r' на '\n'
    pchANSI = strchr(pchANSI, '\n'); // ищем следующее вхождение
}
```

Закончив обработку файла, программа прекращает отображение на адресное пространство представления объекта «проекция файла» и закрывает описатели всех объектов ядра. Кроме того, программа должна удалить нулевой символ, добавленный в конец файла (функция `_strrev` не меняет позицию этого символа). Если бы программа не убрала нулевой символ, то полученный файл оказался бы на 1 символ длиннее, и тогда повторный запуск программы `FileRev` не позволил бы вернуть этот файл в исходное состояние. Чтобы удалить концевой нулевой символ, надо спуститься на уровень ниже и воспользоваться функциями, предназначенными для работы непосредственно с файлами на диске.

Прежде всего установите указатель файла в требуемую позицию (в данном случае — в конец файла) и вызовите функцию `SetEndOfFile`:

```
SetFilePointer(hFile, dwFileSize, NULL, FILE_BEGIN);
SetEndOfFile(hFile);
```





Функцию *SetEndOfFile* нужно вызывать после отмены проецирования представления и закрытия объекта «проекция файла», иначе она вернет FALSE, а функция *GetLastError* — ERROR_USER_MAPPED_FILE. Данная ошибка означает, что операция перемещения указателя в конец файла невозможна, пока этот файл связан с объектом «проекция файла».

Последнее, что делает FileRev, — запускает экземпляр Notepad, чтобы Вы могли увидеть преобразованный файл. Вот как выглядит результат работы программы FileRev применительно к собственному файлу FileRev.cpp.

**File
Rev** FileRev.cpp

```
/*
Модуль: FileRev.cpp
Автор: Copyright (c) 2000, Джейфри Рихтер (Jeffrey Richter)
*/
#include "..\CmnHdr.h"      /* см. приложение A */
#include <windowsx.h>
#include <tchar.h>
#include <commddlg.h>
#include <string.h>          // для доступа к _strrev
#include "Resource.h"

////////// #define FILENAME TEXT("FILEREV.DAT")
//////////

BOOL FileReverse(PCTSTR pszPathname, PBOOL pfIsTextUnicode) {
    *pfIsTextUnicode = FALSE; // предполагаем, что текст в Unicode
    // открываем файл для чтения и записи
    HANDLE hFile = CreateFile(pszPathname, GENERIC_WRITE | GENERIC_READ, 0,
        NULL, OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, NULL);
    if (hFile == INVALID_HANDLE_VALUE) {
        chMB("File could not be opened.");
        return(FALSE);
    }
    // получаем размер файла (я предполагаю, что спроектировать можно весь файл)
    DWORD dwFileSize = GetFileSize(hFile, NULL);

    // Создаем объект "проекция файла". Он на 1 символ больше, чем сам файл, чтобы
    // можно было дописать нулевой символ для корректного завершения строки.
    // Поскольку пока еще неизвестно, содержит файл ANSI- или Unicode-символы,
    // я предполагаю худшее и добавляю размер WCHAR вместо CHAR.
}
```

Рис. 17-2. Программа-пример FileRev

см. след. стр.

Рис. 17-2. продолжение

```

HANDLE hFileMap = CreateFileMapping(hFile, NULL, PAGE_READWRITE,
0, dwFileSize + sizeof(WCHAR), NULL);

if (hFileMap == NULL) {
    chMB("File map could not be opened.");
    CloseHandle(hFile);
    return(FALSE);
}

// получаем адрес, по которому проецируется в память первый байт файла
PVOID pvFile = MapViewOfFile(hFileMap, FILE_MAP_WRITE, 0, 0, 0);

if (pvFile == NULL) {
    chMB("Could not map view of file.");
    CloseHandle(hFileMap);
    CloseHandle(hFile);
    return(FALSE);
}

// что содержит буфер: ANSI- или Unicode-символы?
int iUnicodeTestFlags = -1; // выполнить все проверки
*pfIsTextUnicode = IsTextUnicode(pvFile, dwFileSize, &iUnicodeTestFlags);

if (!*pfIsTextUnicode) {
    // при дальнейших операциях с файлом явно используем ANSI-функции,
    // так как мы имеем дело с ANSI-файлом

    // записываем в самый конец файла нулевой символ
    PSTR pchANSI = (PSTR) pvFile;
    pchANSI[dwFileSize / sizeof(CHAR)] = 0;

    // "переворачиваем" содержимое файла
    _strrev(pchANSI);

    // преобразуем все комбинации "\n\r" обратно в "\r\n", чтобы сохранить
    // нормальную последовательность кодов завершения строки в текстовом файле
    pchANSI = strchr(pchANSI, '\n'); // ищем первое вхождение '\n'

    while (pchANSI != NULL) {
        // вхождение найдено...
        *pchANSI++ = '\r'; // заменяем '\n' на '\r'
        *pchANSI++ = '\n'; // заменяем '\r' на '\n'
        pchANSI = strchr(pchANSI, '\n'); // ищем следующее вхождение
    }

} else {
    // при дальнейших операциях с файлом явно используем Unicode-функции,
    // так как мы имеем дело с Unicode-файлом

    // записываем в самый конец файла нулевой символ
    PWSTR pchUnicode = (PWSTR) pvFile;
    pchUnicode[dwFileSize / sizeof(WCHAR)] = 0;
}

```

Рис. 17-2. продолжение

```

if ((iUnicodeTestFlags & IS_TEXT_UNICODE_SIGNATURE) != 0) {
    // если первый символ - Unicode-маркер порядка байтов (0xFEFF),
    // то оставим этот символ в начале файла
    pchUnicode++;
}

// "переворачиваем" содержимое файла
_wcsrev(pchUnicode);

// преобразуем все комбинации "\n\r" обратно в "\r\n", чтобы сохранить
// нормальную последовательность кодов завершения строки в текстовом файле
pchUnicode = wcschr(pchUnicode, L'\n'); // ищем первое вхождение '\n'

while (pchUnicode != NULL) {
    // вхождение найдено...
    *pchUnicode++ = L'\r'; // заменяем '\n' на '\r'
    *pchUnicode++ = L'\n'; // заменяем '\r' на '\n'
    pchUnicode = wcschr(pchUnicode, L'\n'); // ищем следующее вхождение
}
}

// очищаем все перед завершением
UnmapViewOfFile(pvFile);
CloseHandle(hFileMap);

// удаляем добавленный ранее концевой нулевой байт
SetFilePointer(hFile, dwFileSize, NULL, FILE_BEGIN);
SetEndOfFile(hFile);
CloseHandle(hFile);

return(TRUE);
}

///////////////////////////////
BOOL Dlg_OnInitDialog(HWND hwnd, HWND hwndFocus, LPARAM lParam) {
    chSETDLGICONS(hwnd, IDI_FILEREV);

    // инициализируем диалоговое окно, отключая кнопку Reverse
    EnableWindow(GetDlgItem(hwnd, IDC_REVERSE), FALSE);
    return(TRUE);
}

/////////////////////////////
void Dlg_OnCommand(HWND hwnd, int id, HWND hwndCtl, UINT codeNotify) {
    TCHAR szPathname[MAX_PATH];

    switch (id) {

```

см. след. стр.

Рис. 17-2. продолжение

```

case IDCANCEL:
    EndDialog(hwnd, id);
    break;

case IDC_FILENAME:
    EnableWindow(GetDlgItem(hwnd, IDC_REVERSE),
        Edit_GetTextLength(hwndCtl) > 0);
    break;

case IDC_REVERSE:
    GetDlgItemText(hwnd, IDC_FILENAME, szPathname, chDIMOF(szPathname));

    // делаем копию исходного файла, чтобы случайно не повредить его
    if (!CopyFile(szPathname, FILENAME, FALSE)) {
        chMB("New file could not be created.");
        break;
    }

    BOOL fIsUnicode;
    if (FileReverse(FILENAME, &fIsUnicode)) {
        SetDlgItemText(hwnd, IDC_TEXTPAGE,
            fIsUnicode ? TEXT("Unicode") : TEXT("ANSI"));

        // запускаем Notepad, чтобы увидеть плоды своих трудов
        STARTUPINFO si = { sizeof(si) };
        PROCESS_INFORMATION pi;
        TCHAR sz[] = TEXT("Notepad ") FILENAME;
        if (CreateProcess(NULL, sz,
            NULL, NULL, FALSE, 0, NULL, NULL, &si, &pi)) {
            CloseHandle(pi.hThread);
            CloseHandle(pi.hProcess);
        }
    }
    break;

case IDC_FILESELECT:
    OPENFILENAME ofn = { OPENFILENAME_SIZE_VERSION_400 };
    ofn.hwndOwner = hwnd;
    ofn.lpstrFile = szPathname;
    ofn.lpstrFile[0] = 0;
    ofn.nMaxFile = chDIMOF(szPathname);
    ofn.lpstrTitle = TEXT("Select file for reversing");
    ofn.Flags = OFN_EXPLORER | OFN_FILEMUSTEXIST;
    GetOpenFileName(&ofn);
    SetDlgItemText(hwnd, IDC_FILENAME, ofn.lpstrFile);
    SetFocus(GetDlgItem(hwnd, IDC_REVERSE));
    break;
}
/////////////////////////////////////////////////////////////////

```

Рис. 17-2. продолжение

```

INT_PTR WINAPI Dlg_Proc(HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam) {

    switch (uMsg) {
        chHANDLE_DLGMMSG(hwnd, WM_INITDIALOG, Dlg_OnInitDialog);
        chHANDLE_DLGMMSG(hwnd, WM_COMMAND, Dlg_OnCommand);
    }
    return(FALSE);
}

/////////////////////////////// Конец файла ///////////////////////////////
int WINAPI _tWinMain(HINSTANCE hinstExe, HINSTANCE, PTSTR pszCmdLine, int) {

    DialogBox(hinstExe, MAKEINTRESOURCE(IDD_FILEREV), NULL, Dlg_Proc);
    return(0);
}

/////////////////////////////// Конец файла ///////////////////////////////

```

Обработка больших файлов

Я обещал рассказать, как спроектировать на небольшое адресное пространство файл длиной 16 экзабайтов. Так вот, этого сделать нельзя. Вам придется проектировать не весь файл, а его представление, содержащее лишь некую часть данных. Вы начнете с того, что спроектируете представление самого начала файла. Закончив обработку данных в этом представлении, Вы отключите его и спроектируете представление следующей части файла — и так до тех пор, пока не будет обработан весь файл. Конечно, это делает работу с большими файлами, проектируемыми в память, не слишком удобной, но утешимся тем, что длина большинства файлов достаточно мала.

Рассмотрим сказанное на примере файла размером 8 Гб. Ниже приведен текст подпрограммы, позволяющей в несколько этапов подсчитывать, сколько раз встречается нулевой байт в том или ином двоичном файле данных.

```

__int64 Count0s(void) {
    // начальные границы представлений всегда начинаются по адресам,
    // кратным гранулярности выделения памяти
    SYSTEM_INFO sinfo;
    GetSystemInfo(&sinfo);

    // открываем файл данных
    HANDLE hFile = CreateFile("C:\\\\HugeFile.Big", GENERIC_READ,
        FILE_SHARE_READ, NULL, OPEN_EXISTING, FILE_FLAG_SEQUENTIAL_SCAN, NULL);
    // создаем объект "проекция файла"
    HANDLE hFileMapping = CreateFileMapping(hFile, NULL, PAGE_READONLY, 0, 0, NULL);
    DWORD dwFileSizeHigh;
    __int64 qwFileSize = GetFileSize(hFile, &dwFileSizeHigh);
    qwFileSize += (((__int64) dwFileSizeHigh) << 32);

    // доступ к описателю объекта "файл" нам больше не нужен
    CloseHandle(hFile);

```

см. след. стр.

```
__int64 qwFileOffset = 0, qwNumOf0s = 0;
while (qwFileSize > 0) {
    // определяем, сколько байтов надо спроектировать
    DWORD dwBytesInBlock = sinf.dwAllocationGranularity;
    if (qwFileSize < sinf.dwAllocationGranularity)
        dwBytesInBlock = (DWORD) qwFileSize;

    PBYTE pbFile = (PBYTE) MapViewOfFile(hFileMapping, FILE_MAP_READ,
        (DWORD) (qwFileOffset >> 32),           // начальный байт
        (DWORD) (qwFileOffset & 0xFFFFFFFF),     // в файле
        dwBytesInBlock);                      // число проецируемых байтов

    // подсчитываем количество нулевых байтов в этом блоке
    for (DWORD dwByte = 0; dwByte < dwBytesInBlock; dwByte++) {
        if (pbFile[dwByte] == 0)
            qwNumOf0s++;
    }
    // прекращаем проектирование представления, чтобы в адресном пространстве
    // не образовалось несколько представлений одного файла
    UnmapViewOfFile(pbFile);

    // переходим к следующей группе байтов в файле
    qwFileOffset += dwBytesInBlock;
    qwFileSize -= dwBytesInBlock;
}
CloseHandle(hFileMapping);
return(qwNumOf0s);
}
```

Этот алгоритм проектирует представления по 64 Кб (в соответствии с гранулярностью выделения памяти) или менее. Кроме того, функция *MapViewOfFile* требует, чтобы передаваемое ей смещение в файле тоже было кратно гранулярности выделения памяти. Подпрограмма проектирует на адресное пространство сначала одно представление, подсчитывает в нем количество нулей, затем переходит к другому представлению, и все повторяется. Спроектировав и просмотрев все 64-килобайтовые блоки, подпрограмма закрывает объект «проекция файла».

Проектируемые файлы и когерентность

Система позволяет проектировать сразу несколько представлений одних и тех же файловых данных. Например, можно спроектировать в одно представление первые 10 Кб файла, а затем — первые 4 Кб того же файла в другое представление. Пока Вы проектируете один и тот же объект, система гарантирует *когерентность* (согласованность) отображаемых данных. Скажем, если программа изменяет содержимое файла в одном представлении, это приводит к обновлению данных и в другом. Так происходит потому, что система, несмотря на многократную проекцию страницы на виртуальное адресное пространство процесса, хранит данные на единственной странице оперативной памяти. Поэтому, если представления одного и того же файла данных создаются сразу несколькими процессами, данные по-прежнему сохраняют когерентность — ведь они сопоставлены только с одним экземпляром каждой страницы в оперативной памяти. Все это равносильно тому, как если бы страницы оперативной памяти были спроектированы на адресные пространства нескольких процессов одновременно.



Windows позволяет создавать несколько объектов «проекция файла», связанных с одним и тем же файлом данных. Но тогда у Вас *не будет* гарантий, что содержимое представлений этих объектов когерентно. Такую гарантию Windows дает только для нескольких представлений одного объекта «проекция файла».

Кстати, функция *CreateFile* позволяет Вашему процессу открывать файл, проецируемый в память другим процессом. После этого Ваш процесс сможет считывать или записывать данные в файл (с помощью функций *ReadFile* или *WriteFile*). Разумеется, при вызовах упомянутых функций Ваш процесс будет считывать или записывать данные не в файл, а в некий буфер памяти, который должен быть создан именно этим процессом; буфер не имеет никакого отношения к участку памяти, используемому для проецирования данного файла. Но надо учитывать, что, когда два приложения открывают один файл, могут возникнуть проблемы. Дело в том, что один процесс может вызвать *ReadFile*, считать фрагмент файла, модифицировать данные и записать их обратно в файл с помощью *WriteFile*, а объект «проекция файла», принадлежащий второму процессу, ничего об этом не узнает. Поэтому, вызывая для проецируемого файла функцию *CreateFile*, всегда указывайте нуль в параметре *dwShareMode*. Тем самым Вы сообщите системе, что Вам нужен монопольный доступ к файлу и никакой посторонний процесс не должен его открывать.

Файлы с доступом «только для чтения» не вызывают проблем с когерентностью — значит, это лучшие кандидаты на отображение в память. Ни в коем случае не используйте механизм проецирования для доступа к записываемым файлам, размещенным на сетевых дисках, так как система не сможет гарантировать когерентность представлений данных. Если один компьютер обновит содержимое файла, то другой, у которого исходные данные содержатся в памяти, не узнает об изменении информации.

Базовый адрес файла, проецируемого в память

Помните, как Вы с помощью функции *VirtualAlloc* указывали базовый адрес региона, резервируемого в адресном пространстве? Примерно так же можно указать системе спроектировать файл по определенному адресу — только вместо функции *MapViewOfFile* нужна *MapViewOfFileEx*:

```
VOID MapViewOfFileEx(
    HANDLE hFileMappingObject,
    DWORD dwDesiredAccess,
    DWORD dwFileOffsetHigh,
    DWORD dwFileOffsetLow,
    SIZE_T dwNumberOfBytesToMap,
    PVOID pvBaseAddress);
```

Все параметры и возвращаемое этой функцией значение идентичны применяемым в *MapViewOfFile*, кроме последнего параметра — *pvBaseAddress*. В нем можно задать начальный адрес файла, проецируемого в память. Как и в случае *VirtualAlloc*, базовый адрес должен быть кратным гранулярности выделения памяти в системе (обычно 64 Кб), иначе *MapViewOfFileEx* вернет NULL, сообщив тем самым об ошибке.

Если Вы укажете базовый адрес, не кратный гранулярности выделения памяти, то *MapViewOfFileEx* в Windows 2000 завершится с ошибкой, и *GetLastError* вернет код 1132 (ERROR_MAPPED_ALIGNMENT), а в Windows 98 базовый адрес будет округлен до ближайшего меньшего значения, кратного гранулярности выделения памяти.

Если система не в состоянии спроектировать файл по этому адресу (чаще всего из-за того, что файл слишком велик и мог бы перекрыть другие регионы зарезервированного адресного пространства), функция также возвращает NULL. В этом случае она не пытается подобрать диапазон адресов, подходящий для данного файла. Но если Вы укажете NULL в параметре *pvBaseAddress*, она поведет себя идентично *MapViewOfFile*.

MapViewOfFileEx удобна, когда механизм проектирования файлов в память применяется для совместного доступа нескольких процессов к одним данным. Поясню. Допустим, нужно спроектировать файл в память по определенному адресу; при этом два или более приложений совместно используют одну группу структур данных, содержащих указатели на другие структуры данных. Отличный тому пример — связанный список. Каждый узел, или элемент, такого списка хранит адрес другого узла списка. Для просмотра списка надо узнать адрес первого узла, а затем сделать ссылку на то его поле, где содержится адрес следующего узла. Но при использовании файлов, проектируемых в память, это весьма проблематично.

Если один процесс подготовил в проектируемом файле связанный список, а затем разделил его с другим процессом, не исключено, что второй процесс спроектирует этот файл в своем адресном пространстве на совершенно иной регион. А дальше будет вот что. Попытавшись просмотреть связанный список, второй процесс проверит первый узел списка, прочитает адрес следующего узла и, сделав на него ссылку, получит совсем не то, что ему было нужно, — адрес следующего элемента в первом узле некорректен для второго процесса.

У этой проблемы два решения. Во-первых, второй процесс, проектируя файл со связанным списком на свое адресное пространство, может вызвать *MapViewOfFileEx* вместо *MapViewOfFile*. Для этого второй процесс должен знать адрес, по которому файл спроектирован на адресное пространство первого процесса на момент создания списка. Если оба приложения разработаны с учетом взаимодействия друг с другом (а так чаще всего и делают), нужный адрес может быть просто заложен в код этих программ или же один процесс как-то уведомляет другой (скажем, посылкой сообщения в окно).

А можно и так. Процесс, создающий связанный список, должен записывать в каждый узел смещение следующего узла в пределах адресного пространства. Тогда программа, чтобы получить доступ к каждому узлу, будет суммировать это смещение с базовым адресом проектируемого файла. Несмотря на простоту, этот способ не лучший: дополнительные операции замедлят работу программы и увеличат объем ее кода (так как компилятор для выполнения всех вычислений, естественно, генерирует дополнительный код). Кроме того, при этом способе вероятность ошибок значительно выше. Тем не менее он имеет право на существование, и поэтому компиляторы Microsoft поддерживают указатели со смещением относительно базового значения (*based-pointers*), для чего предусмотрено ключевое слово *_based*.

WINDOWS 98 В Windows 98 при вызове *MapViewOfFileEx* следует указывать адрес в диапазоне от 0x80000000 до 0xBFFFFFFF, иначе функция вернет NULL.

WINDOWS 2000 В Windows 2000 при вызове *MapViewOfFileEx* следует указывать адрес в границах пользовательского раздела адресного пространства процесса, иначе функция вернет NULL.

Особенности проецирования файлов на разных платформах

Механизм проецирования файлов в Windows 2000 и Windows 98 реализован по-разному. Вы должны знать об этих отличиях, поскольку они могут повлиять на код программ и целостность используемых ими данных.

В Windows 98 представление всегда проецируется на раздел адресного пространства, расположенный в диапазоне от 0x80000000 до 0xBFFFFFFF. Значит, после успешного вызова функция *MapViewOfFile* вернет какой-нибудь адрес из этого диапазона. Но вспомните: данные в этом разделе доступны всем процессам. Если один из процессов отображает сюда представление объекта «проекция файла», то принадлежащие этому объекту данные физически доступны всем процессам, и уже неважно: проецируют ли они сами представление того же объекта. Если другой процесс вызывает *MapViewOfFile*, используя тот же объект «проекция файла», Windows 98 возвращает адрес памяти, идентичный тому, что она сообщила первому процессу. Поэтому два процесса обращаются к одним и тем же данным и представления их объектов когерентны.

В Windows 98 один процесс может вызвать *MapViewOfFile* и, воспользовавшись какой-либо формой межпроцессной связи, передать возвращенный ею адрес памяти потоку другого процесса. Как только этот поток получит нужный адрес, ему уже ничего не помешает получить доступ к тому же представлению объекта «проекция файла». Но прибегать к такой возможности не следует по двум причинам:

- приложение не будет работать в Windows 2000 (и я только что рассказал — почему);
- если первый процесс вызовет *UnMapViewOfFile*, регион адресного пространства освободится. А значит, при попытке потока второго процесса обратиться к участку памяти, где когда-то находилось представление, возникнет нарушение доступа.

Чтобы второй процесс получил доступ к представлению проецируемого файла, его поток тоже должен вызвать *MapViewOfFile*. Тогда система увеличит счетчик числа пользователей объекта «проекция файла». И если первый процесс обратится к *UnMapViewOfFile*, регион адресного пространства, занятый представлением, не будет освобожден, пока второй процесс тоже не вызовет *UnMapViewOfFile*. А вызвав *MapViewOfFile*, второй процесс получит тот же адрес, что и первый. Таким образом, необходимость в передаче адреса от первого процесса второму отпадает.

В Windows 2000 механизм проецирования файлов реализован удачнее, чем в Windows 98, потому что Windows 2000 для доступа к файловым данным в адресном пространстве требует вызова *MapViewOfFile*. При обращении к этой функции система резервирует для проецируемого файла закрытый регион адресного пространства, и никакой другой процесс не получает к нему доступ автоматически. Чтобы посторонний процесс мог обратиться к данным того же объекта «проекция файла», его поток тоже должен вызвать *MapViewOfFile*, и система отведет регион для представления объекта в адресном пространстве второго процесса.

Адрес, полученный при вызове *MapViewOfFile* первым процессом, скорее всего не совпадет с тем, что получит при ее вызове второй процесс, — даже несмотря на то что оба процесса проецируют представление одного и того же объекта. И хотя в Windows 98 адреса, получаемые процессами при вызове *MapViewOfFile*, совпадают,

лучше не полагаться на эту особенность — иначе приложение не станет работать в Windows 2000!

Рассмотрим еще одно различие механизмов проецирования файлов у Windows 2000 и Windows 98. Взгляните на текст программы, проецирующей два представления единственного объекта «проекция файла».

```
#include <Windows.h>

int WINAPI WinMain(HINSTANCE hinstExe, HINSTANCE,
PTSTR pszCmdLine, int nCmdShow) {

    // открываем существующий файл; он должен быть больше 64 Кб
    HANDLE hFile = CreateFile(pszCmdLine, GENERIC_READ | GENERIC_WRITE, 0,
        NULL, OPEN_ALWAYS, FILE_ATTRIBUTE_NORMAL, NULL);

    // создаем объект "проекция файла", связанный с файлом данных
    HANDLE hFileMapping = CreateFileMapping(hFile, NULL, PAGE_READWRITE, 0, 0, NULL);

    // проецируем представление всего файла на наше адресное пространство
    PBYTE pbFile = (PBYTE) MapViewOfFile(hFileMapping, FILE_MAP_WRITE, 0, 0, 0);

    // проецируем второе представление файла, начиная со смещения 64 Кб
    PBYTE pbFile2 = (PBYTE) MapViewOfFile(hFileMapping, FILE_MAP_WRITE, 0, 65536, 0);

    if ((pbFile + 65536) == pbFile2) {
        // если адреса перекрываются, оба представления проецируются на один
        // регион, и мы работаем в Windows 98
        MessageBox(NULL, "We are running under Windows 98", NULL, MB_OK);
    } else {
        // если адреса не перекрываются, каждое представление размещается в
        // своем регионе адресного пространства, и мы работаем в Windows 2000
        MessageBox(NULL, "We are running under Windows 2000", NULL, MB_OK);
    }
    UnmapViewOfFile(pbFile2);
    UnmapViewOfFile(pbFile);
    CloseHandle(hFileMapping);
    CloseHandle(hFile);

    return(0);
}
```

Когда приложение в Windows 98 отображает на адресное пространство представление объекта «проекция файла», ему отводится регион, достаточно большой для размещения всего объекта. Это происходит, даже если Вы просите *MapViewOfFile* спроектировать лишь малую часть такого объекта. Поэтому спроектировать объект размером 1 Гб не удастся, даже если указать, что представление должно быть не более 64 Кб.

При вызове каким-либо процессом функции *MapViewOfFile* ему возвращается адрес в пределах региона, зарезервированного для целого объекта «проекция файла». Так что в показанной выше программе первый вызов этой функции дает базовый адрес региона, содержащего весь спроектированный файл, а второй — адрес, смещенный «вглубь» того же региона на 64 Кб.

Windows 2000 и здесь ведет себя совершенно иначе. Два вызова функции *MapViewOfFile* (как в показанном выше коде) приведут к тому, что будут зарезервированы два

региона адресного пространства. Объем первого будет равен размеру объекта «проекция файла», объем второго — размеру объекта минус 64 Кб. Хотя регионы — разные, система гарантирует когерентность данных, так как оба представления созданы на основе одного объекта «проекция файла». А в Windows 98 такие представления когерентны потому, что они расположены в одном участке памяти.

Совместный доступ процессов к данным через механизм проецирования

В Windows всегда было много механизмов, позволяющих приложениям легко и быстро разделять какие-либо данные. К этим механизмам относятся RPC, COM, OLE, DDE, оконные сообщения (особенно WM_COPYDATA), буфер обмена, почтовые ящики, сокеты и т. д. Самый низкоуровневый механизм совместного использования данных на одной машине — проецирование файла в память. На нем так или иначе базируются все перечисленные мной механизмы разделения данных. Поэтому, если Вас интересует максимальное быстродействие с минимумом издержек, лучше всего применять именно проецирование.

Совместное использование данных в этом случае происходит так: два или более процесса проецируют в память представления одного и того же объекта «проекция файла», т. е. делят одни и те же страницы физической памяти. В результате, когда один процесс записывает данные в представление общего объекта «проекция файла», изменения немедленно отражаются на представлениях в других процессах. Но при этом все процессы должны использовать одинаковое имя объекта «проекция файла».

А вот что происходит при запуске приложения. При открытии EXE-файла на диске система вызывает *CreateFile*, с помощью *CreateFileMapping* создает объект «проекция файла» и, наконец, вызывает *MapViewOfFileEx* (с флагом SEC_IMAGE) для отображения EXE-файла на адресное пространство только что созданного процесса. *MapViewOfFileEx* вызывается вместо *MapViewOfFile*, чтобы представление файла было спроектировано по базовому адресу, значение которого хранится в самом EXE-файле. Потом создается первичный поток процесса, адрес первого байта исполняемого кода в спроектированном представлении заносится в регистр указателя команд (IP), и процессор приступает к исполнению кода.

Если пользователь запустит второй экземпляр того же приложения, система увидит, что объект «проекция файла» для нужного EXE-файла уже существует и не станет создавать новый объект. Она просто спроектирует еще одно представление файла — на этот раз в контексте адресного пространства только что созданного второго процесса, т. е. одновременно спроектирует один и тот же файл на два адресных пространства. Это позволяет эффективнее использовать память, так как оба процесса делят одни и те же страницы физической памяти, содержащие порции исполняемого кода.

Как и все объекты ядра, проекции файлов можно совместно использовать из нескольких процессов тремя методами: наследованием описателей, именованием и дублированием описателей. Подробное объяснение этих трех методов см. в главе 3.

Файлы, проецируемые на физическую память из страничного файла

До сих пор мы говорили о методах, позволяющих проецировать представление файла, размещенного на диске. В то же время многие программы при выполнении создают данные, которые им нужно разделять с другими процессами. А создавать файл на диске и хранить там данные только с этой целью очень неудобно.

Прекрасно понимая это, Microsoft добавила возможность проецирования файлов непосредственно на физическую память из страничного файла, а не из специально создаваемого дискового файла. Этот способ даже проще стандартного — основанного на создании дискового файла, проецируемого в память. Во-первых, не надо вызывать *CreateFile*, так как создавать или открывать специальный файл не требуется. Вы просто вызываете, как обычно, *CreateFileMapping* и передаете *INVALID_HANDLE_VALUE* в параметре *bFile*. Тем самым Вы указываете системе, что создавать объект «проекция файла», физическая память которого находится на диске, не надо; вместо этого следует выделить физическую память из страничного файла. Объем выделяемой памяти определяется параметрами *dwMaximumSizeHigh* и *dwMaximumSizeLow*.

Создав объект «проекция файла» и спроектировав его представление на адресное пространство своего процесса, его можно использовать так же, как и любой другой регион памяти. Если Вы хотите, чтобы данные стали доступны другим процессам, вызовите *CreateFileMapping* и передайте в параметре *pszName* строку с нулевым символом в конце. Тогда посторонние процессы — если им понадобится сюда доступ — смогут вызвать *CreateFileMapping* или *OpenFileMapping* и передать ей то же имя.

Когда необходимость в доступе к объекту «проекция файла» отпадет, процесс должен вызвать *CloseHandle*. Как только все описатели объекта будут закрыты, система освободит память, переданную из страничного файла.



Есть одна интересная ловушка, в которую может попасть неискушенный программист. Попробуйте догадаться, что неверно в этом фрагменте кода:

```
HANDLE hFile = CreateFile(...);
HANDLE hMap = CreateFileMapping(hFile, ...);
if (hMap == NULL)
    return(GetLastError());
:
```

Если вызов *CreateFile* не удастся, она вернет *INVALID_HANDLE_VALUE*. Но программист, написавший этот код, не дополнил его проверкой на успешное создание файла. Поэтому, когда в дальнейшем код обращается к функции *CreateFileMapping*, в параметре *bFile* ей передается *INVALID_HANDLE_VALUE*, что заставляет систему создать объект «проекция файла» из ресурсов страничного файла, а не из дискового файла, как предполагалось в программе. Весь последующий код, который использует проецируемый файл, будет работать правильно. Но при уничтожении объекта «проекция файла» все данные, записанные в спроектированную память (страничный файл), пропадут. И разработчик будет долго чесать затылок, пытаясь понять, в чем дело!

Программа-пример MMFShare

Эта программа, «17 MMFShare.exe» (см. листинг на рис. 17-3), демонстрирует, как происходит обмен данными между двумя и более процессами с помощью файлов, проецируемых в память. Файлы исходного кода и ресурсов этой программы находятся в каталоге 17-MMFShare на компакт-диске, прилагаемом к книге.

Чтобы наблюдать за происходящим, нужно запустить минимум две копии MMFShare. Каждый экземпляр программы создаст свое диалоговое окно.

Чтобы переслать данные из одной копии MMFShare в другую, наберите какой-нибудь текст в поле Data. Затем щелкните кнопку Create Mapping Of Data. Программа вызовет функцию *CreateFileMapping*, чтобы создать объект «проекция файла» размером 4 Кб и присвоить ему имя *MMFSharedData* (ресурсы выделяются объекту из стра-

ничного файла). Увидев, что объект с таким именем уже существует, программа выдаст сообщение, что не может создать объект. А если такого объекта нет, программа создаст объект, спроектирует представление файла на адресное пространство процесса и скопирует данные из поля Data в проецируемый файл.



Далее MMFShare прекратит проецировать представление файла, отключит кнопку Create Mapping Of Data и активизирует кнопку Close Mapping Of Data. На этот момент проецируемый в память файл с именем *MMFSharedData* будет просто «сидеть» где-то в системе. Никакие процессы пока не проецируют представление на данные, содержащиеся в файле.

Если Вы теперь перейдете в другую копию MMFShare и щелкнете там кнопку Open Mapping And Get Data, программа попытается найти объект «проекция файла» с именем *MMFSharedData* через функцию *OpenFileMapping*. Если ей не удастся найти объект с таким именем, программа выдаст соответствующее сообщение. В ином случае она спроектирует представление объекта на адресное пространство своего процесса и скопирует данные из проецируемого файла в поле Data. Вот и все! Вы переслали данные из одного процесса в другой.

Кнопка Close Mapping Of Data служит для закрытия объекта «проекция файла», что высвобождает физическую память, занимаемую им в страничном файле. Если же объект «проекция файла» не существует, никакой другой экземпляр программы MMFShare не сможет открыть этот объект и получить от него данные. Кроме того, если один экземпляр программы создал объект «проекция файла», то остальным повторить его создание и тем самым перезаписать данные, содержащиеся в файле, уже не удастся.

```


MMFShare.cpp

/*
Модуль: MMFShare.cpp
Автор: Copyright (c) 2000, Джейфри Рихтер (Jeffrey Richter)
*/

#include "..\CmnHdr.h"      /* см. приложение A */
#include <windowsx.h>
#include <tchar.h>
#include "Resource.h"

///////////////////////////////
BOOL Dlg_OnInitDialog(HWND hwnd, HWND hwndFocus, LPARAM lParam) {
    chSETDLGICONS(hwnd, IDI_MMFSHARE);
    // инициализируем поле ввода тестовыми данными
}

```

Рис. 17-3. Программа-пример MMFShare

см. след. стр.

Рис. 17-3. продолжение

```

Edit_SetText(GetDlgItem(hwnd, IDC_DATA), TEXT("Some test data"));

// отключаем кнопку Close, так как файл нельзя закрыть,
// если он не создан или не открыт
Button_Enable(GetDlgItem(hwnd, IDC_CLOSEFILE), FALSE);
return(TRUE);
}

///////////////////////////////



void Dlg_OnCommand(HWND hwnd, int id, HWND hwndCtl, UINT codeNotify) {

// описатель открытого файла, проецируемого в память
static HANDLE s_hFileMap = NULL;

switch (id) {
    case IDCANCEL:
        EndDialog(hwnd, id);
        break;

    case IDC_CREATEFILE:
        if (codeNotify != BN_CLICKED)
            break;

        // создаем в памяти проецируемый файл с данными, набранными
        // в поле ввода; он занимает 4 Кб и называется MMFSharedData
        // (память выделяется из страничного файла)
        s_hFileMap = CreateFileMapping(INVALID_HANDLE_VALUE, NULL,
            PAGE_READWRITE, 0, 4 * 1024, TEXT("MMFSharedData"));

        if (s_hFileMap != NULL) {

            if (GetLastError() == ERROR_ALREADY_EXISTS) {
                chMB("Mapping already exists - not created.");
                CloseHandle(s_hFileMap);

            } else {

                // создание проецируемого файла завершилось успешно;
                // проецируем представление файла на адресное пространство
                PVOID pView = MapViewOfFile(s_hFileMap,
                    FILE_MAP_READ | FILE_MAP_WRITE, 0, 0, 0);

                if (pView != NULL) {
                    // поместим содержимое поля ввода в проецируемый файл
                    Edit_GetText(GetDlgItem(hwnd, IDC_DATA),
                        (LPTSTR) pView, 4 * 1024);

                    // прекращаем проецирование; это защитит
                    // данные от "блуждающих" указателей
                    UnmapViewOfFile(pView);

                }
            }
        }
}

```

Рис. 17-3. продолжение

```

        // пользователь не может создать сейчас еще один файл
        Button_Enable(hwndCtl, FALSE);

        // пользователь закрыл файл
        Button_Enable(GetDlgItem(hwnd, IDC_CLOSEFILE), TRUE);

    } else {
        chMB("Can't map view of file.");
    }
}

} else {
    chMB("Can't create file mapping.");
}
break;

case IDC_CLOSEFILE:
if (codeNotify != BN_CLICKED)
    break;

if (CloseHandle(s_hFileMap)) {
    // пользователь закрыл файл; новый файл создать можно,
    // но закрыть его нельзя
    Button_Enable(GetDlgItem(hwnd, IDC_CREATEFILE), TRUE);
    Button_Enable(hwndCtl, FALSE);
}
break;

case IDC_OPENFILE:
if (codeNotify != BN_CLICKED)
    break;

// смотрим: не существует ли проецируемый в память файл
// с именем MMFSharedData
HANDLE hFileMapT = OpenFileMapping(FILE_MAP_READ | FILE_MAP_WRITE,
    FALSE, TEXT("MMFSharedData"));

if (hFileMapT != NULL) {
    // такой файл есть; проецируем его представление
    // на адресное пространство процесса
    PVOID pView = MapViewOfFile(hFileMapT,
        FILE_MAP_READ | FILE_MAP_WRITE, 0, 0, 0);

    if (pView != NULL) {

        // помещаем содержимое файла в поле ввода
        Edit_SetText(GetDlgItem(hwnd, IDC_DATA), (LPTSTR) pView);
        UnmapViewOfFile(pView);
    } else {
        chMB("Can't map view.");
    }
}

```

см. след. стр.

Рис. 17-3. продолжение

```

        CloseHandle(hFileMapT);

    } else {
        chMB("Can't open mapping.");
    }
    break;
}
}

/////////////////////////////// Конец файла ///////////////////////////////
INT_PTR WINAPI Dlg_Proc(HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam) {

switch (uMsg) {
    chHANDLE_DLGMMSG(hwnd, WM_INITDIALOG, Dlg_OnInitDialog);
    chHANDLE_DLGMMSG(hwnd, WM_COMMAND, Dlg_OnCommand);
}
return(FALSE);
}

/////////////////////////////// Конец файла ///////////////////////////////
int WINAPI _tWinMain(HINSTANCE hinstExe, HINSTANCE, PTSTR pszCmdLine, int) {

DialogBox(hinstExe, MAKEINTRESOURCE(IDD_MMFSHARE), NULL, Dlg_Proc);
return(0);
}

/////////////////////////////// Конец файла ///////////////////////////////

```

Частичная передача физической памяти проецируемым файлам

До сих пор мы видели, что система требует передавать проецируемым файлам всю физическую память либо из файла данных на диске, либо из страничного файла. Это значит, что память используется не очень эффективно. Давайте вспомним то, что я говорил в разделе «В какой момент региону передают физическую память» главы 15. Допустим, Вы хотите сделать всю таблицу доступной другому процессу. Если применить для этого механизм проецирования файлов, придется передать физическую память целой таблице:

```
CELLDATA CellData[200][256];
```

Если структура CELldata занимает 128 байтов, показанный массив потребует 6 553 600 ($200 \times 256 \times 128$) байтов физической памяти. Это слишком много — тем более, что в таблице обычно заполняют всего несколько строк.

Очевидно, что в данном случае, создав объект «проекция файла», желательно не передавать ему заранее всю физическую память. Функция *CreateFileMapping* предусматривает такую возможность, для чего в параметр *fdwProtect* нужно передать один из флагов: *SEC_RESERVED* или *SEC_COMMIT*.

Эти флаги имеют смысл, только если Вы создаете объект «проекция файла», использующий физическую память из страничного файла. Флаг SEC_COMMIT заставляет *CreateFileMapping* сразу же передать память из страничного файла. (То же самое происходит, если никаких флагов не указано.) Но когда Вы задаете флаг SEC_RESERVE, система не передает физическую память из страничного файла, а просто возвращает описатель объекта «проекция файла». Далее, вызвав *MapViewOfFile* или *MapViewOfFileEx*, можно создать представление этого объекта. При этом *MapViewOfFile* или *MapViewOfFileEx* резервирует регион адресного пространства, не передавая ему физической памяти. Любая попытка обращения по одному из адресов зарезервированного региона приведет к нарушению доступа.

Таким образом, мы имеем регион зарезервированного адресного пространства и описатель объекта «проекция файла», идентифицирующий этот регион. Другие процессы могут использовать данный объект для проецирования представления того же региона адресного пространства. Физическая память региону по-прежнему не передается, так что, если потоки в других процессах попытаются обратиться по одному из адресов представления в своих регионах, они тоже вызовут нарушение доступа.

А теперь самое интересное. Оказывается, все, что нужно для передачи физической памяти общему (совместно используемому) региону, — вызвать функцию *VirtualAlloc*:

```
VOID* VirtualAlloc(
    PVOID pvAddress,
    SIZE_T dwSize,
    DWORD fdwAllocationType,
    DWORD fdwProtect);
```

Эту функцию мы уже рассматривали (и очень подробно) в главе 15. Вызвать *VirtualAlloc* для передачи физической памяти представлению региона — то же самое, что вызвать *VirtualAlloc* для передачи памяти региону, ранее зарезервированному вызовом *VirtualAlloc* с флагом MEM_RESERVE. Получается, что региону, зарезервированному функциями *MapViewOfFile* или *MapViewOfFileEx*, — как и региону, зарезервированному функцией *VirtualAlloc*, — тоже можно передавать физическую память порциями, а не всю сразу. И если Вы поступаете именно так, учтите, что все процессы, спроектировавшие на этот регион представление одного и того же объекта «проекция файла», теперь тоже получат доступ к страницам физической памяти, переданным региону.

Итак, флаг SEC_RESERVE и функция *VirtualAlloc* позволяют сделать табличную матрицу *CellData* «общедоступной» и эффективнее использовать память.

WINDOWS 98 Обычно *VirtualAlloc* не срабатывает, если Вы передаете ей адрес памяти, выходящий за пределы диапазона от 0x00400000 до 0x7FFFFFFF. Однако при передаче физической памяти проецируемому файлу, созданному с флагом SEC_RESERVED, в *VirtualAlloc* нужно передать адрес, укладывающийся в диапазон от 0x80000000 до 0xBFFFFFFF. Только тогда Windows 98 поймет, что физическая память передается региону, зарезервированному под проецируемый файл, и даст благополучно выполнить вызов функции.

WINDOWS 2000 В Windows 2000 функция *VirtualFree* не годится для возврата физической памяти, переданной в свое время проецируемому файлу (созданному с флагом SEC_RESERVED). Однако в Windows 98 такого ограничения нет.

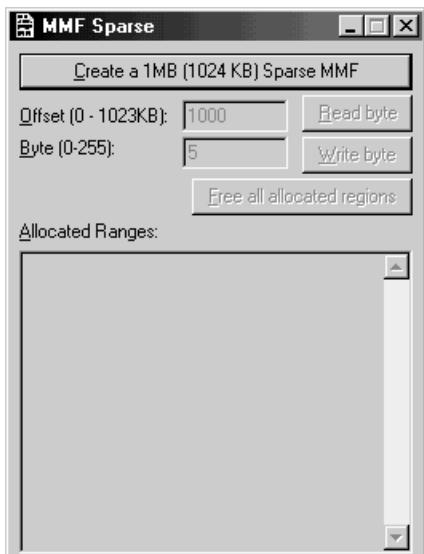
Файловая система NTFS 5 поддерживает так называемые разреженные файлы (sparse files). Это потрясающая новинка. Она позволяет легко создавать и использовать

разреженные проецируемые файлы (sparse memory-mapped files), которым физическая память предоставляется не из страничного, а из обычного дискового файла.

Вот пример того, как можно было бы воспользоваться этой новинкой. Допустим, Вы хотите создать проецируемый в память файл (MMF) для записи аудиоданных. При этом Вы должны записывать речь в виде цифровых аудиоданных в буфер памяти, связанный с дисковым файлом. Самый простой и эффективный способ решить эту задачу — применить разреженный MMF. Все дело в том, что Вам заранее не известно, сколько времени будет говорить пользователь, прежде чем щелкнет кнопку Stop. Может, пять минут, а может, пять часов — разница большая! Однако при использовании разреженного MMF это не проблема.

Программа-пример MMFSparse

Эта программа, «17 MMFSparse.exe» (см. листинг на рис. 17-4), демонстрирует, как создать проецируемый в память файл, связанный с разреженным файлом NTFS 5. Файлы исходного кода и ресурсов этой программы находятся в каталоге 17-MMFSparse на компакт-диске, прилагаемом к книге. После запуска MMFSparse на экране появляется окно, показанное ниже.

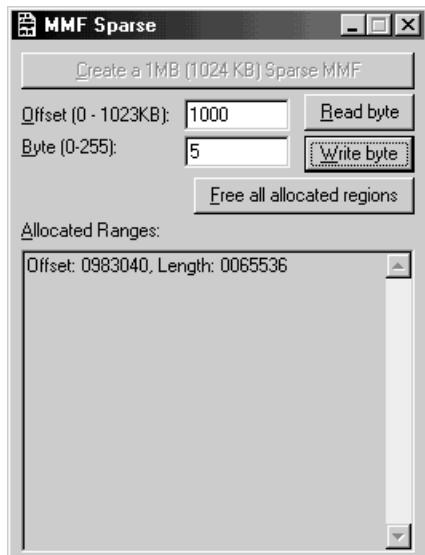


Когда Вы щелкнете кнопку Create a 1MB (1024 KB) Sparse MMF, программа попытается создать разреженный файл «C:\MMFSparse». Если Ваш диск С не является томом NTFS 5, у программы ничего не получится, и ее процесс завершится. А если Вы создали том NTFS 5 на каком-то другом диске, модифицируйте мою программу и перекомпилируйте ее, чтобы посмотреть, как она работает.

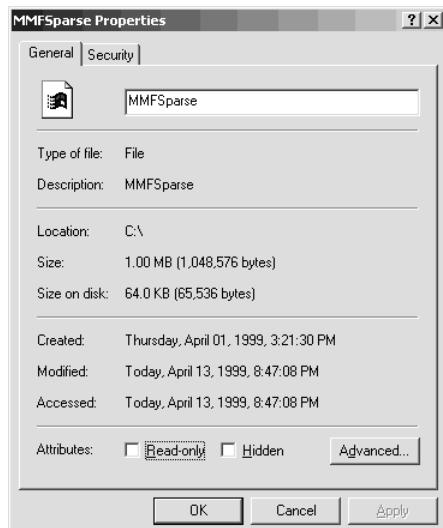
После создания разреженный файл проецируется на адресное пространство процесса. В поле Allocated Ranges (внизу окна) показывается, какие части файла действительно связаны с дисковой памятью. Изначально файл не связан ни с какой памятью, и в этом поле сообщается «No allocated ranges in the file» («В файле нет выделенных диапазонов»).

Чтобы считать байт, просто введите число в поле Offset и щелкните кнопку Read Byte. Введенное Вами число умножается на 1024 (1 Кб), и программа, считав байт по полученному адресу, выводит его значение в поле Byte. Если адрес попадает в область, не связанную с физической памятью, в этом поле всегда показывается нулевой байт.

Для записи байта введите число в поле Offset, а значение байта (0–255) — в поле Byte. Потом, когда Вы щелкнете кнопку Write Byte, смещение будет умножено на 1024, и байт по соответствующему адресу получит новое значение. Операция записи может заставить файловую систему передать физическую память какой-либо части файла. Содержимое поля Allocated Ranges обновляется после каждой операции чтения или записи, показывая, какие части файла связаны с физической памятью на данный момент. Вот как выглядит окно программы после записи всего одного байта по смещению 1 024 000 (1000 × 1024).



На этой иллюстрации видно, что физическая память выделена только одному диапазону адресов — размером 65 536 байтов, начиная с логического смещения 983 040 от начала файла. С помощью Explorer Вы можете просмотреть свойства файла C:\MMFSparse, как показано ниже.



Заметьте: на этой странице свойств сообщается, что длина файла равна 1 Мб (это виртуальный размер файла), но на деле он занимает на диске только 64 Кб.

Последняя кнопка, Free All Allocated Regions, заставляет программу высвободить всю физическую память, выделенную для файла; таким образом, соответствующее дисковое пространство освобождается, а все байты в файле обнуляются.

Теперь поговорим о том, как работает эта программа. Чтобы упростить ее исходный код, я создал C++-класс CSparseStream (который содержится в файле SparseStream.h). Этот класс инкапсулирует поддержку операций с разреженным файлом или потоком данных (stream). В файле MMFSparse.cpp я создал другой C++-класс, CMMFSparse, производный от CSparseStream. Так что объект класса CMMFSparse обладает не только функциональностью CSparseStream, но и дополнительной, необходимой для использования разреженного потока данных как проецируемого в память файла. В процессе создается единственный глобальный экземпляр класса CMMFSparse — переменная *g_mmf*. Манипулируя разреженным проецируемым файлом, программа часто ссылается на эту глобальную переменную.

Когда пользователь щелкает кнопку Create a 1MB (1024 KB) Sparse MMF, программа вызывает *CreateFile* для создания нового файла в дисковом разделе NTFS 5. Пока что это обычный, самый заурядный файл. Но потом я вызываю метод *Initialize* глобального объекта *g_mmf*, передавая ему описатель и максимальный размер файла (1 Мб). Метод *Initialize* в свою очередь обращается к *CreateFileMapping* и создает объект ядра «проекция файла» указанного размера, а затем вызывает *MapViewOfFile*, чтобы сделать разреженный файл видимым в адресном пространстве данного процесса.

Когда *Initialize* возвращает управление, вызывается функция *Dlg_ShowAllocatedRanges*. Используя Windows-функции, она перечисляет диапазоны логических адресов в разреженном файле, которым передана физическая память. Начальное смещение и длина каждого такого диапазона показываются в нижнем поле диалогового окна. В момент инициализации объекта *g_mmf* файлу на диске еще не выделена физическая память, и данное поле отражает этот факт.

Теперь пользователь может попытаться считать или записать какие-то байты в пределах разреженного проецируемого файла. При записи программа извлекает значение байта и смещение из соответствующих полей, а затем помещает этот байт по вычисленному адресу в объект *g_mmf*. Такая операция может потребовать от файловой системы передачи физической памяти логическому блоку файла, но программа не принимает в этом участия.

При чтении объекта *g_mmf* возвращается либо реальное значение байта, если данному диапазону адресов передана физическая память, либо 0, если память не передана.

Моя программа также демонстрирует, как вернуть файл в исходное состояние, высвободив все выделенные ему диапазоны адресов (после этого он фактически не занимает места на диске). Реализуется это так. Пользователь щелкает кнопку Free All Allocated Regions. Однако освободить все диапазоны адресов, выделенные файлу, который проецируется в память, нельзя. Поэтому первое, что делает программа, — вызывает метод *ForceClose* объекта *g_mmf*. Этот метод обращается к *UnmapViewOfFile*, а потом — к *CloseHandle*, передавая описатель объекта ядра «проекция файла».

Далее вызывается метод *DecommitPortionOfStream*, который освобождает всю память, выделенную логическим байтам в файле. Наконец, программа вновь обращается к методу *Initialize* объекта *g_mmf*, и тот повторно инициализирует файл, проецируемый на адресное пространство данного процесса. Чтобы подтвердить освобождение всей выделенной памяти, программа вызывает функцию *Dlg_ShowAllocatedRanges*, которая выводит в поле строку «No allocated ranges in the file».

И последнее. Используя разреженный проецируемый файл в реальном приложении, Вы, наверное, захотите при закрытии файла урезать его логический размер до фактического. Отсечение концевой части разреженного файла, содержащей нулевые

байты, не влияет на занимаемый им объем дискового пространства, но позволяет Explorer и команде dir сообщать точный размер файла. С этой целью Вы должны после вызова метода *ForceClose* использовать функции *SetFilePointer* и *SetEndOfFile*.



MMFSparse.cpp

```

/*****
Модуль: MMFSparse.cpp
Автор: Copyright (c) 2000, Джейфри Рихтер (Jeffrey Richter)
*****/

#include "..\CmnHdr.h"      /* см. приложение A */
#include <tchar.h>
#include <WindowsX.h>
#include <WinIoCtl.h>
#include "SparseStream.h"
#include "Resource.h"

///////////////////////////////



// этот класс упрощает работу с разреженными проецируемыми файлами
class CMMFSparse : public CSparseStream {
private:
    HANDLE m_hfilemap;      // объект "проекция файла"
    PVOID  m_pvFile;        // адрес начала проецируемого файла

public:
    // создает разреженный MMF и проецирует его на адресное пространство процесса
    CMMFSparse(HANDLE hstream = NULL, SIZE_T dwStreamSizeMax = 0);

    // закрывает разреженный MMF
    virtual ~CMMFSparse() { ForceClose(); }

    // создает разреженный MMF и проецирует его на адресное пространство процесса
    BOOL Initialize(HANDLE hstream, SIZE_T dwStreamSizeMax);

    // оператор приведения MMF к BYTE возвращает адрес первого байта
    // в разреженном MMF
    operator PBYTE() const { return((PBYTE) m_pvFile); }

    // позволяет явно закрывать MMF, не дожидаясь вызова деструктора
    VOID ForceClose();
};

///////////////////////////////



CMMFSparse::CMMFSparse(HANDLE hstream, SIZE_T dwStreamSizeMax) {
    Initialize(hstream, dwStreamSizeMax);
}

///////////////////////////////

```

Рис. 17-4. Программа-пример MMFSparse

см. след. стр.

Рис. 17-4. продолжение

```

BOOL CMMFSparse::Initialize(HANDLE hstream, SIZE_T dwStreamSizeMax) {

    if (m_hfilemap != NULL)
        ForceClose();

    // инициализируем значением NULL на случай, если что-то пойдет не так
    m_pvFile = m_hfilemap = NULL;

    BOOL fOk = TRUE; // предполагаем, что все будет хорошо

    if (hstream != NULL) {
        if (dwStreamSizeMax == 0) {
            DebugBreak(); // недопустимый размер потока данных
        }

        CSparseStream::Initialize(hstream);
        fOk = MakeSparse(); // делаем поток разреженным
        if (fOk) {
            // создаем объект "проекция файла"
            m_hfilemap = ::CreateFileMapping(hstream, NULL, PAGE_READWRITE,
                (DWORD) (dwStreamSizeMax >> 32i64), (DWORD) dwStreamSizeMax, NULL);

            if (m_hfilemap != NULL) {
                // проецируем поток данных на адресное пространство процесса
                m_pvFile = ::MapViewOfFile(m_hfilemap,
                    FILE_MAP_WRITE | FILE_MAP_READ, 0, 0, 0);
            } else {
                // спроектировать файл не удалось; проводим очистку
                CSparseStream::Initialize(NULL);
                ForceClose();
                fOk = FALSE;
            }
        }
    }
    return(fOk);
}

///////////////////////////////



VOID CMMFSparse::ForceClose() {

    // очищаем все, что было успешно создано
    if (m_pvFile != NULL) {
        ::UnmapViewOfFile(m_pvFile);
        m_pvFile = NULL;
    }
    if (m_hfilemap != NULL) {
        ::CloseHandle(m_hfilemap);
        m_hfilemap = NULL;
    }
}

```

Рис. 17-4. продолжение

```
//////////  

#define STREAMSIZE      (1 * 1024 * 1024) // 1 Мб (1024 Кб)  

TCHAR szPathname[] = TEXT("C:\\\\MMFSparse.");  

HANDLE g_hstream = INVALID_HANDLE_VALUE;  

CMMFSparse g_mmf;  

//////////  

BOOL Dlg_OnInitDialog(HWND hwnd, HWND hwndFocus, LPARAM lParam) {  

    chSETDLGICONS(hwnd, IDI_MMFSPARSE);  

    // инициализируем элементы управления в диалоговом окне  

    EnableWindow(GetDlgItem(hwnd, IDC_OFFSET), FALSE);  

    Edit_LimitText(GetDlgItem(hwnd, IDC_OFFSET), 4);  

    SetDlgItemInt(hwnd, IDC_OFFSET, 1000, FALSE);  

    EnableWindow(GetDlgItem(hwnd, IDC_BYTE), FALSE);  

    Edit_LimitText(GetDlgItem(hwnd, IDC_BYTE), 3);  

    SetDlgItemInt(hwnd, IDC_BYTE, 5, FALSE);  

    EnableWindow(GetDlgItem(hwnd, IDC_WRITEBYTE), FALSE);  

    EnableWindow(GetDlgItem(hwnd, IDC_READBYTE), FALSE);  

    EnableWindow(GetDlgItem(hwnd, IDC_FREEALLOCATEDREGIONS), FALSE);  

    return(TRUE);  

}  

//////////  

void Dlg_ShowAllocatedRanges(HWND hwnd) {  

    // заполняем поле Allocated Ranges  

    DWORD dwNumEntries;  

    FILE_ALLOCATED_RANGE_BUFFER* pfarb =  

        g_mmf.QueryAllocatedRanges(&dwNumEntries);  

    if (dwNumEntries == 0) {  

        SetDlgItemText(hwnd, IDC_FILESTATUS,  

            TEXT("No allocated ranges in the file"));  

    } else {  

        TCHAR sz[4096] = { 0 };  

        for (DWORD dwEntry = 0; dwEntry < dwNumEntries; dwEntry++) {  

            wsprintf(_tcschr(sz, 0), TEXT("Offset: %7.7u, Length: %7.7u\r\n"),  

                pfarb[dwEntry].FileOffset.LowPart, pfarb[dwEntry].Length.LowPart);  

        }  

        SetDlgItemText(hwnd, IDC_FILESTATUS, sz);  

    }  

    g_mmf.FreeAllocatedRanges(pfarb);  

}
```

см. след. стр.

Рис. 17-4. продолжение

```
//////////  

void Dlg_OnCommand(HWND hwnd, int id, HWND hwndCtl, UINT codeNotify) {  

    switch (id) {  

        case IDCANCEL:  

            if (g_hstream != INVALID_HANDLE_VALUE)  

                CloseHandle(g_hstream);  

            EndDialog(hwnd, id);  

            break;  

        case IDC_CREATEMMF:  

            // создаем файл  

            g_hstream = CreateFile(szPathname, GENERIC_READ | GENERIC_WRITE,  

                0, NULL, CREATE_ALWAYS, FILE_ATTRIBUTE_NORMAL, NULL);  

            if (g_hstream == INVALID_HANDLE_VALUE) {  

                chFAIL("Failed to create file.");  

            }  

            // используя этот файл, создаем MMF размером 1 Мб (1024 Кб)  

            if (!g_mmf.Initialize(g_hstream, STREAMSIZE)) {  

                chFAIL("Failed to initialize Sparse MMF.");  

            }  

            Dlg_ShowAllocatedRanges(hwnd);  

            // активизируем или отключаем остальные элементы управления  

            EnableWindow(GetDlgItem(hwnd, IDC_CREATEMMF), FALSE);  

            EnableWindow(GetDlgItem(hwnd, IDC_OFFSET), TRUE);  

            EnableWindow(GetDlgItem(hwnd, IDC_BYTEx), TRUE);  

            EnableWindow(GetDlgItem(hwnd, IDC_WRITEBYTE), TRUE);  

            EnableWindow(GetDlgItem(hwnd, IDC_READBYTE), TRUE);  

            EnableWindow(GetDlgItem(hwnd, IDC_FREEALLOCATEDREGIONS), TRUE);  

            // переводим фокус в поле Offset  

            SetFocus(GetDlgItem(hwnd, IDC_OFFSET));  

            break;  

        case IDC_WRITEBYTE:  

            {  

                BOOL fTranslated;  

                DWORD dwOffset = GetDlgItemInt(hwnd, IDC_OFFSET, &fTranslated, FALSE);  

                if (fTranslated) {  

                    g_mmf[dwOffset * 1024] = (BYTE)  

                        GetDlgItemInt(hwnd, IDC_BYTEx, NULL, FALSE);  

                    Dlg_ShowAllocatedRanges(hwnd);  

                }  

            }  

            break;  

        case IDC_READBYTE:  

            {  

                BOOL fTranslated;
```

Рис. 17-4. продолжение

```

        DWORD dwOffset = GetDlgItemInt(hwnd, IDC_OFFSET, &fTranslated, FALSE);
        if (fTranslated) {
            SetDlgItemInt(hwnd, IDC_BYTEx, g_mmf[dwOffset * 1024], FALSE);
            Dlg_ShowAllocatedRanges(hwnd);
        }
    }
    break;

case IDC_FREEALLOCATEDREGIONS:
    // обычно проекцию файла закрывает деструктор, но в данном случае
    // мы хотим сами закрыть ее, чтобы можно было вернуть часть файла
    // в исходное состояние
    g_mmf.ForceClose();

    // мы вызываем ForceClose, потому что попытка обнуления части файла
    // при его проецировании приводит к провалу вызова DeviceIoControl
    // с ошибкой ERROR_USER_MAPPED_FILE ("Запрошенная операция с файлом
    // невозможна, пока открыт раздел, проецируемый пользователем")
    g_mmf.DecommitPortionOfStream(0, STREAMSIZE);
    g_mmf.Initialize(g_hstream, STREAMSIZE);
    Dlg_ShowAllocatedRanges(hwnd);
    break;
}
}

/////////////////////////////// Конец файла /////////////////////////////////
INT_PTR WINAPI Dlg_Proc(HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam) {
    switch (uMsg) {
        chHANDLE_DLGMMSG(hwnd, WM_INITDIALOG, Dlg_OnInitDialog);
        chHANDLE_DLGMMSG(hwnd, WM_COMMAND, Dlg_OnCommand);
    }
    return(FALSE);
}

/////////////////////////////// Конец файла /////////////////////////////////
int WINAPI _tWinMain(HINSTANCE hinstExe, HINSTANCE, PTSTR pszCmdLine, int) {
    chWindows2000Required();

    DialogBox(hinstExe, MAKEINTRESOURCE(IDD_MMFSPARSE), NULL, Dlg_Proc);
    return(0);
}

/////////////////////////////// Конец файла /////////////////////////////////

```

см. след. стр.

Рис. 17-4. продолжение

SparseStream.h

```
*****  
Модуль: SparseStream.h  
Автор: Copyright (c) 2000, Джейфри Рихтер (Jeffrey Richter)  
*****  
  
#include "..\CmnHdr.h"      /* см. приложение A */  
#include <WinIoCtl.h>  
  
//////////  
  
#pragma once  
  
//////////  
  
class CSparseStream {  
public:  
    static BOOL DoesFileSystemSupportSparseStreams(PCTSTR pszVolume);  
    static BOOL DoesFileContainAnySparseStreams(PCTSTR pszPathname);  
  
public:  
    CSparseStream(HANDLE hstream = INVALID_HANDLE_VALUE) {  
        Initialize(hstream);  
    }  
  
    virtual ~CSparseStream() {}  
  
    void Initialize(HANDLE hstream = INVALID_HANDLE_VALUE) {  
        m_hstream = hstream;  
    }  
  
public:  
    operator HANDLE() const { return(m_hstream); }  
  
public:  
    BOOL IsStreamSparse() const;  
    BOOL MakeSparse();  
    BOOL DecommitPortionOfStream(  
        __int64 qwFileOffsetStart, __int64 qwFileOffsetEnd);  
  
    FILE_ALLOCATED_RANGE_BUFFER* QueryAllocatedRanges(PDWORD pdwNumEntries);  
    BOOL FreeAllocatedRanges(FILE_ALLOCATED_RANGE_BUFFER* pfarb);  
  
private:  
    HANDLE m_hstream;  
  
private:  
    static BOOL AreFlagsSet(DWORD fdwFlagBits, DWORD fFlagsToCheck) {  
        return((fdwFlagBits & fFlagsToCheck) == fFlagsToCheck);  
    }  
};
```

Рис. 17-4. продолжение

```
//////////  

inline BOOL CSparseStream::DoesFileSystemSupportSparseStreams(  

    PCTSTR pszVolume) {  

    DWORD dwFileSystemFlags = 0;  

    BOOL fOk = GetVolumeInformation(pszVolume, NULL, 0, NULL, NULL,  

        &dwFileSystemFlags, NULL, 0);  

    fOk = fOk && AreFlagsSet(dwFileSystemFlags, FILE_SUPPORTS_SPARSE_FILES);  

    return(fOk);  

}  

//////////  

inline BOOL CSparseStream::IsStreamSparse() const {  

    BY_HANDLE_FILE_INFORMATION bhfi;  

    GetFileInformationByHandle(m_hstream, &bhfi);  

    return(AreFlagsSet(bhfi.dwFileAttributes, FILE_ATTRIBUTE_SPARSE_FILE));  

}  

//////////  

inline BOOL CSparseStream::MakeSparse() {  

    DWORD dw;  

    return(DeviceIoControl(m_hstream, FSCTL_SET_SPARSE,  

        NULL, 0, NULL, 0, &dw, NULL));  

}  

//////////  

inline BOOL CSparseStream::DecommitPortionOfStream(  

    __int64 qwOffsetStart, __int64 qwOffsetEnd) {  

    // Примечание: эта функция не сработает, если файл проецируется в память  

    DWORD dw;  

    FILE_ZERO_DATA_INFORMATION fzdi;  

    fzdi.FileOffset.QuadPart = qwOffsetStart;  

    fzdi.BeyondFinalZero.QuadPart = qwOffsetEnd + 1;  

    return(DeviceIoControl(m_hstream, FSCTL_SET_ZERO_DATA, (LPVOID) &fzdi,  

        sizeof(fzdi), NULL, 0, &dw, NULL));  

}  

//////////  

inline BOOL CSparseStream::DoesFileContainAnySparseStreams(  

    PCTSTR pszPathname) {  

    DWORD dw = GetFileAttributes(pszPathname);  

    return((dw == 0xffffffff)
```

см. след. стр.

Рис. 17-4. продолжение

```
? FALSE : AreFlagsSet(dw, FILE_ATTRIBUTE_SPARSE_FILE));
}

////////// inline FILE_ALLOCATED_RANGE_BUFFER* CSparseStream::QueryAllocatedRanges(
PDWORD pdwNumEntries) {

FILE_ALLOCATED_RANGE_BUFFER farb;
farb.FileOffset.QuadPart = 0;
farb.Length.LowPart =
GetFileSize(m_hstream, (PDWORD) &farb.Length.HighPart);

// правильно определить размер блока памяти до попытки сбора этих данных
// нельзя, и я просто беру 100 * sizeof(*pfarb)
DWORD cb = 100 * sizeof(farb);
FILE_ALLOCATED_RANGE_BUFFER* pfarb = (FILE_ALLOCATED_RANGE_BUFFER*)
HeapAlloc(GetProcessHeap(), HEAP_ZERO_MEMORY, cb);

DeviceIoControl(m_hstream, FSCTL_QUERY_ALLOCATED_RANGES,
&farb, sizeof(farb), pfarb, cb, &cb, NULL);
*pdwNumEntries = cb / sizeof(*pfarb);
return(pfarb);
}

////////// inline BOOL CSparseStream::FreeAllocatedRanges(
FILE_ALLOCATED_RANGE_BUFFER* pfarb) {

// освобождаем выделенную память
return(HeapFree(GetProcessHeap(), 0, pfarb));
}

////////// Конец файла /////////////////
```

Динамически распределяемая память

Третий, и последний, механизм управления памятью — динамически распределяемые области памяти, или кучи (heaps). Они весьма удобны при создании множества небольших блоков данных. Например, связанными списками и деревьями проще манипулировать, используя именно кучи, а не виртуальную память (глава 15) или файлы, проецируемые в память (глава 17). Преимущество динамически распределяемой памяти в том, что она позволяет Вам игнорировать гранулярность выделения памяти и размер страниц и сосредоточиться непосредственно на своей задаче. А недостаток — выделение и освобождение блоков памяти проходит медленнее, чем при использовании других механизмов, и, кроме того, Вы теряете прямой контроль над передачей физической памяти и ее возвратом системе.

Куча — это регион зарезервированного адресного пространства. Первоначально большей его части физическая память не передается. По мере того, как программа занимает эту область под данные, специальный диспетчер, управляющий кучами (heap manager), постринично передает ей физическую память (из страничного файла). А при освобождении блоков в куче диспетчер возвращает системе соответствующие страницы физической памяти.

Microsoft не документирует правила, по которым диспетчер передает или отбирает физическую память. Эти правила различны в Windows 98 и Windows 2000. Могу сказать Вам лишь следующее: Windows 98 больше озабочена эффективностью использования памяти и поэтому старается как можно быстрее отобрать у куч физическую память. Однако Windows 2000 нацелена главным образом на максимальное быстродействие, в связи с чем возвращает физическую память в страничный файл, только если страницы не используются в течение определенного времени. Microsoft постоянно проводит стрессовое тестирование своих операционных систем и прогоняет разные сценарии, чтобы определить, какие правила в большинстве случаев работают лучше. Их приходится менять по мере появления как нового программного обеспечения, так и оборудования. Если эти правила важны Вашим программам, использовать динамически распределяемую память не стоит — работайте с функциями виртуальной памяти (т. е. *VirtualAlloc* и *VirtualFree*), и тогда Вы сможете сами контролировать эти правила.

Стандартная куча процесса

При инициализации процесса система создает в его адресном пространстве стандартную кучу (process's default heap). Ее размер по умолчанию — 1 Мб. Но система позволяет увеличивать этот размер, для чего надо указать компоновщику при сборке про-

граммы ключ /HEAP. (Однако при сборке DLL этим ключом пользоваться нельзя, так как для DLL куча не создается.)

/HEAP:reserve[, commit]

Стандартная куча процесса необходима многим Windows-функциям. Например, функции ядра Windows 2000 выполняют все операции с использованием Unicode-символов и строк. Если вызвать ANSI-версию какой-нибудь Windows-функции, ей придется, преобразовав строки из ANSI в Unicode, вызывать свою Unicode-версию. Для преобразования строк ANSI-функции нужно выделить блок памяти, в котором она размещает Unicode-версию строки. Этот блок памяти заимствуется из стандартной кучи вызывающего процесса. Есть и другие функции, использующие временные блоки памяти, которые тоже выделяются из стандартной кучи процесса. Из нее же черпают себе память и функции 16-разрядной Windows, управляющие кучами (*LocalAlloc* и *GlobalAlloc*).

Поскольку стандартную кучу процесса используют многие Windows-функции, а потоки Вашего приложения могут одновременно вызвать массу таких функций, доступ к этой куче разрешается только по очереди. Иными словами, система гарантирует, что в каждый момент времени только один поток сможет выделить или освободить блок памяти в этой куче. Если же два потока попытаются выделить в ней блоки памяти одновременно, второй поток будет ждать, пока первый поток не выделит свой блок. Принцип последовательного доступа потоков к куче немного снижает производительность многопоточной программы. Если в программе всего один поток, для быстрейшего доступа к куче нужно создать отдельную кучу и не использовать стандартную. Но Windows-функциям этого, увы, не прикажешь — они работают с кучей только последнего типа.

Как я уже говорил, куч у одного процесса может быть несколько. Они создаются и разрушаются в период его существования. Но стандартная куча процесса создается в начале его исполнения и автоматически уничтожается по его завершении — сами уничтожить ее Вы не можете. Каждую кучу идентифицирует свой описатель, и все Windows-функции, которые выделяют и освобождают блоки в ее пределах, требуют передавать им этот описатель как параметр.

Описатель стандартной кучи процесса возвращает функция *GetProcessHeap*:

```
HANDLE GetProcessHeap();
```

Дополнительные кучи в процессе

В адресном пространстве процесса допускается создание дополнительных куч. Для чего они нужны? Тому может быть несколько причин:

- защита компонентов;
- более эффективное управление памятью;
- локальный доступ;
- исключение издержек, связанных с синхронизацией потоков;
- быстрое освобождение всей памяти в куче.

Рассмотрим эти причины подробнее.

Защита компонентов

Допустим, программа должна обрабатывать два компонента: связанный список структур NODE и двоичное дерево структур BRANCH. Представим также, что у Вас есть два

файла исходного кода: LnkLst.cpp, содержащий функции для обработки связанного списка, и BinTree.cpp с функциями для обработки двоичного дерева.

Если структуры NODE и BRANCH хранятся в одной куче, то она может выглядеть примерно так, как показано на рис. 18-1.

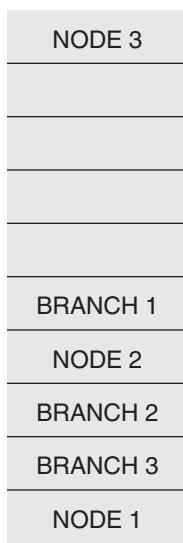


Рис. 18-1. Единая куча, в которой размещены структуры NODE и BRANCH

Теперь предположим, что в коде, обрабатывающем связанный список, «сидит жучок», который приводит к случайной перезаписи 8 байтов после NODE 1. А это в свою очередь влечет порчу данных в BRANCH 3. Впоследствии, когда код из файла BinTree.cpp пытается «пройти» по двоичному дереву, происходит сбой из-за того, что часть данных в памяти испорчена. Можно подумать, что ошибка возникает из-за «жучка» в коде двоичного дерева, тогда как на самом деле он — в коде связанных списков. А поскольку разные типы объектов смешаны в одну кучу (в прямом и переносном смысле), то отловить «жучков» в коде становится гораздо труднее.

Создав же две отдельные кучи — одну для NODE, другую для BRANCH, — Вы локализуете место возникновения ошибки. И тогда «жучок» в коде связанных списков не испортит целостности двоичного дерева, и наоборот. Конечно, всегда остается вероятность такой фатальной ошибки в коде, которая приведет к записи данных в стороннюю кучу, но это случается значительно реже.

Более эффективное управление памятью

Кучами можно управлять гораздо эффективнее, создавая в них объекты одинакового размера. Допустим, каждая структура NODE занимает 24 байта, а каждая структура BRANCH — 32. Память для всех этих объектов выделяется из одной кучи. На рис. 18-2 показано, как выглядит полностью занятая куча с некоторыми объектами NODE и BRANCH. Если объекты NODE 2 и NODE 4 удаляются, память в куче становится фрагментированной. И если после этого попытаться выделить в ней память для структуры BRANCH, ничего не выйдет — даже несмотря на то что в куче свободно 48 байтов, а структура BRANCH требует всего 32.

Если бы в каждой куче содержались объекты одинакового размера, удаление одного из них позволило бы в дальнейшем разместить другой объект того же типа.

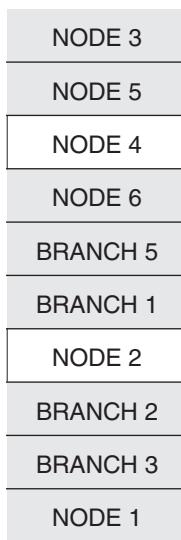


Рис. 18-2. Фрагментированная куча, содержащая несколько объектов NODE и BRANCH

Локальный доступ

Перекачка страницы из оперативной памяти в страничный файл занимает ощутимое время. Та же задержка происходит и в момент загрузки страницы данных обратно в оперативную память. Обращаясь в основном к памяти, локализованной в небольшом диапазоне адресов, Вы снизите вероятность перекачки страниц между оперативной памятью и страничным файлом.

Поэтому при разработке приложения старайтесь размещать объекты, к которым необходим частый доступ, как можно плотнее друг к другу. Возвращаясь к примеру со связанным списком и двоичным деревом, отмечу, что просмотр списка не связан с просмотром двоичного дерева. Разместив все структуры NODE друг за другом в одной куче, Вы, возможно добьетесь того, что по крайней мере несколько структур NODE уместятся в пределах одной страницы физической памяти. И тогда просмотр связанного списка не потребует от процессора при каждом обращении к какой-либо структуре NODE переключаться с одной страницы на другую.

Если же «свалить» оба типа структур в одну кучу, объекты NODE неизбежно будут размещены строго друг за другом. При самом неблагоприятном стечении обстоятельств на странице окажется всего одна структура NODE, а остальное место займут структуры BRANCH. В этом случае просмотр связанного списка будет приводить к ошибке страницы (page fault) при обращении к каждой структуре NODE, что в результате может чрезвычайно замедлить скорость выполнения Вашего процесса.

Исключение издержек, связанных с синхронизацией потоков

Доступ к кучам упорядочивается по умолчанию, поэтому при одновременном обращении нескольких потоков к куче данные в ней никогда не повреждаются. Однако для этого функциям, работающим с кучами, приходится выполнять дополнительный код. Если Вы интенсивно манипулируете с динамически распределяемой памятью, выполнение дополнительного кода может заметно снизить быстродействие Вашей программы. Создавая новую кучу, Вы можете сообщить системе, что единовременно к этой куче обращается только один поток, и тогда дополнительный код выполняться не

будет. Но берегитесь: теперь Вы берете всю ответственность за целостность этой кучи на себя. Система не станет присматривать за Вами.

Быстрое освобождение всей памяти в куче

Наконец, использование отдельной кучи для какой-то структуры данных позволяет освобождать всю кучу, не перебирая каждый блок памяти. Например, когда Windows Explorer перечисляет иерархию каталогов на жестком диске, он формирует их дерево в памяти. Получив команду обновить эту информацию, он мог бы просто разрушить кучу, содержащую это дерево, и начать все заново (если бы, конечно, он использовал кучу, выделенную только для информации о дереве каталогов). Во многих приложениях это было бы очень удобно, да и быстродействие тоже возросло бы.

Создание дополнительной кучи

Дополнительные кучи в процессе создаются вызовом *HeapCreate*:

```
HANDLE HeapCreate(
    DWORD fdwOptions,
    SIZE_T dwInitialSize,
    SIZE_T dwMaximumSize);
```

Параметр *fdwOptions* модифицирует способ выполнения операций над кучей. В нем можно указать 0, HEAP_NO_SERIALIZE, HEAP_GENERATE_EXCEPTIONS или комбинацию последних двух флагов.

По умолчанию действует принцип последовательного доступа к куче, что позволяет не опасаться одновременного обращения к ней сразу нескольких потоков. При попытке выделения из кучи блока памяти функция *HeapAlloc* (ее параметры мы обсудим чуть позже) делает следующее:

1. Просматривает связанный список выделенных и свободных блоков памяти.
2. Находит адрес свободного блока.
3. Выделяет новый блок, помечая свободный как занятый.
4. Добавляет новый элемент в связанный список блоков памяти.

Флаг HEAP_NO_SERIALIZE использовать не следует, и вот почему. Допустим, два потока одновременно пытаются выделить блоки памяти из одной кучи. Первый поток выполняет операции по пп. 1 и 2 и получает адрес свободного блока памяти. Но только он собирается перейти к третьему этапу, как его вытеснит второй поток и тоже выполнит операции по пп. 1 и 2. Поскольку первый поток не успел дойти до этапа 3, второй поток обнаружит тот же свободный блок памяти.

Итак, оба потока считают, что они нашли свободный блок памяти в куче. Поэтому поток 1 обновляет связанный список, помечая новый блок как занятый. После этого и поток 2 обновляет связанный список, помечая *тот же* блок как занятый. Ни один из потоков пока ничего не подозревает, хотя оба получили адреса, указывающие на один и тот же блок памяти.

Ошибку такого рода обнаружить очень трудно, поскольку она проявляется не сразу. Но в конце концов сбой произойдет и, будьте уверены, это случится в самый неподходящий момент. Вот какие проблемы это может вызвать.

- Повреждение связанного списка блоков памяти. Эта проблема не проявится до попытки выделения или освобождения блока.

- Оба потока делят один и тот же блок памяти. Оба записывают в него свою информацию. Когда поток 1 начнет просматривать содержимое блока, он не поймет данные, записанные потоком 2.
- Один из потоков, закончив работу с блоком, освобождает его, и это приводит к тому, что другой поток записывает данные в невыделенную память. Происходит повреждение кучи.

Решение этих проблем — предоставить одному из потоков монопольный доступ к куче и ее связенному списку (пока он не закончит все необходимые операции с кучей). Именно так и происходит в отсутствие флага `HEAP_NO_SERIALIZE`. Этот флаг можно использовать без опаски только при выполнении следующих условий:

- в процессе существует лишь один поток;
- в процессе несколько потоков, но с кучей работает лишь один из них;
- в процессе несколько потоков, но он сам регулирует доступ потоков к куче, применяя различные формы взаимоисключения, например критические секции, объекты-мьютексы или семафоры (см. главы 8 и 9).

Если Вы не уверены, нужен ли Вам флаг `HEAP_NO_SERIALIZE`, лучше не пользуйтесь им. В его отсутствие скорость работы многопоточной программы может чуть снизиться из-за задержек при вызовах функций, управляющих кучами, но зато Вы избежите риска повреждения кучи и ее данных.

Другой флаг, `HEAP_GENERATE_EXCEPTIONS`, заставляет систему генерировать исключение при любом провале попытки выделения блока в куче. Исключение (см. главы 23, 24 и 25) — еще один способ уведомления программы об ошибке. Иногда приложение удобнее разрабатывать, полагаясь на перехват исключений, а не на проверку значений, возвращаемых функциями.

Второй параметр функции `HeapCreate` — `dwInitialSize` — определяет количество байтов, первоначально передаваемых куче. При необходимости функция округляет это значение до ближайшей большей величины, кратной размеру страниц. И последний параметр, `dwMaximumSize`, указывает максимальный объем, до которого может расширяться куча (пределный объем адресного пространства, резервируемого под кучу). Если он больше 0, Вы создадите кучу именно такого размера и не сможете его увеличить. А если этот параметр равен 0, система резервирует регион и, если надо, расширяет его до максимально возможного объема. При успешном создании кучи `HeapCreate` возвращает описатель, идентифицирующий новую кучу. Он используется и другими функциями, работающими с кучами.

Выделение блока памяти из кучи

Для этого достаточно вызвать функцию `HeapAlloc`:

```
PVOID HeapAlloc(  
    HANDLE hHeap,  
    DWORD  fdwFlags,  
    SIZE_T dwBytes);
```

Параметр `bHeap` идентифицирует описатель кучи, из которой выделяется память. Параметр `dwBytes` определяет число выделяемых в куче байтов, а параметр `fdwFlags` позволяет указывать флаги, влияющие на характер выделения памяти. В настоящее время поддерживается только три флага: `HEAP_ZERO_MEMORY`, `HEAP_GENERATE_EXCEPTIONS` и `HEAP_NO_SERIALIZE`.

Назначение флага `HEAP_ZERO_MEMORY` очевидно. Он приводит к заполнению содержимого блока нулями перед возвратом из `HeapAlloc`. Второй флаг заставляет эту функцию генерировать программное исключение, если в куче не хватает памяти для удовлетворения запроса. Вспомните, этот флаг можно указывать и при создании кучи функцией `HeapCreate`; он сообщает диспетчеру, управляющему кучами, что при невозможности выделения блока в куче надо генерировать соответствующее исключение. Если Вы включили данный флаг при вызове `HeapCreate`, то при вызове `HeapAlloc` указывать его уже не нужно. С другой стороны, Вы могли создать кучу без флага `HEAP_GENERATE_EXCEPTIONS`. В таком случае, если Вы укажете его при вызове `HeapAlloc`, он повлияет лишь на данный ее вызов.

Если функция `HeapAlloc` завершилась неудачно и при этом разрешено генерировать исключения, она может вызвать одно из двух исключений, перечисленных в следующей таблице.

Идентификатор	Описание
<code>STATUS_NO_MEMORY</code>	Попытка выделения памяти не удалась из-за ее нехватки
<code>STATUS_ACCESS_VIOLATION</code>	Попытка выделения памяти не удалась из-за повреждения кучи или неверных параметров функции

При успешном выделении блока `HeapAlloc` возвращает его адрес. Если памяти недостаточно и флаг `HEAP_GENERATE_EXCEPTIONS` не указан, функция возвращает `NULL`.

Флаг `HEAP_NO_SERIALIZE` заставляет `HeapAlloc` при данном вызове не применять принцип последовательного доступа к куче. Этим флагом нужно пользоваться с величайшей осторожностью, так как куча (особенно стандартная куча процесса) может быть повреждена при одновременном доступе к ней нескольких потоков.

WINDOWS 98 Вызов `HeapAlloc` с требованием выделить блок размером более 256 Мб Windows 98 считает ошибкой. Заметьте, что в этом случае функция всегда возвращает `NULL`, а исключение никогда не генерируется, даже если при создании кучи или попытке выделить блок Вы указали флаг `HEAP_GENERATE_EXCEPTIONS`.



Для выделения больших блоков памяти (от 1 Мб) рекомендуется использовать функцию `VirtualAlloc`, а не функции, оперирующие с кучами.

Изменение размера блока

Часто бывает необходимо изменить размер блока памяти. Некоторые приложения изначально выделяют больший, чем нужно, блок, а затем, разместив в нем данные, уменьшают его. Но некоторые, наоборот, сначала выделяют небольшой блок памяти и потом увеличивают его по мере записи новых данных. Для изменения размера блока памяти вызывается функция `HeapReAlloc`:

```
PVOID HeapReAlloc(
    HANDLE hHeap,
    DWORD fdwFlags,
    PVOID pvMem,
    SIZE_T dwBytes);
```

Как всегда, параметр `hHeap` идентифицирует кучу, в которой содержится изменяемый блок. Параметр `fdwFlags` указывает флаги, используемые при изменении разме-

ра блока: HEAP_GENERATE_EXCEPTIONS, HEAP_NO_SERIALIZE, HEAP_ZERO_MEMORY или HEAP_REALLOC_IN_PLACE_ONLY.

Первые два флага имеют тот же смысл, что и при использовании с *HeapAlloc*. Флаг HEAP_ZERO_MEMORY полезен только при увеличении размера блока памяти. В этом случае дополнительные байты, включаемые в блок, предварительно обнуляются. При уменьшении размера блока этот флаг не действует.

Флаг HEAP_REALLOC_IN_PLACE_ONLY сообщает *HeapReAlloc*, что данный блок памяти перемещать внутри кучи не разрешается (а именно это и может попытаться сделать функция при расширении блока). Если функция сможет расширить блок без его перемещения, она расширит его и вернет исходный адрес блока. С другой стороны, если для расширения блока его надо переместить, она возвращает адрес нового, большего по размеру блока. Если блок затем снова уменьшается, функция вновь возвращает исходный адрес первоначального блока. Флаг HEAP_REALLOC_IN_PLACE_ONLY имеет смысл указывать, когда блок является частью связанного списка или дерева. В этом случае в других узлах списка или дерева могут содержаться указатели на данный узел, и его перемещение в куче непременно приведет к нарушению целостности связанного списка.

Остальные два параметра (*pvMem* и *dwBytes*) определяют текущий адрес изменяемого блока и его новый размер (в байтах). Функция *HeapReAlloc* возвращает либо адрес нового, измененного блока, либо NULL, если размер блока изменить не удалось.

Определение размера блока

Выделив блок памяти, можно вызвать *HeapSize* и узнать его истинный размер:

```
SIZE_T HeapSize(
    HANDLE hHeap,
    DWORD fdwFlags,
    LPVOID pvMem);
```

Параметр *hHeap* идентифицирует кучу, а параметр *pvMem* сообщает адрес блока. Параметр *fdwFlags* принимает два значения: 0 или HEAP_NO_SERIALIZE.

Освобождение блока

Для этого служит функция *HeapFree*:

```
BOOL HeapFree(
    HANDLE hHeap,
    DWORD fdwFlags,
    PVOID pvMem);
```

Она освобождает блок памяти и при успешном вызове возвращает TRUE. Параметр *fdwFlags* принимает два значения: 0 или HEAP_NO_SERIALIZE. Обращение к этой функции может привести к тому, что диспетчер, управляющий кучами, вернет часть физической памяти системе, но это не обязательно.

Уничтожение кучи

Кучу можно уничтожить вызовом *HeapDestroy*:

```
BOOL HeapDestroy(HANDLE hHeap);
```

Обращение к этой функции приводит к освобождению всех блоков памяти внутри кучи и возврату системе физической памяти и зарезервированного региона адрес-

ного пространства, занятых кучей. При успешном выполнении функция возвращает TRUE. Если при завершении процесса Вы не уничтожаете кучу, это делает система, но — подчеркну еще раз — только в момент завершения процесса. Если куча создана потоком, она будет уничтожена лишь при завершении всего процесса.

Система не позволит уничтожить стандартную кучу процесса — она разрушается только при завершении процесса. Если Вы передадите описатель этой кучи функции *HeapDestroy*, система просто проигнорирует Ваш вызов.

Использование куч в программах на C++

Чтобы в полной мере использовать преимущества динамически распределяемой памяти, следует включить ее поддержку в существующие программы, написанные на C++. В этом языке выделение памяти для объекта класса выполняется вызовом оператора *new*, а не функцией *malloc*, как в обычной библиотеке С. Когда необходимость в данном объекте класса отпадает, вместо библиотечной С-функции *free* следует применять оператор *delete*. Скажем, у нас есть класс CSomeClass, и мы хотим создать экземпляр этого класса. Для этого нужно написать что-то вроде:

```
CSomeClass* pSomeClass = new CSomeClass;
```

Дойдя до этой строки, компилятор C++ сначала проверит, содержит ли класс CSomeClass функцию-член, переопределяющую оператор *new*. Если да, компилятор генерирует код для вызова этой функции. Нет — создает код для вызова стандартного C++-оператора *new*.

Созданный объект уничтожается обращением к оператору *delete*:

```
delete pSomeClass;
```

Переопределяя операторы *new* и *delete* для нашего C++-класса, мы получаем возможность использовать преимущества функций, управляющих кучами. Для этого определим класс CSomeClass в заголовочном файле, скажем, так:

```
class CSomeClass {
private:
    static HANDLE s_hHeap;
    static UINT s_uNumAllocsInHeap;

    // здесь располагаются закрытые данные и функции-члены
    :

public:
    void* operator new (size_t size);
    void operator delete (void* p);
    // здесь располагаются открытые данные и функции-члены
    :
};
```

Я объявил два элемента данных, *s_hHeap* и *s_uNumAllocsInHeap*, как статические переменные. А раз так, то компилятор C++ заставит все экземпляры класса CSomeClass использовать одни и те же переменные. Иначе говоря, он не станет выделять отдельные переменные *s_hHeap* и *s_uNumAllocsInHeap* для каждого создаваемого экземпляра класса. Это очень важно: ведь мы хотим, чтобы все экземпляры класса CSomeClass были созданы в одной куче.

Переменная *s_hHeap* будет содержать описатель кучи, в которой создаются объекты CSomeClass. Переменная *s_uNumAllocsInHeap* — просто счетчик созданных в куче

объектов CSomeClass. Она увеличивается на 1 при создании в куче нового объекта CSomeClass и соответственно уменьшается при его уничтожении. Когда счетчик обнуляется, куча освобождается. Для управления кучей в CPP-файл следует включить примерно такой код:

```
HANDLE CSomeClass::s_hHeap = NULL;
UINT CSomeClass::s_uNumAllocsInHeap = 0;

void* CSomeClass::operator new (size_t size) {
    if (s_hHeap == NULL) {
        // куча не существует; создаем ее
        s_hHeap = HeapCreate(HEAP_NO_SERIALIZE, 0, 0);

        if (s_hHeap == NULL)
            return(NULL);
    }
    // куча для объектов CSomeClass существует
    void* p = HeapAlloc(s_hHeap, 0, size);
    if (p != NULL) {
        // память выделена успешно; увеличиваем счетчик объектов CSomeClass в куче
        s_uNumAllocsInHeap++;
    }
    // возвращаем адрес созданного объекта CSomeClass
    return(p);
}
```

Заметьте, что сначала я объявил два статических элемента данных, *s_hHeap* и *s_uNumAllocsInHeap*, а затем инициализировал их значениями NULL и 0 соответственно.

Оператор *new* принимает один параметр — *size*, указывающий число байтов, нужных для хранения CSomeClass. Первым делом он создает кучу, если таковой нет. Для проверки анализируется значение переменной *s_hHeap*: если оно NULL, кучи нет, и тогда она создается функцией *HeapCreate*, а описатель, возвращаемый функцией, сохраняется в переменной *s_hHeap*, чтобы при следующем вызове оператора *new* использовать существующую кучу, а не создавать еще одну.

Вызывая *HeapCreate*, я указал флаг *HEAP_NO_SERIALIZE*, потому что данная программа построена как однопоточная. Остальные параметры, указанные при вызове *HeapCreate*, определяют начальный и максимальный размер кучи. Я подставил на их место по нулю. Первый нуль означает, что у кучи нет начального размера, второй — что куча должна расширяться по мере необходимости.

Не исключено, что Вам показалось, будто параметр *size* оператора *new* стоит передать в *HeapCreate* как второй параметр. Вроде бы тогда можно инициализировать кучу так, чтобы она была достаточно большой для размещения одного экземпляра класса. И в таком случае функция *HeapAlloc* при первом вызове работала бы быстрее, так как не пришлось бы изменять размер кучи под экземпляр класса. Увы, мир устроен не так, как хотелось бы. Из-за того, что с каждым выделенным внутри кучи блоком памяти связан свой заголовок, при вызове *HeapAlloc* все равно пришлось бы менять размер кучи, чтобы в нее поместился не только экземпляр класса, но и связанный с ним заголовок.

После создания кучи из нее можно выделять память под новые объекты CSomeClass с помощью функции *HeapAlloc*. Первый параметр — описатель кучи, второй — размер объекта CSomeClass. Функция возвращает адрес выделенного блока.

Если выделение прошло успешно, я увеличиваю переменную-счетчик *s_uNumAllocsInHeap*, чтобы знать число выделенных блоков в куче. Наконец, оператор *new* возвращает адрес только что созданного объекта *CSomeClass*.

Вот так происходит создание нового объекта *CSomeClass*. Теперь рассмотрим, как этот объект разрушается, — если он больше не нужен программе. Эта задача возлагается на функцию, переопределяющую оператор *delete*:

```
void CSomeClass::operator delete (void* p) {
    if (HeapFree(s_hHeap, 0, p)) {
        // объект удален успешно
        s_uNumAllocsInHeap--;
    }

    if (s_uNumAllocsInHeap == 0) {
        // если в куче больше нет объектов, уничтожаем ее
        if (HeapDestroy(s_hHeap)) {
            // описатель кучи приравниваем NULL, чтобы оператор new
            // мог создать новую кучу при создании нового объекта CSomeClass
            s_hHeap = NULL;
        }
    }
}
```

Оператор *delete* принимает только один параметр: адрес удаляемого объекта. Сначала он вызывает *HeapFree* и передает ей описатель кучи и адрес высвобождаемого объекта. Если объект освобожден успешно, *s_uNumAllocsInHeap* уменьшается, показывая, что одним объектом *CSomeClass* в куче стало меньше. Далее оператор проверяет: не равна ли эта переменная 0, и, если да, вызывает *HeapDestroy*, передавая ей описатель кучи. Если куча уничтожена, *s_hHeap* присваивается NULL. Это важно: ведь в будущем наша программа может попытаться создать другой объект *CSomeClass*. При этом будет вызван оператор *new*, который проверит значение *s_hHeap*, чтобы определить, нужно ли использовать существующую кучу или создать новую.

Данный пример иллюстрирует очень удобную схему работы с несколькими кучами. Этот код легко подстроить и включить в Ваши классы. Но сначала, может быть, стоит поразмыслить над проблемой наследования. Если при создании нового класса Вы используете класс *CSomeClass* как базовый, то производный класс унаследует операторы *new* и *delete*, принадлежащие классу *CSomeClass*. Новый класс унаследует и его кучу, а это значит, что применение оператора *new* к производному классу повлечет выделение памяти для объекта этого класса из той же кучи, которую использует и класс *CSomeClass*. Хорошо это или нет, зависит от конкретной ситуации. Если объекты сильно различаются размерами, это может привести к фрагментации кучи, что затруднит выявление таких ошибок в коде, о которых я рассказывал в разделах «Защита компонентов» и «Более эффективное управление памятью».

Если Вы хотите использовать отдельную кучу для производных классов, нужно продублировать все, что я сделал для класса *CSomeClass*. А конкретнее — включить еще один набор переменных *s_hHeap* и *s_uNumAllocsInHeap* и повторить еще раз код для операторов *new* и *delete*. Компилятор увидит, что Вы переопределили в производном классе операторы *new* и *delete*, и сформирует обращение именно к ним, а не к тем, которые содержатся в базовом классе.

Если Вы не будете создавать отдельные кучи для каждого класса, то получите единственное преимущество: Вам не придется выделять память под каждую кучу и соответствующие заголовки. Но кучи и заголовки не занимают значительных объемов

памяти, так что даже это преимущество весьма сомнительно. Неплохо, конечно, если каждый класс, используя свою кучу, в то же время имеет доступ к куче базового класса. Но делать так стоит лишь после полной отладки приложения. И, кстати, проблему фрагментации куч это не снимает.

Другие функции управления кучами

Кроме уже упомянутых, в Windows есть еще несколько функций, предназначенных для управления кучами. В этом разделе я вкратце расскажу Вам о них.

ToolHelp-функции (упомянутые в конце главы 4) дают возможность перечислять кучи процесса, а также выделенные внутри них блоки памяти. За более подробной информацией я отсылаю Вас к документации Platform SDK: ищите разделы по функциям *Heap32First*, *Heap32Next*, *Heap32ListFirst* и *Heap32ListNext*. Самое ценное в этих функциях то, что они доступны как в Windows 98, так и в Windows 2000. Прочие функции, о которых пойдет речь в этом разделе, есть только в Windows 2000.

В адресном пространстве процесса может быть несколько куч, и функция *GetProcessHeaps* позволяет получить их описатели «одним махом»:

```
DWORD GetProcessHeaps(
    DWORD dwNumHeaps,
    PHANDLE pHeaps);
```

Предварительно Вы должны создать массив описателей, а затем вызвать функцию так, как показано ниже.

```
HANDLE hHeaps[25];
DWORD dwHeaps = GetProcessHeaps(25, hHeaps);
if (dwHeaps > 25) {
    // у процесса больше куч, чем мы ожидали
} else {
    // элементы от hHeaps[0] до hHeaps[dwHeaps - 1]
    // идентифицируют существующие кучи
}
```

Имейте в виду, что описатель стандартной кучи процесса тоже включается в этот массив описателей, возвращаемый функцией *GetProcessHeaps*.

Целостность кучи позволяет проверить функция *HeapValidate*:

```
BOOL HeapValidate(
    HANDLE hHeap,
    DWORD fdwFlags,
    LPCVOID pvMem);
```

Обычно ее вызывают, передавая в *hHeap* описатель кучи, в *fdwFlags* — 0 (этот параметр допускает еще флаг *HEAP_NO_SERIALIZE*), а в *pvMem* — NULL. Функция просматривает все блоки в куче, чтобы убедиться в отсутствии поврежденных блоков. Чтобы она работала быстрее, в параметре *pvMem* можно передать адрес конкретного блока. Тогда функция проверит только этот блок.

Для объединения свободных блоков в куче, а также для возврата системе любых страниц памяти, на которых нет выделенных блоков, предназначена функция *HeapCompact*:

```
UINT HeapCompact(
    HANDLE hHeap,
    DWORD fdwFlags);
```

Обычно в параметре *fdwFlags* передают 0, но можно передать и HEAP_NO_SERIALIZE.

Следующие две функции — *HeapLock* и *HeapUnlock* — используются парно:

```
BOOL HeapLock(HANDLE hHeap);
BOOL HeapUnlock(HANDLE hHeap);
```

Они предназначены для синхронизации потоков. После успешного вызова *HeapLock* поток, который вызывал эту функцию, становится владельцем указанной кучи. Если другой поток обращается к этой куче, указывая тот же описатель кучи, система приостанавливает его выполнение до тех пор, пока куча не будет разблокирована вызовом *HeapUnlock*.

Функции *HeapAlloc*, *HeapSize*, *HeapFree* и другие — все обращаются к *HeapLock* и *HeapUnlock*, чтобы обеспечить последовательный доступ к куче. Самостоятельно вызывать эти функции Вам вряд ли понадобится.

Последняя функция, предназначенная для работы с кучами, — *HeapWalk*:

```
BOOL HeapWalk(
    HANDLE hHeap,
    PPROCESS_HEAP_ENTRY pHeapEntry);
```

Она предназначена только для отладки и позволяет просматривать содержимое кучи. Обычно ее вызывают по несколько раз, передавая адрес структуры PROCESS_HEAP_ENTRY (Вы должны сами создать ее экземпляр и инициализировать):

```
typedef struct _PROCESS_HEAP_ENTRY {
    PVOID lpData;
    DWORD cbData;
    BYTE cbOverhead;
    BYTE iRegionIndex;
    WORD wFlags;
    union {
        struct {
            HANDLE hMem;
            DWORD dwReserved[ 3 ];
        } Block;
        struct {
            DWORD dwCommittedSize;
            DWORD dwUnCommittedSize;
            LPVOID lpFirstBlock;
            LPVOID lpLastBlock;
        } Region;
    };
} PROCESS_HEAP_ENTRY, *LPPPROCESS_HEAP_ENTRY, *PPPROCESS_HEAP_ENTRY;
```

Прежде чем перечислять блоки в куче, присвойте NULL элементу *lpData*, и это заставит функцию *HeapWalk* инициализировать все элементы структуры. Чтобы перейти к следующему блоку, вызовите *HeapWalk* еще раз, передав ей тот же описатель кучи и адрес той же структуры PROCESS_HEAP_ENTRY. Если *HeapWalk* вернет FALSE, значит, блоков в куче больше нет. Подробное описание элементов структуры PROCESS_HEAP_ENTRY см. в документации Platform SDK.

Обычно вызовы функции *HeapWalk* «обрамляют» вызовами *HeapLock* и *HeapUnlock*, чтобы посторонние потоки не портили картину, создавая или удаляя блоки в просматриваемой куче.

ДИНАМИЧЕСКИ ПОДКЛЮЧАЕМЫЕ БИБЛИОТЕКИ



DLL: ОСНОВЫ

Динамически подключаемые библиотеки (dynamic-link libraries, DLL) — краеугольный камень операционной системы Windows, начиная с самой первой ее версии. В DLL содержатся все функции Windows API. Три самые важные DLL: Kernel32.dll (управление памятью, процессами и потоками), User32.dll (поддержка пользовательского интерфейса, в том числе функции, связанные с созданием окон и передачей сообщений) и GDI32.dll (графика и вывод текста).

В Windows есть и другие DLL, функции которых предназначены для более специализированных задач. Например, в AdvAPI32.dll содержатся функции для защиты объектов, работы с реестром и регистрации событий, в ComDlg32.dll — стандартные диалоговые окна (вроде File Open и File Save), а ComCtl32.dll поддерживает стандартные элементы управления.

В этой главе я расскажу, как создавать DLL-модули в Ваших приложениях. Вот лишь некоторые из причин, по которым нужно применять DLL:

- **Расширение функциональности приложения.** DLL можно загружать в адресное пространство процесса динамически, что позволяет приложению, определив, какие действия от него требуются, подгружать нужный код. Поэтому одна компания, создав какое-то приложение, может предусмотреть расширение его функциональности за счет DLL от других компаний.
- **Возможность использования разных языков программирования.** У Вас есть выбор, на каком языке писать ту или иную часть приложения. Так, пользовательский интерфейс приложения Вы скорее всего будете создавать на Microsoft Visual Basic, но прикладную логику лучше всего реализовать на C++. Программа на Visual Basic может загружать DLL, написанные на C++, Коболе, Фортране и др.
- **Более простое управление проектом.** Если в процессе разработки программного продукта отдельные его модули создаются разными группами, то при использовании DLL таким проектом управлять гораздо проще. Однако конечная версия приложения должна включать как можно меньше файлов. (Знал я одну компанию, которая поставляла свой продукт с сотней DLL. Их приложение запускалось ужасающе долго — перед началом работы ему приходилось открывать сотню файлов на диске.)
- **Экономия памяти.** Если одну и ту же DLL использует несколько приложений, в оперативной памяти может храниться только один ее экземпляр, доступный этим приложениям. Пример — DLL-версия библиотеки C/C++. Ею пользуются многие приложения. Если всех их скомпоновать со статически подключаемой версией этой библиотеки, то код таких функций, как *sprintf*, *strcpy*, *malloc* и др., будет многократно дублироваться в памяти. Но если они компонуются с DLL-версией библиотеки C/C++, в памяти будет присутствовать лишь

одна копия кода этих функций, что позволит гораздо эффективнее использовать оперативную память.

- **Разделение ресурсов.** DLL могут содержать такие ресурсы, как шаблоны диалоговых окон, строки, значки и битовые карты (растровые изображения). Эти ресурсы доступны любым программам.
- **Упрощение локализации.** DLL нередко применяются для локализации приложений. Например, приложение, содержащее только код без всяких компонентов пользовательского интерфейса, может загружать DLL с компонентами локализованного интерфейса.
- **Решение проблем, связанных с особенностями различных платформ.** В разных версиях Windows содержатся разные наборы функций. Зачастую разработчикам нужны новые функции, существующие в той версии системы, которой они пользуются. Если Ваша версия Windows не поддерживает эти функции, Вам не удастся запустить такое приложение: загрузчик попросту откажется его запускать. Но если эти функции будут находиться в отдельной DLL, Вы загрузите программу даже в более ранних версиях Windows, хотя воспользоваться ими Вы все равно не сможете.
- **Реализация специфических возможностей.** Определенная функциональность в Windows доступна только при использовании DLL. Например, отдельные виды ловушек (устанавливаемых вызовом *SetWindowsHookEx* и *SetWinEventHook*) можно задействовать при том условии, что функция уведомления ловушки размещена в DLL. Кроме того, расширение функциональности оболочки Windows возможно лишь за счет создания COM-объектов, существование которых допустимо только в DLL. Это же относится и к загружаемым Web-браузером ActiveX-элементам, позволяющим создавать Web-страницы с более богатой функциональностью.

DLL и адресное пространство процесса

Зачастую создать DLL проще, чем приложение, потому что она является лишь набором автономных функций, пригодных для использования любой программой, причем в DLL обычно нет кода, предназначенного для обработки циклов выборки сообщений или создания окон. DLL представляет собой набор модулей исходного кода, в каждом из которых содержится определенное число функций, вызываемых приложением (исполняемым файлом) или другими DLL. Файлы с исходным кодом компилируются и компонуются так же, как и при создании EXE-файла. Но, создавая DLL, Вы должны указывать компоновщику ключ /DLL. Тогда компоновщик записывает в конечный файл информацию, по которой загрузчик операционной системы определяет, что данный файл — DLL, а не приложение.

Чтобы приложение (или другая DLL) могло вызывать функции, содержащиеся в DLL, образ ее файла нужно сначала спроектировать на адресное пространство вызывающего процесса. Это достигается либо за счет неявного связывания при загрузке, либо за счет явного — в период выполнения. Подробнее о неявном связывании мы поговорим чуть позже, а о явном — в главе 20.

Как только DLL спроектирована на адресное пространство вызывающего процесса, ее функции доступны всем потокам этого процесса. Фактически библиотеки при этом теряют почти всю индивидуальность: для потоков код и данные DLL — просто дополнительные код и данные, оказавшиеся в адресном пространстве процесса. Когда поток вызывает из DLL какую-то функцию, та считывает свои параметры из стека

потока и размещает в этом стеке собственные локальные переменные. Кроме того, любые созданные кодом DLL объекты принадлежат вызывающему потоку или процессу — DLL ничем не владеет.

Например, если DLL-функция вызывает *VirtualAlloc*, резервируется регион в адресном пространстве того процесса, которому принадлежит поток, обратившийся к DLL-функции. Если DLL будет выгружена из адресного пространства процесса, зарезервированный регион не освободится, так как система не фиксирует того, что регион зарезервирован DLL-функцией. Считается, что он принадлежит процессу и поэтому освободится, только если поток этого процесса вызовет *VirtualFree* или завершится сам процесс.

Вы уже знаете, что глобальные и статические переменные EXE-файла не разделяются его параллельно выполняемыми экземплярами. В Windows 98 это достигается за счет выделения специальной области памяти для таких переменных при проецировании EXE-файла на адресное пространство процесса, а в Windows 2000 — с помощью механизма копирования при записи, рассмотренного в главе 13. Глобальные и статические переменные DLL обрабатываются точно так же. Когда какой-то процесс проецирует образ DLL-файла на свое адресное пространство, система создает также экземпляры глобальных и статических переменных.



Важно понимать, что единое адресное пространство состоит из одного исполняемого модуля и нескольких DLL-модулей. Одни из них могут быть скомпонованы со статически подключаемой библиотекой C/C++, другие — с DLL-версией той же библиотеки, а третьи (написанные не на C/C++) вообще ею не пользуются. Многие разработчики допускают ошибку, забывая, что в одном адресном пространстве может одновременно находиться несколько библиотек C/C++. Взгляните на этот код:

```
VOID EXEFunc() {
    PVOID pv = DLLFunc();
    // обращаемся к памяти, на которую указывает pv;
    // предполагаем, что pv находится в C/C++-куче EXE-файла
    free(pv);
}

PVOID DLLFunc() {
    // выделяем блок в C/C++-куче DLL
    return(malloc(100));
}
```

Ну и что Вы думаете? Будет ли этот код правильно работать? Освободит ли EXE-функция блок, выделенный DLL-функцией? Ответы на все вопросы одинаковы: может быть. Для точных ответов информации слишком мало. Если оба модуля (EXE и DLL) скомпонованы с DLL-версией библиотеки C/C++, код будет работать совершенно正常но. Но если хотя бы один из модулей связан со статической библиотекой C/C++, вызов *free* окажется неудачным. Я не раз видел, как разработчики обжигались на подобном коде.

На самом деле проблема решается очень просто: если в модуле есть функция, выделяющая память, в нем обязательно должна быть и противоположная функция, которая освобождает память. Давайте-ка перепишем предыдущий код так:

```
VOID EXEFunc() {
    PVOID pv = DLLFunc();
```

```

// обращаемся к памяти, на которую указывает pv;
// не делаем никаких предположений по поводу C/C++-кучи
DLLFreeFunc(pv);
}

PVOID DLLFunc() {
    // выделяем блок в C/C++-куче DLL
    PVOID pv = malloc(100);
    return(pv);
}

BOOL DLLFreeFunc(PVOID pv) {
    // освобождаем блок, выделенный в C/C++-куче DLL
    return(free(pv));
}

```

Этот код будет работать при любых обстоятельствах. Создавая свой модуль, не забывайте, что функции других модулей могут быть написаны на других языках, а значит, и ничего не знать о *malloc* и *free*. Не стройте свой код на подобных допущениях. Кстати, то же относится и к C++-операторам *new* и *delete*, реализованным с использованием *malloc* и *free*.

Общая картина

Попробуем разобраться в том, как работают DLL и как они используются Вами и системой. Начнем с общей картины (рис. 19-1).

Для начала рассмотрим неявное связывание EXE- и DLL-модулей. *Неявное связывание* (*implicit linking*) — самый распространенный на сегодняшний день метод. (Windows поддерживает явное связывание, но об этом — в главе 20.)

Как видно на рис. 19-1, когда некий модуль (например, EXE) обращается к функциям и переменным, находящимся в DLL, в этом процессе участвует несколько файлов и компонентов. Для упрощения будем считать, что исполняемый модуль (EXE) импортирует функции и переменные из DLL, а DLL-модули, наоборот, экспортируют их в исполняемый модуль. Но учтите, что DLL может (и это не редкость) импортировать функции и переменные из других DLL.

Собирая исполняемый модуль, который импортирует функции и переменные из DLL, Вы должны сначала создать эту DLL. А для этого нужно следующее.

1. Прежде всего Вы должны подготовить заголовочный файл с прототипами функций, структурами и идентификаторами, экспортируемыми из DLL. Этот файл включается в исходный код всех модулей Вашей DLL. Как Вы потом увидите, этот же файл понадобится и при сборке исполняемого модуля (или модулей), который использует функции и переменные из Вашей DLL.
2. Вы пишете на C/C++ модуль (или модули) исходного кода с телами функций и определениями переменных, которые должны находиться в DLL. Так как эти модули исходного кода не нужны для сборки исполняемого модуля, они могут остаться коммерческой тайной компании-разработчика.
3. Компилятор преобразует исходный код модулей DLL в OBJ-файлы (по одному на каждый модуль).
4. Компоновщик собирает все OBJ-модули в единый загрузочный DLL-модуль, в который в конечном итоге помещаются двоичный код и переменные (глобаль-

ные и статические), относящиеся к данной DLL. Этот файл потребуется при компиляции исполняемого модуля.

5. Если компоновщик обнаружит, что DLL экспортирует хотя бы одну переменную или функцию, то создаст и LIB-файл. Этот файл совсем крошечный, поскольку в нем нет ничего, кроме списка символьных имен функций и переменных, экспортимемых из DLL. Этот LIB-файл тоже понадобится при компиляции EXE-файла.

Создав DLL, можно перейти к сборке исполняемого модуля.

6. Во все модули исходного кода, где есть ссылки на внешние функции, переменные, структуры данных или идентификаторы, надо включить заголовочный файл, предоставленный разработчиком DLL.

СОЗДАНИЕ DLL

- 1) Заголовочный файл с экспортимыми прототипами, структурами и идентификаторами (символьными именами)
- 2) Исходные файлы C/C++, в которых реализованы функции и определены переменные
- 3) Компилятор создает OBJ-файл из каждого исходного файла C/C++
- 4) Компоновщик собирает DLL из OBJ-модулей
- 5) Если DLL экспортирует хотя бы одну переменную или функцию, компоновщик создает и LIB-файл

СОЗДАНИЕ EXE

- 6) Заголовочный файл с импортирумыми прототипами, структурами и идентификаторами
- 7) Исходные файлы C/C++, из которых вызываются импортируемые функции и переменные
- 8) Компилятор создает OBJ-файл из каждого исходного файла C/C++
- 9) Используя OBJ-модули и LIB-файл и учитывая ссылки на импортируемые идентификаторы, компоновщик собирает EXE-модуль (в котором также размещается таблица импорта — список необходимых DLL и импортируемых идентификаторов)

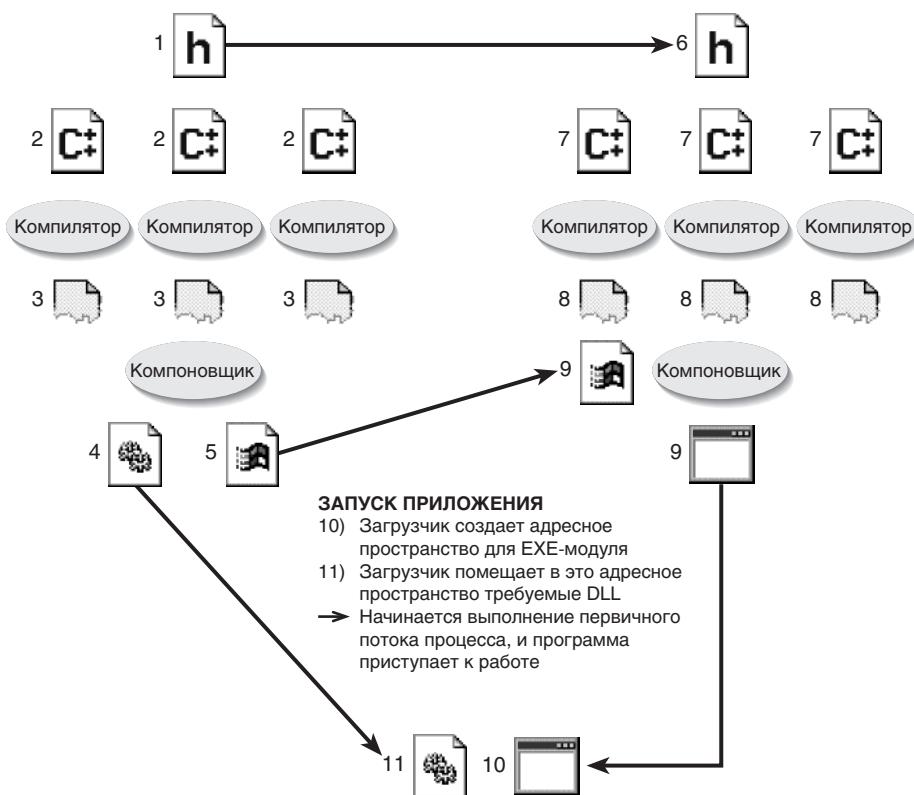


Рис. 19-1. Так DLL создается и неявно связывается с приложением

7. Вы пишете на С/C++ модуль (или модули) исходного кода с телами функций и определениями переменных, которые должны находиться в EXE-файле. Естественно, ничто не мешает Вам ссылаться на функции и переменные, определенные в заголовочном файле DLL-модуля.
8. Компилятор преобразует исходный код модулей EXE в OBJ-файлы (по одному на каждый модуль).
9. Компоновщик собирает все OBJ-модули в единый загрузочный EXE-модуль, в который в конечном итоге помещаются двоичный код и переменные (глобальные и статические), относящиеся к данному EXE. В нем также создается раздел импорта, где перечисляются имена всех необходимых DLL-модулей (информацию о разделах см. в главе 17). Кроме того, для каждой DLL в этом разделе указывается, на какие символьные имена функций и переменных ссылается двоичный код исполняемого файла. Эти сведения потребуются загрузчику операционной системы, а как именно он ими пользуется — мы узнаем чуть позже.

Создав DLL- и EXE-модули, приложение можно запустить. При его запуске загрузчик операционной системы выполняет следующие операции.

10. Загрузчик операционной системы создает виртуальное адресное пространство для нового процесса и проецирует на него исполняемый модуль.
11. Далее загрузчик анализирует раздел импорта, находит все необходимые DLL-модули и тоже проецирует на адресное пространство процесса. Заметьте, что DLL может импортировать функции и переменные из другой DLL, а значит, у нее может быть собственный раздел импорта. Заканчивая подготовку процесса к работе, загрузчик просматривает раздел импорта каждого модуля и проецирует все требуемые DLL-модули на адресное пространство этого процесса. Как видите, на инициализацию процесса может уйти довольно длительное время.

После отображения EXE- и всех DLL-модулей на адресное пространство процесса его первичный поток готов к выполнению, и приложение может начать работу. Далее мы подробно рассмотрим, как именно это происходит.

Создание DLL-модуля

Создавая DLL, Вы создаете набор функций, которые могут быть вызваны из EXE-модуля (или другой DLL). DLL может экспортить переменные, функции или C++-классы в другие модули. На самом деле я бы не советовал экспортить переменные, потому что это снижает уровень абстрагирования Вашего кода и усложняет его поддержку. Кроме того, C++-классы можно экспортить, только если импортирующие их модули транслируются тем же компилятором. Так что избегайте экспорта C++-классов, если Вы не уверены, что разработчики EXE-модулей будут пользоваться тем же компилятором.

При разработке DLL Вы сначала создаете заголовочный файл, в котором содержатся экспортимые из нее переменные (типы и имена) и функции (прототипы и имена). В этом же файле надо определить все идентификаторы и структуры данных, используемые экспортимыми функциями и переменными. Заголовочный файл включается во все модули исходного кода Вашей DLL. Более того, Вы должны поставлять его вместе со своей DLL, чтобы другие разработчики могли включать его в свои модули исходного кода, которые импортируют Ваши функции или переменные. Еди-

ный заголовочный файл, используемый при сборке DLL и любых исполняемых модулей, существенно облегчает поддержку приложения.

Вот пример единого заголовочного файла, включаемого в исходный код DLL- и EXE-модулей.

```
*****  
Модуль: MyLib.h  
*****  
  
#ifdef MYLIBAPI  
  
// MYLIBAPI должен быть определен во всех модулях исходного кода DLL  
// до включения этого файла  
  
// здесь размещаются все экспортируемые функции и переменные  
  
#else  
  
// этот заголовочный файл включается в исходный код EXE-файла;  
// указываем, что все функции и переменные импортируются  
#define MYLIBAPI extern "C" __declspec(dllimport)  
  
#endif  
  
//////////  
  
// здесь определяются все структуры данных и идентификаторы (символы)  
  
//////////  
  
// Здесь определяются экспортируемые переменные.  
// Примечание: избегайте экспортации переменных.  
MYLIBAPI int g_nResult;  
  
//////////  
  
// здесь определяются прототипы экспортируемых функций  
MYLIBAPI int Add(int nLeft, int nRight);  
  
////////// Конец файла //////////
```

Этот заголовочный файл надо включать в самое начало исходных файлов Вашей DLL следующим образом.

```
*****  
Модуль: MyLibFile1.cpp  
*****  
  
// сюда включаются стандартные заголовочные файлы Windows и библиотеки С  
#include <windows.h>  
  
// этот файл исходного кода DLL экспортирует функции и переменные  
#define MYLIBAPI extern "C" __declspec(dllexport)
```

```
// включаем экспортируемые структуры данных, идентификаторы, функции и переменные
#include "MyLib.h"

/////////////////////////////// Конец файла //////////////////////////////

// здесь размещается исходный код этой DLL
int g_nResult;

int Add(int nLeft, int nRight) {
    g_nResult = nLeft + nRight;
    return(g_nResult);
}

/////////////////////////////// Конец файла //////////////////////////////
```

При компиляции исходного файла DLL, показанного на предыдущем листинге, MYLIBAPI определяется как `_declspec(dllexport` до включения заголовочного файла MyLib.h. Такой модификатор означает, что данная переменная, функция или C++-класс экспортируется из DLL. Заметьте, что идентификатор MYLIBAPI помещен в заголовочный файл до определения экспортируемой переменной или функции.

Также обратите внимание, что в файле MyLibFile1.cpp перед экспортируемой переменной или функцией не ставится идентификатор MYLIBAPI. Он здесь не нужен: проанализировав заголовочный файл, компилятор запоминает, какие переменные и функции являются экспортируемыми.

Идентификатор MYLIBAPI включает `extern`. Пользуйтесь этим модификатором только в коде на C++, но ни в коем случае не в коде на стандартном С. Обычно компиляторы C++ искажают (mangle) имена функций и переменных, что может приводить к серьезным ошибкам при компоновке. Представьте, что DLL написана на C++, а исполняемый код — на стандартном С. При сборке DLL имя функции будет искажено, но при сборке исполняемого модуля — нет. Пытаясь скомпоновать исполняемый модуль, компоновщик сообщит об ошибке: исполняемый модуль обращается к несуществующему идентификатору. Модификатор `extern` не дает компилятору искажать имена переменных или функций, и они становятся доступными исполняемым модулем, написанным на С, C++ или любом другом языке программирования.

Теперь Вы знаете, как используется заголовочный файл в исходных файлах DLL. А как насчет исходных файлов EXE-модуля? В них MYLIBAPI определять не надо: включая заголовочный файл, Вы определяете этот идентификатор как `_declspec(dllimport`, и при компиляции исходного кода EXE-модуля компилятор поймет, что переменные и функции импортируются из DLL.

Просмотрев стандартные заголовочные файлы Windows (например, WinBase.h), Вы обнаружите, что практически тот же подход исповедует и Microsoft.

Что такое экспорт

В предыдущем разделе я упомянул о модификаторе `_declspec(dllexport`. Если он указан перед переменной, прототипом функции или C++-классом, компилятор Microsoft C/C++ встраивает в конечный OBJ-файл дополнительную информацию. Она понадобится компоновщику при сборке DLL из OBJ-файлов.

Обнаружив такую информацию, компоновщик создает LIB-файл со списком идентификаторов, экспортируемых из DLL. Этот LIB-файл нужен при сборке любого EXE-модуля, ссылающегося на такие идентификаторы. Компоновщик также вставляет в конечный DLL-файл таблицу экспортируемых идентификаторов — *раздел экспорта*,

в котором содержится список (в алфавитном порядке) идентификаторов экспортируемых функций, переменных и классов. Туда же помещается *относительный виртуальный адрес* (relative virtual address, RVA) каждого идентификатора внутри DLL-модуля.

Воспользовавшись утилитой DumpBin.exe (с ключом *-exports*) из состава Microsoft Visual Studio, мы можем увидеть содержимое раздела экспортов в DLL-модуле. Вот лишь небольшой фрагмент такого раздела для Kernel32.dll:

```
C:\WINNT\SYSTEM32>DUMPBIN -exports Kernel32.DLL
```

```
Microsoft (R) COFF Binary File Dumper Version 6.00.8168
Copyright (C) Microsoft Corp 1992-1998. All rights reserved.
```

```
Dump of file kernel32.dll
```

```
File Type: DLL
```

```
Section contains the following exports for KERNEL32.dll
```

```
0 characteristics
36DB3213 time date stamp Mon Mar 01 16:34:27 1999
0.00 version
    1 ordinal base
829 number of functions
829 number of names
```

```
ordinal hint RVA
```

```
name
```

```
1    0 0001A3C6 AddAtomA
2    1 0001A367 AddAtomW
3    2 0003F7C4 AddConsoleAliasA
4    3 0003F78D AddConsoleAliasW
5    4 0004085C AllocConsole
6    5 0002C91D AllocateUserPhysicalPages
7    6 00005953 AreFileApisANSI
8    7 0003F1A0 AssignProcessToJobObject
9    8 00021372 BackupRead
10   9 000215CE BackupSeek
11   A 00021F21 BackupWrite
```

```
:
```

```
828  33B 00003200 lstrlenA
829  33C 000040D5 lstrlenW
```

```
Summary
```

```
3000 .data
4000 .reloc
4D000 .rsrc
59000 .text
```

Как видите, идентификаторы расположены по алфавиту; в графе RVA указывается смещение в образе DLL-файла, по которому можно найти экспортируемый иденти-

фикатор. Значения в графе ordinal предназначены для обратной совместимости с исходным кодом, написанным для 16-разрядной Windows, — применять их в современных приложениях не следует. Данные из графы hint используются системой и для нас интереса не представляют.



Многие разработчики — особенно те, у кого большой опыт программирования для 16-разрядной Windows, — привыкли экспортить функции из DLL, присваивая им порядковые номера. Но Microsoft не публикует такую информацию по системным DLL и требует связывать EXE- или DLL-файлы с Windows-функциями только по именам. Используя порядковый номер, Вы рискуете тем, что Ваша программа не будет работать в других версиях Windows.

Кстати, именно это и случилось со мной. В журнале *Microsoft Systems Journal* я опубликовал программу, построенную на применении порядковых номеров. В Windows NT 3.1 программа работала прекрасно, но сбояла при запуске в Windows NT 3.5. Чтобы избавиться от сбоев, пришлось заменить порядковые номера именами функций, и все встало на свои места.

Я поинтересовался, почему Microsoft отказывается от порядковых номеров, и получил такой ответ: «Мы (Microsoft) считаем, что PE-формат позволяет сочетать преимущества порядковых номеров (быстрый поиск) с гибкостью импорта по именам. Учтите и то, что в любой момент в API могут появиться новые функции. А с порядковыми номерами в большом проекте работать очень трудно — тем более, что такие проекты многократно пересматриваются.»

Работая с собственными DLL-модулями и связывая их со своими EXE-файлами, порядковые номера использовать вполне можно. Microsoft гарантирует, что этот метод будет работоспособен даже в будущих версиях операционной системы. Но лично я стараюсь избегать порядковых номеров и отныне применяю при связывании только имена.

Создание DLL для использования с другими средствами разработки (отличными от Visual C++)

Если Вы используете Visual C++ для сборки как DLL, так и обращающегося к ней EXE-файла, то все сказанное ранее справедливо, и Вы можете спокойно пропустить этот раздел. Но если Вы создаете DLL на Visual C++, а EXE-файл — с помощью средств разработки от других поставщиков, Вам не миновать дополнительной работы.

Я уже упоминал о том, как применять модификатор *extern* при «смешанном» программировании на С и C++. Кроме того, я говорил, что из-за искажения имен нужно применять один и тот же компилятор. Даже при программировании на стандартном С инструментальные средства от разных поставщиков создают проблемы. Дело в том, что компилятор Microsoft C, экспортируя С-функцию, искажает ее имя, даже если Вы вообще не пользуетесь C++. Это происходит, только когда Ваша функция экспортируется по соглашению *_stdcall*. (Увы, это самое популярное соглашение.) Тогда компилятор Microsoft искажает имя С-функции: впереди ставит знак подчеркивания, а к концу добавляет суффикс, состоящий из символа @ и числа байтов, передаваемых функции в качестве параметров. Например, следующая функция экспортируется в таблицу экспорта DLL как *_MyFunc@8*:

```
_declspec(dllexport) LONG __stdcall MyFunc(int a, int b);
```

Если Вы решите создать EXE-файл с помощью средств разработки от другого поставщика, то компоновщик попытается скомпоновать функцию *MyFunc*, которой нет в файле DLL, созданном компилятором Microsoft, и, естественно, произойдет ошибка.

Чтобы средствами Microsoft собрать DLL, способную работать с инструментарием от другого поставщика, нужно указать компилятору Microsoft экспортировать имя функции без искажений. Сделать это можно двумя способами. Первый — создать DEF-файл для Вашего проекта и включить в него раздел EXPORTS так:

```
EXPORTS  
    MyFunc
```

Компоновщик от Microsoft, анализируя этот DEF-файл, увидит, что экспортируется надо обе функции: *_MyFunc@8* и *MyFunc*. Поскольку их имена идентичны (не считая вышеописанных искажений), компоновщик на основе информации из DEF-файла экспортирует только функцию с именем *MyFunc*, а функцию *_MyFunc@8* не экспортирует вообще.

Может, Вы подумали, что при сборке EXE-файла с такой DLL компоновщик от Microsoft, ожидая имя *_MyFunc@8*, не найдет Вашу функцию? В таком случае Вам будет приятно узнать, что компоновщик все сделает правильно и корректно скомпонует EXE-файл с функцией *MyFunc*.

Если Вам не по душе DEF-файлы, можете экспортировать неискаженное имя функции еще одним способом. Добавьте в один из файлов исходного кода DLL такую строку:

```
#pragma comment(linker, "/export:MyFunc=_MyFunc@8")
```

Тогда компилятор потребует от компоновщика экспортировать функцию *MyFunc* с той же точкой входа, что и *_MyFunc@8*. Этот способ менее удобен, чем первый, так как здесь приходится самостоятельно вставлять дополнительную директиву с искаженным именем функции. И еще один минус этого способа в том, что из DLL экспортируются два идентификатора одной и той же функции: *MyFunc* и *_MyFunc@8*, тогда как при первом способе — только идентификатор *MyFunc*. По сути, второй способ не имеет особых преимуществ перед первым — он просто избавляет от DEF-файла.

Создание EXE-модуля

Вот пример исходного кода EXE-модуля, который импортирует идентификаторы, экспортируемые DLL, и ссылается на них в процессе выполнения.

```
/*********************************************  
Модуль: MyExeFile1.cpp  
*****  
  
// сюда включаются стандартные заголовочные файлы Windows и библиотеки C  
#include <windows.h>  
  
// включаем экспортируемые структуры данных, идентификаторы, функции и переменные  
#include "MyLib\MyLib.h"  
  
//////////  
  
int WINAPI WinMain(HINSTANCE hinstExe, HINSTANCE, LPTSTR pszCmdLine, int) {
```

```

int nLeft = 10, nRight = 25;

TCHAR sz[100];
wsprintf(sz, TEXT("%d + %d = %d"), nLeft, nRight, Add(nLeft, nRight));
MessageBox(NULL, sz, TEXT("Calculation"), MB_OK);

wsprintf(sz, TEXT("The result from the last Add is: %d"), g_nResult);
MessageBox(NULL, sz, TEXT("Last Result"), MB_OK);
return(0);
}

//////////////////////////// Конец файла ///////////////////////////////

```

Создавая файлы исходного кода для EXE-модуля, Вы должны включить в них заголовочный файл DLL, иначе импортируемые идентификаторы окажутся неопределенными, и компилятор выдаст массу предупреждений и сообщений об ошибках.

MYLIBAPI в исходных файлах EXE-модуля до заголовочного файла DLL не определяется. Поэтому при компиляции приведенного выше кода MYLIBAPI за счет заголовочного файла MyLib.h будет определен как `__declspec(dllimport)`. Встречая такой модификатор перед именем переменной, функции или C++-класса, компилятор понимает, что данный идентификатор импортируется из какого-то DLL-модуля. Из какого именно, ему не известно, да это его и не интересует. Компилятору нужно лишь убедиться в корректности обращения к импортируемым идентификаторам.

Далее компоновщик собирает все OBJ-модули в конечный EXE-модуль. Для этого он должен знать, в каких DLL содержатся импортируемые идентификаторы, на которые есть ссылки в коде. Информацию об этом он получает из передаваемого ему LIB-файла. Я уже говорил, что этот файл — просто список идентификаторов, экспортируемых DLL. Компоновщик должен удостовериться в существовании идентификатора, на который Вы ссылаетесь в коде, и узнать, в какой DLL он находится. Если компоновщик сможет разрешить все ссылки на внешние идентификаторы, на свет появится EXE-модуль.

Что такое импорт

В предыдущем разделе я упомянул о модификаторе `__declspec(dllimport)`. Импортируя идентификатор, необязательно прибегать к `__declspec(dllimport)` — можно использовать стандартное ключевое слово `extern` языка С. Но компилятор создаст чуть более эффективный код, если ему будет заранее известно, что идентификатор, на который мы ссылаемся, импортируется из LIB-файла DLL-модуля. Вот почему я настоятельно рекомендую пользоваться ключевым словом `__declspec(dllimport)` для импортируемых функций и идентификаторов данных. Именно его подставляет за Вас операционная система, когда Вы вызываете любую из стандартных Windows-функций.

Разрешая ссылки на импортируемые идентификаторы, компоновщик создает в конечном EXE-модуле *раздел импорта* (*imports section*). В нем перечисляются DLL, необходимые этому модулю, и идентификаторы, на которые есть ссылки из всех используемых DLL.

Воспользовавшись утилитой DumpBin.exe (с ключом `-imports`), мы можем увидеть содержимое раздела импорта. Ниже показан фрагмент полученной с ее помощью таблицы импорта Calc.exe.

```
C:\WINNT\SYSTEM32>DUMPBIN -imports Calc.EXE

Microsoft (R) COFF Binary File Dumper Version 6.00.8168
Copyright (C) Microsoft Corp 1992-1998. All rights reserved.

Dump of file calc.exe

File Type: EXECUTABLE IMAGE

Section contains the following imports:

SHELL32.dll
    10010F4 Import Address Table
    1012820 Import Name Table
    FFFFFFFF time date stamp
    FFFFFFFF Index of first forwarder reference

    77C42983    7A  ShellAboutW

MSVCRT.dll
    1001094 Import Address Table
    10127C0 Import Name Table
    FFFFFFFF time date stamp
    FFFFFFFF Index of first forwarder reference

    78010040    295  memmove
    78018124    42   _EH_prolog
    78014C34    2D1  toupper
    78010F6E    2DD  wcschr
    78010668    2E3  wcslen

    ::

ADVAPI32.dll
    1001000 Import Address Table
    101272C Import Name Table
    FFFFFFFF time date stamp
    FFFFFFFF Index of first forwarder reference

    779858F4    19A  RegQueryValueExA
    77985196    190  RegOpenKeyExA
    77984BA1    178  RegCloseKey

KERNEL32.dll
    100101C Import Address Table
    1012748 Import Name Table
    FFFFFFFF time date stamp
    FFFFFFFF Index of first forwarder reference

    77ED4134    336  lstrcpyW
    77ED33E8    1E5  LocalAlloc
    77EDEF36    DB   GetCommandLineW
    77ED1610    15E  GetProfileIntW
    77ED4BA4    1EC  LocalReAlloc
```

:

```
Header contains the following bound import information:
  Bound to SHELL32.dll [36E449E0] Mon Mar 08 14:06:24 1999
  Bound to MSVCRT.dll [36BB8379] Fri Feb 05 15:49:13 1999
  Bound to ADVAPI32.dll [36E449E1] Mon Mar 08 14:06:25 1999
  Bound to KERNEL32.dll [36DDAD55] Wed Mar 03 13:44:53 1999
  Bound to GDI32.dll [36E449E0] Mon Mar 08 14:06:24 1999
  Bound to USER32.dll [36E449E0] Mon Mar 08 14:06:24 1999
```

Summary

```
2000 .data
3000 .rsrc
13000 .text
```

Как видите, в разделе есть записи по каждой DLL, необходимой Calc.exe: Shell32.dll, MSVCRT.dll, AdvAPI32.dll, Kernel32.dll, GDI32.dll и User32.dll. Под именем DLL-модуля выводится список идентификаторов, импортируемых программой Calc.exe. Например, Calc.exe обращается к следующим функциям из Kernel32.dll: *lstrcpyW, LocalAlloc, GetCommandLineW, GetProfileIntW* и др.

Число слева от импортируемого идентификатора называется «подсказкой» (hint) и для нас несущественно. Крайнее левое число в строке для идентификатора сообщает адрес, по которому он размещен в адресном пространстве процесса. Такой адрес показывается, только если было проведено связывание (binding) исполняемого модуля, но об этом — в главе 20.

Выполнение EXE-модуля

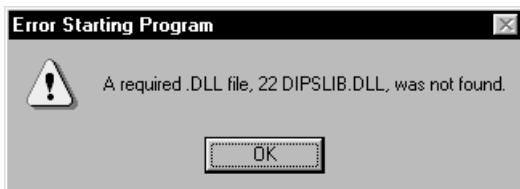
При запуске EXE-файла загрузчик операционной системы создает для его процесса виртуальное адресное пространство и проецирует на него исполняемый модуль. Далее загрузчик анализирует раздел импорта и пытается спроектировать все необходимые DLL на адресное пространство процесса.

Поскольку в разделе импорта указано только имя DLL (без пути), загрузчику придется самому искать ее на дисковых устройствах в компьютере пользователя. Поиск DLL осуществляется в следующей последовательности.

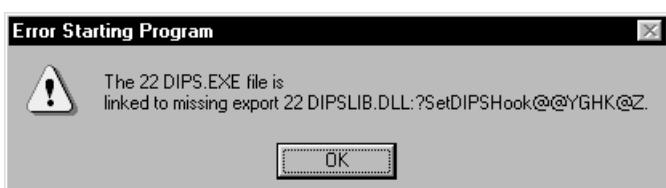
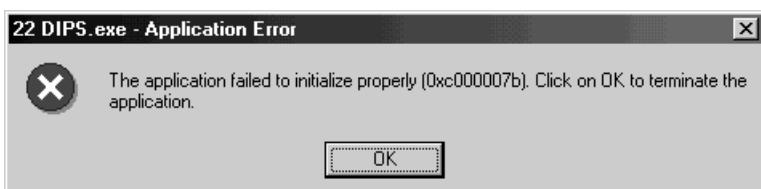
1. Каталог, содержащий EXE-файл.
2. Текущий каталог процесса.
3. Системный каталог Windows.
4. Основной каталог Windows.
5. Каталоги, указанные в переменной окружения PATH.

Учтите, что на процесс поиска библиотек могут повлиять и другие факторы (см. главу 20). Проецируя DLL-модули на адресное пространство, загрузчик проверяет в каждом из них раздел импорта. Если у DLL есть раздел импорта (что обычно и бывает), загрузчик проецирует следующий DLL-модуль. При этом загрузчик ведет учет загружаемых DLL и проецирует их только один раз, даже если загрузки этих DLL требуют и другие модули.

Если найти файл DLL не удается, загрузчик выводит одно из двух сообщений (первое — в Windows 2000, а второе — в Windows 98).



Найдя и спроектировав на адресное пространство процесса все необходимые DLL-модули, загрузчик настраивает ссылки на импортируемые идентификаторы. Для этого он вновь просматривает разделы импорта в каждом модуле, проверяя наличие указанного идентификатора в соответствующей DLL. Не обнаружив его (что происходит крайне редко), загрузчик выводит одно из двух сообщений (первое — в Windows 2000, а второе — в Windows 98):



Было бы неплохо, если бы в версии этого окна для Windows 2000 сообщалось имя недостающей функции, а не маловразумительный для пользователя код ошибки вроде 0xC000007B. Ну да ладно, может, в следующей версии Windows это будет исправлено.

Если же идентификатор найден, загрузчик отыскивает его RVA и прибавляет к виртуальному адресу, по которому данная DLL размещена в адресном пространстве процесса, а затем сохраняет полученный виртуальный адрес в разделе импорта EXE-модуля. И с этого момента ссылка в коде на импортируемый идентификатор приводит к выборке его адреса из раздела импорта вызывающего модуля, открывая таким образом доступ к импортируемой переменной, функции или функции-члену C++-класса. Вот и все — динамические связи установлены, первичный поток процесса начал выполняться, и приложение наконец-то работает!

Естественно, загрузка всех этих DLL и настройка ссылок занимает какое-то время. Но, поскольку такие операции выполняются лишь при запуске процесса, на производительности приложения это не сказывается. Тем не менее для многих программ подобная задержка при инициализации неприемлема. Чтобы сократить время загруз-

ки приложения, Вы должны модифицировать базовые адреса своих EXE- и DLL-модулей и провести их (модулей) связывание. Увы, лишь немногие разработчики знают, как это делается, хотя эти приемы очень важны. Если бы ими пользовались все компании-разработчики, система работала бы куда быстрее. Я даже считаю, что операционную систему нужно поставлять с утилитой, позволяющей автоматически выполнять эти операции. О модификации базовых адресов модулей и о связывании я расскажу в следующей главе.

DLL: более сложные методы программирования

В предыдущей главе мы говорили в основном о неявном связывании, поскольку это самый популярный метод. Представленной там информации вполне достаточно для создания большинства приложений. Однако DLL открывают нам гораздо больше возможностей, и в этой главе Вас ждет целый «букет» новых методов, относящихся к программированию DLL. Во многих приложениях эти методы скорее всего не понадобятся, тем не менее они очень полезны, и познакомиться с ними стоит. Я бы посоветовал, как минимум, прочесть разделы «Модификация базовых адресов модулей» и «Связывание модулей»; подходы, изложенные в них, помогут существенно повысить быстродействие всей системы.

Явная загрузка DLL и связывание идентификаторов

Чтобы поток мог вызвать функцию из DLL-модуля, последний надо спроектировать на адресное пространство процесса, которому принадлежит этот поток. Делается это двумя способами. Первый состоит в том, что код Вашего приложения просто ссылается на идентификаторы, содержащиеся в DLL, и тем самым заставляет загрузчик явно загружать (и связывать) нужную DLL при запуске приложения.

Второй способ — явная загрузка и связывание требуемой DLL в период выполнения приложения. Иначе говоря, его поток явно загружает DLL в адресное пространство процесса, получает виртуальный адрес необходимой DLL-функции и вызывает ее по этому адресу. Изящество такого подхода в том, что все происходит в уже выполняемом приложении.

На рис. 20-1 показано, как приложение явно загружает DLL и связывается с ней.

Явная загрузка DLL

В любой момент поток может спроектировать DLL на адресное пространство процесса, вызвав одну из двух функций:

```
HINSTANCE LoadLibrary(PCTSTR pszDLLPathName);  
  
HINSTANCE LoadLibraryEx(  
    PCTSTR pszDLLPathName,  
    HANDLE hFile,  
    DWORD dwFlags);
```

Обе функции ищут образ DLL-файла (в каталогах, список которых приведен в предыдущей главе) и пытаются спроектировать его на адресное пространство вызывающего процесса. Значение типа HINSTANCE, возвращаемое этими функциями, со-

общает адрес виртуальной памяти, по которому спроектирован образ файла. Если спроектировать DLL на адресное пространство процесса не удалось, функции возвращают NULL. Дополнительную информацию об ошибке можно получить вызовом *GetLastError*.

Очевидно, Вы обратили внимание на два дополнительных параметра функции *LoadLibraryEx*: *bFile* и *dwFlags*. Первый зарезервирован для использования в будущих версиях и должен быть NULL. Во втором можно передать либо 0, либо комбинацию флагов *DONT_RESOLVE_DLL_REFERENCES*, *LOAD_LIBRARY_AS_DATAFILE* и *LOAD_WITH_ALTERED_SEARCH_PATH*, о которых мы сейчас и поговорим.

СОЗДАНИЕ DLL

- 1) Заголовочный файл с экспортируемыми прототипами, структурами и идентификаторами (символьными именами)
- 2) Исходные файлы С/C++, в которых реализованы экспортируемые функции и определены переменные
- 3) Компилятор создает OBJ-файл из каждого исходного файла С/C++
- 4) Компоновщик собирает DLL из OBJ-модулей
- 5) Если DLL экспортирует хотя бы одну переменную или функцию, компоновщик создает LIB-файл (при явном связывании этот файл не используется)

СОЗДАНИЕ EXE

- 6) Заголовочный файл с импортируемыми прототипами, структурами и идентификаторами
- 7) Исходные файлы С/C++, в которых нет ссылок на импортируемые функции и переменные
- 8) Компилятор создает OBJ-файл из каждого исходного файла С/C++
- 9) Компоновщик собирает EXE-модуль из OBJ-модулей (LIB-файл DLL не нужен, так как нет прямых ссылок на экспортируемые идентификаторы; раздел импорта в EXE-модуле отсутствует)

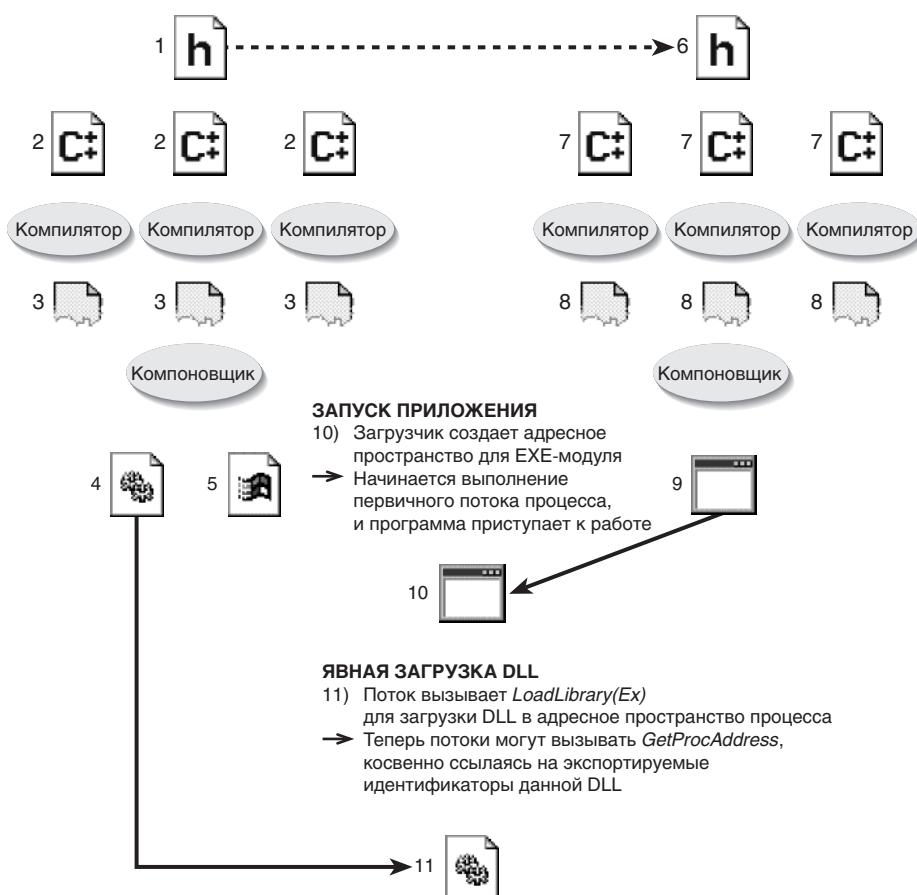


Рис. 20-1. Так DLL создается и явно связывается с приложением

DONT_RESOLVE_DLL_REFERENCES

Этот флаг указывает системе спроектировать DLL на адресное пространство вызывающего процесса. Проектируя DLL, система обычно вызывает из нее специальную функцию *DllMain* (о ней — чуть позже) и с ее помощью инициализирует библиотеку. Так вот, данный флаг заставляет систему проектировать DLL, не обращаясь к *DllMain*.

Кроме того, DLL может импортировать функции из других DLL. При загрузке библиотеки система проверяет, использует ли она другие DLL; если да, то загружает и их. При установке флага DONT_RESOLVE_DLL_REFERENCES дополнительные DLL автоматически не загружаются.

LOAD_LIBRARY_AS_DATAFILE

Этот флаг очень похож на предыдущий. DLL проектируется на адресное пространство процесса так, будто это файл данных. При этом система не тратит дополнительное время на подготовку к выполнению какого-либо кода из данного файла. Например, когда DLL проектируется на адресное пространство, система считывает информацию из DLL-файла и на ее основе определяет, какие атрибуты защиты страниц следует присвоить разным частям файла. Если флаг LOAD_LIBRARY_AS_DATAFILE не указан, атрибуты защиты устанавливаются такими, чтобы код из данного файла можно было выполнять.

Этот флаг может понадобиться по нескольким причинам. Во-первых, его стоит указать, если DLL содержит только ресурсы и никаких функций. Тогда DLL проектируется на адресное пространство процесса, после чего при вызове функций, загружающих ресурсы, можно использовать значение HINSTANCE, возвращенное функцией *LoadLibraryEx*. Во-вторых, он пригодится, если Вам нужны ресурсы, содержащиеся в каком-нибудь EXE-файле. Обычно загрузка такого файла приводит к запуску нового процесса, но этого не произойдет, если его загрузить вызовом *LoadLibraryEx* в адресное пространство Вашего процесса. Получив значение HINSTANCE для спроектированного EXE-файла, Вы фактически получаете доступ к его ресурсам. Так как в EXE-файле нет *DllMain*, при вызове *LoadLibraryEx* для загрузки EXE-файла нужно указать флаг LOAD_LIBRARY_AS_DATAFILE.

LOAD_WITH_ALTERED_SEARCH_PATH

Этот флаг изменяет алгоритм, используемый *LoadLibraryEx* при поиске DLL-файла. Обычно поиск осуществляется так, как я рассказывал в главе 19. Однако, если данный флаг установлен, функция ищет файл, просматривая каталоги в таком порядке:

1. Каталог, заданный в параметре *pszDLLPathName*.
2. Текущий каталог процесса.
3. Системный каталог Windows.
4. Основной каталог Windows.
5. Каталоги, перечисленные в переменной окружения PATH.

Явная выгрузка DLL

Если необходимость в DLL отпадает, ее можно выгрузить из адресного пространства процесса, вызвав функцию:

```
BOOL FreeLibrary(HINSTANCE hinstDll);
```

Вы должны передать в *FreeLibrary* значение типа HINSTANCE, которое идентифицирует выгружаемую DLL. Это значение Вы получаете после вызова *LoadLibrary(Ex)*.

DLL можно выгрузить и с помощью другой функции:

```
VOID FreeLibraryAndExitThread(
    HINSTANCE hinstDll,
    DWORD dwExitCode);
```

Она реализована в Kernel32.dll так:

```
VOID FreeLibraryAndExitThread(HINSTANCE hinstDll, DWORD dwExitCode) {
    FreeLibrary(hinstDll);
    ExitThread(dwExitCode);
}
```

На первый взгляд, в ней нет ничего особенного, и Вы, наверное, удивляйтесь, с чего это Microsoft решила ее написать. Но представьте такой сценарий. Вы пишете DLL, которая при первом отображении на адресное пространство процесса создает поток. Последний, закончив свою работу, отключает DLL от адресного пространства процесса и завершается, вызывая сначала *FreeLibrary*, а потом *ExitThread*.

Если поток станет сам вызывать *FreeLibrary* и *ExitThread*, возникнет очень серьезная проблема: *FreeLibrary* тут же отключит DLL от адресного пространства процесса. После возврата из *FreeLibrary* код, содержащий вызов *ExitThread*, окажется недоступен, и поток попытается выполнить не известно что. Это приведет к нарушению доступа и завершению всего процесса!

С другой стороны, если поток обратится к *FreeLibraryAndExitThread*, она вызовет *FreeLibrary*, и та сразу же отключит DLL. Но следующая исполняемая инструкция находится в Kernel32.dll, а не в только что отключенной DLL. Значит, поток сможет продолжить выполнение и вызвать *ExitThread*, которая корректно завершит его, не возвращая управления.

Впрочем, *FreeLibraryAndExitThread* может и не понадобиться. Мне она пригодилась лишь раз, когда я занимался весьма нетипичной задачей. Да и код я писал под Windows NT 3.1, где этой функции не было. Наверное, поэтому я так обрадовался, обнаружив ее в более новых версиях Windows.

На самом деле *LoadLibrary* и *LoadLibraryEx* лишь увеличивают счетчик числа пользователей указанной библиотеки, а *FreeLibrary* и *FreeLibraryAndExitThread* его уменьшают. Так, при первом вызове *LoadLibrary* для загрузки DLL система проецирует образ DLL-файла на адресное пространство вызывающего процесса и присваивает единицу счетчику числа пользователей этой DLL. Если поток того же процесса вызывает *LoadLibrary* для той же DLL еще раз, DLL больше не проецируется; система просто увеличивает счетчик числа ее пользователей — вот и все.

Чтобы выгрузить DLL из адресного пространства процесса, *FreeLibrary* придется теперь вызывать дважды: первый вызов уменьшит счетчик до 1, второй — до 0. Обнаружив, что счетчик числа пользователей DLL обнулен, система отключит ее. После этого попытка вызова какой-либо функции из данной DLL приведет к нарушению доступа, так как код по указанному адресу уже не отображается на адресное пространство процесса.

Система поддерживает в каждом процессе свой счетчик DLL, т. е. если поток процесса A вызывает приведенную ниже функцию, а затем тот же вызов делает поток в процессе B, то MyLib.dll проецируется на адресное пространство обоих процессов, а счетчики числа пользователей DLL в каждом из них приравниваются 1.

```
HINSTANCE hinstDll = LoadLibrary("MyLib.dll");
```

Если же поток процесса B вызовет далее:

```
FreeLibrary(hinstDll);
```

счетчик числа пользователей DLL в процессе В обнуляется, что приведет к отключению DLL от адресного пространства процесса В. Но проекция DLL на адресное пространство процесса А не затрагивается, и счетчик числа пользователей DLL в нем остается прежним.

Чтобы определить, спроектирована ли DLL на адресное пространство процесса, поток может вызвать функцию *GetModuleHandle*:

```
HINSTANCE GetModuleHandle(PCTSTR pszModuleName);
```

Например, следующий код загружает MyLib.dll, только если она еще не спроектирована на адресное пространство процесса:

```
HINSTANCE hinstDll = GetModuleHandle("MyLib"); // подразумевается расширение .dll
if (hinstDll == NULL) {
    hinstDll = LoadLibrary("MyLib"); // подразумевается расширение .dll
}
```

Если у Вас есть значение HINSTANCE для DLL, можно определить полное (вместе с путем) имя DLL или EXE с помощью *GetModuleFileName*:

```
DWORD GetModuleFileName(
    HINSTANCE hinstModule,
    PTSTR pszPathName,
    DWORD cchPath);
```

Первый параметр этой функции — значение типа HINSTANCE нужной DLL (или EXE). Второй параметр, *pszPathName*, задает адрес буфера, в который она запишет полное имя файла. Третий, и последний, параметр (*cchPath*) определяет размер буфера в символах.

Явное подключение экспортируемого идентификатора

Поток получает адрес экспортируемого идентификатора из явно загруженной DLL вызовом *GetProcAddress*:

```
FARPROC GetProcAddress(
    HINSTANCE hinstDll,
    PCSTR pszSymbolName);
```

Параметр *hinstDll* — описатель, возвращенный *LoadLibrary(Ex)* или *GetModuleHandle* и относящийся к DLL, которая содержит нужный идентификатор. Параметр *pszSymbolName* разрешается указывать в двух формах. Во-первых, как адрес строки с нулевым символом в конце, содержащей имя интересующей Вас функции:

```
FARPROC pfn = GetProcAddress(hinstDll, "SomeFuncInDll");
```

Заметьте: тип параметра *pszSymbolName* — PCSTR, а не PCTSTR. Это значит, что функция *GetProcAddress* принимает только ANSI-строки — ей нельзя передать Unicode-строку. А причина в том, что идентификаторы функций и переменных в разделе экспорта DLL всегда хранятся как ANSI-строки.

Вторая форма параметра *pszSymbolName* позволяет указывать порядковый номер нужной функции:

```
FARPROC pfn = GetProcAddress(hinstDll, MAKEINTRESOURCE(2));
```

Здесь подразумевается, что Вам известен порядковый номер (2) искомого идентификатора, присвоенный ему автором данной DLL. И вновь повторю, что Microsoft

настоятельно не рекомендует пользоваться порядковыми номерами; поэтому Вы редко встретите второй вариант вызова *GetProcAddress*.

При любом способе Вы получаете адрес содержащегося в DLL идентификатора. Если идентификатор не найден, *GetProcAddress* возвращает NULL.

Учтите, что первый способ медленнее, так как системе приходится проводить поиск и сравнение строк. При втором способе, если Вы передаете порядковый номер, не присвоенный ни одной из экспортруемых функций, *GetProcAddress* может вернуть значение, отличное от NULL. В итоге Ваша программа, ничего не подозревая, получит неправильный адрес. Попытка вызова функции по этому адресу почти наверняка приведет к нарушению доступа. Я и сам — когда только начинал программировать под Windows и не очень четко понимал эти вещи — несколько раз попадал в эту ловушку. Так что будьте внимательны. (Вот Вам, кстати, и еще одна причина, почему от использования порядковых номеров следует отказаться в пользу символьных имен — идентификаторов.)

Функция входа/выхода

В DLL может быть лишь одна функция входа/выхода. Система вызывает ее в некоторых ситуациях (о чем речь еще впереди) сугубо в информационных целях, и обычно она используется DLL для инициализации и очистки ресурсов в конкретных процессах или потоках. Если Вашей DLL подобные уведомления не нужны, Вы не обязаны реализовывать эту функцию. Пример — DLL, содержащая только ресурсы. Но если же уведомления необходимы, функция должна выглядеть так:

```
BOOL WINAPI DllMain(HINSTANCE hinstDll, DWORD fdwReason, PVOID fImpLoad) {  
  
    switch (fdwReason) {  
        case DLL_PROCESS_ATTACH:  
            // DLL проецируется на адресное пространство процесса  
            break;  
  
        case DLL_THREAD_ATTACH:  
            // создается поток  
            break;  
  
        case DLL_THREAD_DETACH:  
            // поток корректно завершается  
            break;  
  
        case DLL_PROCESS_DETACH:  
            // DLL отключается от адресного пространства процесса  
            break;  
    }  
    return(TRUE); // используется только для DLL_PROCESS_ATTACH  
}
```



При вызове *DllMain* надо учитывать регистр букв. Многие случайно вызывают *DLLMain*, и это вполне объяснимо: термин *DLL* обычно пишется заглавными буквами. Если Вы назовете функцию входа/выхода не *DllMain*, а как-то иначе (пусть даже только один символ будет набран в другом регистре), компиляция и компоновка Вашего кода пройдет без проблем, но система проигнорирует такую функцию входа/выхода, и Ваша DLL никогда не будет инициализирована.

Параметр *binstDll* содержит описатель экземпляра DLL. Как и *binstExe* функции *(w)WinMain*, это значение — виртуальный адрес проекции файла DLL на адресное пространство процесса. Обычно последнее значение сохраняется в глобальной переменной, чтобы его можно было использовать и при вызовах функций, загружающих ресурсы (типа *DialogBox* или *LoadString*). Последний параметр, *fImpLoad*, отличен от 0, если DLL загружена неявно, и равен 0, если она загружена явно.

Параметр *fdwReason* сообщает о причине, по которой система вызвала эту функцию. Он принимает одно из четырех значений: *DLL_PROCESS_ATTACH*, *DLL_PROCESS_DETACH*, *DLL_THREAD_ATTACH* или *DLL_THREAD_DETACH*. Мы рассмотрим их в следующих разделах.



Не забывайте, что DLL инициализируют себя, используя функции *DllMain*. К моменту выполнения Вашей *DllMain* другие DLL в том же адресном пространстве могут не успеть выполнить свои функции *DllMain*, т. е. они окажутся неинициализированными. Поэтому Вы должны избегать обращений из *DllMain* к функциям, импортируемым из других DLL. Кроме того, не вызывайте из *DllMain* функции *LoadLibrary(Ex)* и *FreeLibrary*, так как это может привести к взаимной блокировке.

В документации Platform SDK утверждается, что *DllMain* должна выполнять лишь простые виды инициализации — настройку локальной памяти потока (см. главу 21), создание объектов ядра, открытие файлов и т. д. Избегайте обращений к функциям, связанным с User, Shell, ODBC, COM, RPC и сокетами (а также к функциям, которые их вызывают), потому что соответствующие DLL могут быть еще не инициализированы. Кроме того, подобные функции могут вызывать *LoadLibrary(Ex)* и тем самым приводить к взаимной блокировке.

Аналогичные проблемы возможны и при создании глобальных или статических C++-объектов, поскольку их конструктор или деструктор вызывается в то же время, что и Ваша *DllMain*.

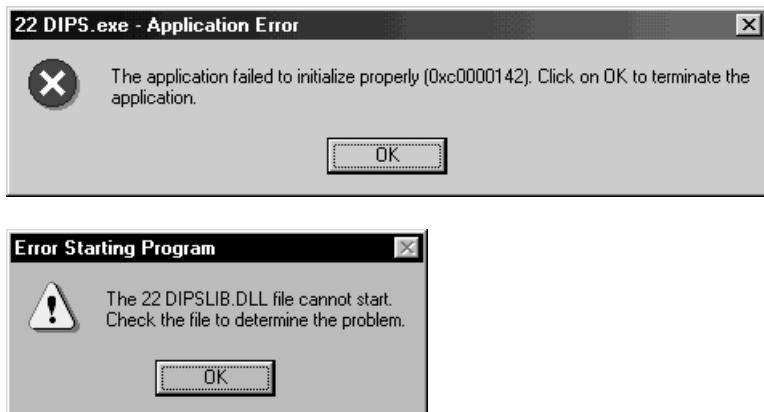
Уведомление *DLL_PROCESS_ATTACH*

Система вызывает *DllMain* с этим значением параметра *fdwReason* сразу после того, как DLL спроектирована на адресное пространство процесса. А это происходит, только когда образ DLL-файла проецируется в первый раз. Если затем поток вызовет *LoadLibrary(Ex)* для уже спроектированной DLL, система просто увеличит счетчик числа пользователей этой DLL; так что *DllMain* вызывается со значением *DLL_PROCESS_ATTACH* лишь раз.

Обрабатывая *DLL_PROCESS_ATTACH*, библиотека должна выполнить в процессе инициализацию, необходимую ее функциям. Например, в DLL могут быть функции, которым нужна своя куча (создаваемая в адресном пространстве процесса). В этом случае *DllMain* могла бы создать такую кучу, вызвав *HeapCreate* при обработке уведомления *DLL_PROCESS_ATTACH*, а описатель созданной кучи сохранить в глобальной переменной, доступной функциям DLL.

При обработке уведомления *DLL_PROCESS_ATTACH* значение, возвращаемое функцией *DllMain*, указывает, корректно ли прошла инициализация DLL. Например, если вызов *HeapCreate* закончился благополучно, следует вернуть *TRUE*. А если кучу создать не удалось — *FALSE*. Для любых других значений *fdwReason* — *DLL_PROCESS_DETACH*, *DLL_THREAD_ATTACH* или *DLL_THREAD_DETACH* — значение, возвращаемое *DllMain*, системой игнорируется.

Конечно, где-то в системе должен быть поток, отвечающий за выполнение кода *DllMain*. При создании нового процесса система выделяет для него адресное пространство, куда проецируется EXE-файл и все необходимые ему DLL-модули. Далее создается первичный поток процесса, используемый системой для вызова *DllMain* из каждой DLL со значением *DLL_PROCESS_ATTACH*. Когда все спроектированные DLL отвечают на это уведомление, система заставит первичный поток процесса выполнить стартовый код из библиотеки C/C++, а потом — входную функцию EXE-файла (*main*, *wmain*, *WinMain* или *wWinMain*). Если *DllMain* хотя бы одной из DLL вернет FALSE, сообщая об ошибке при инициализации, система завершит процесс, удалив из его адресного пространства образы всех файлов; после этого пользователь увидит окно с сообщением о том, что процесс запустить не удалось. Ниже показаны соответствующие окна для Windows 2000 и Windows 98.



Теперь посмотрим, что происходит при явной загрузке DLL. Когда поток вызывает *LoadLibrary(Ex)*, система отыскивает указанную DLL и проецирует ее на адресное пространство процесса. Затем вызывает *DllMain* со значением *DLL_PROCESS_ATTACH*, используя поток, вызвавший *LoadLibrary(Ex)*. Как только *DllMain* обработает уведомление, произойдет возврат из *LoadLibrary(Ex)*, и поток продолжит работу в обычном режиме. Если же *DllMain* вернет FALSE (неудачная инициализация), система автоматически отключит образ файла DLL от адресного пространства процесса, а вызов *LoadLibrary(Ex)* даст NULL.

Уведомление *DLL_PROCESS_DETACH*

При отключении DLL от адресного пространства процесса вызывается ее функция *DllMain* со значением *DLL_PROCESS_DETACH* в параметре *fdwReason*. Обрабатывая это значение, DLL должна провести очистку в данном процессе. Например, вызвать *HeapDestroy*, чтобы разрушить кучу, созданную ею при обработке уведомления *DLL_PROCESS_ATTACH*. Обратите внимание: если функция *DllMain* вернула FALSE, получив уведомление *DLL_PROCESS_ATTACH*, то ее нельзя вызывать с уведомлением *DLL_PROCESS_DETACH*. Если DLL отключается из-за завершения процесса, то за выполнение кода *DllMain* отвечает поток, вызвавший *ExitProcess* (обычно это первичный поток приложения). Когда Ваша входная функция возвращает управление стартовому коду из библиотеки C/C++, тот явно вызывает *ExitProcess* и завершает процесс.

Если DLL отключается в результате вызова *FreeLibrary* или *FreeLibraryAndExitThread*, код *DllMain* выполняется потоком, вызвавшим одну из этих функций. В случае обращения к *FreeLibrary* управление не возвращается, пока *DllMain* не закончит обработку уведомления *DLL_PROCESS_DETACH*.

Учтите также, что DLL может помешать завершению процесса, если, например, ее *DllMain* входит в бесконечный цикл, получив уведомление *DLL_PROCESS_DETACH*. Операционная система уничтожает процесс только после того, как все DLL-модули обработают уведомление *DLL_PROCESS_DETACH*.

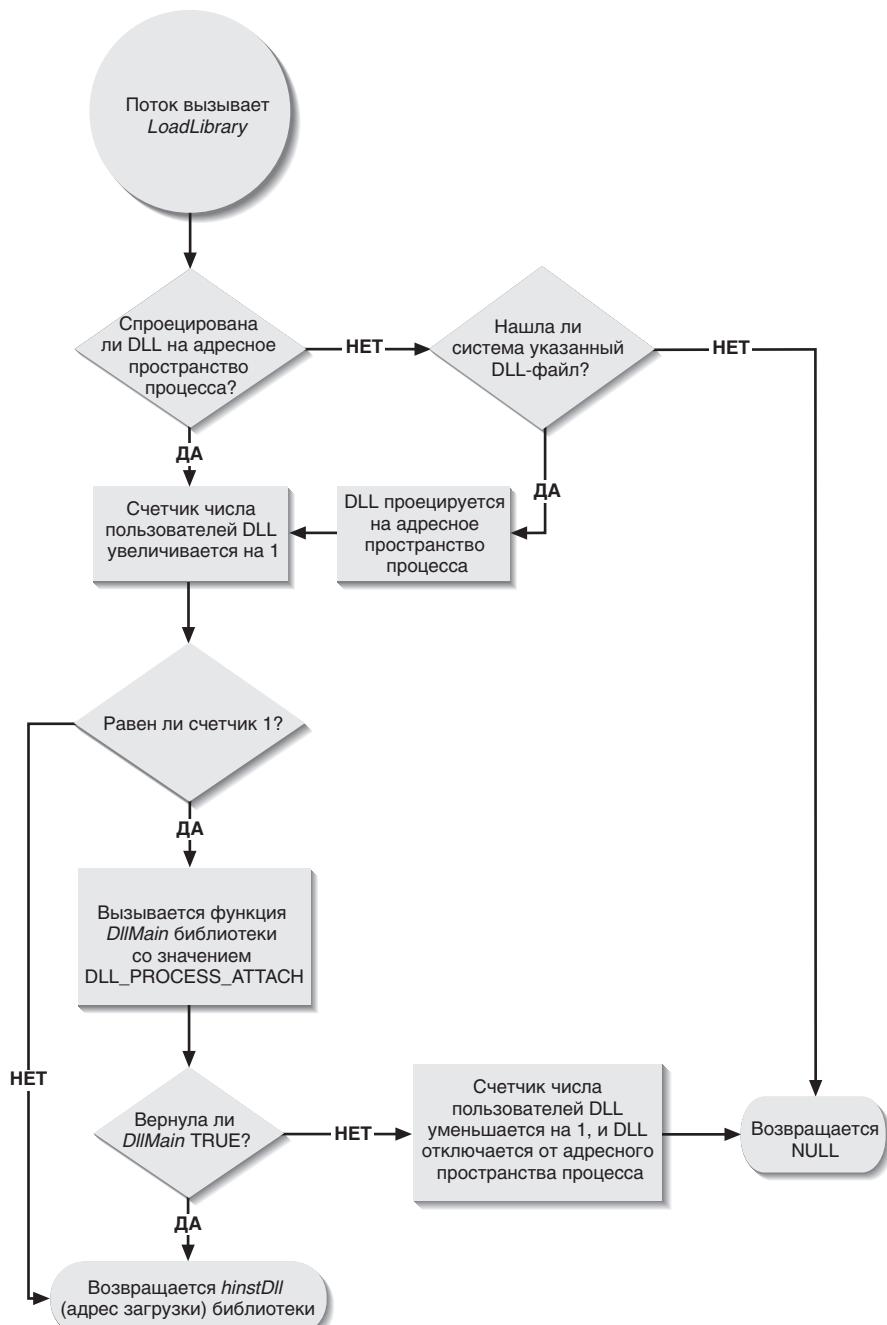


Рис. 20-2. Операции, выполняемые системой при вызове потоком функции *LoadLibrary*



Если процесс завершается в результате вызова *TerminateProcess*, система не вызывает *DllMain* со значением *DLL_PROCESS_DETACH*. А значит, ни одна DLL, спроектированная на адресное пространство процесса, не получит шанса на очистку до завершения процесса. Последствия могут быть плачевны — вплоть до потери данных. Вызывайте *TerminateProcess* только в самом крайнем случае!

На рис. 20-2 показаны операции, выполняемые при вызове *LoadLibrary*, а на рис. 20-3 — при вызове *FreeLibrary*.

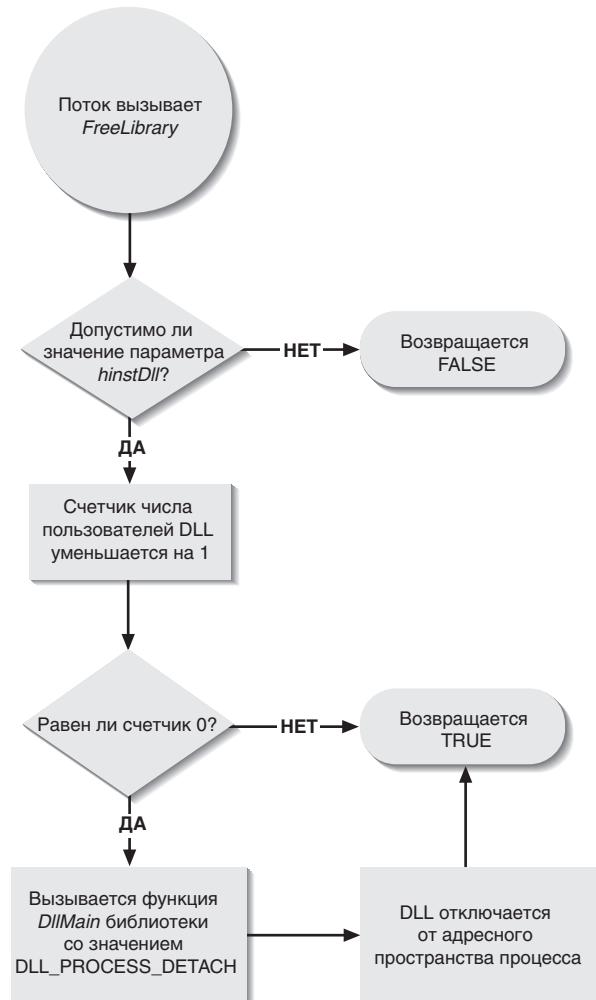


Рис. 20-3. Операции, выполняемые системой при вызове потоком функции *FreeLibrary*

Уведомление *DLL_THREAD_ATTACH*

Когда в процессе создается новый поток, система просматривает все DLL, спроектированные в данный момент на адресное пространство этого процесса, и в каждой из таких DLL вызывает *DllMain* со значением *DLL_THREAD_ATTACH*. Тем самым она уведомляет DLL-модули о необходимости инициализации, связанной с данным потоком. Только что созданный поток отвечает за выполнение кода в функциях *DllMain* всех

DLL. Работа его собственной (стартовой) функции начинается лишь после того, как все DLL-модули обработают уведомление `DLL_THREAD_ATTACH`.

Если в момент проецирования DLL на адресное пространство процесса в нем выполняется несколько потоков, система *не* вызывает `DllMain` со значением `DLL_THREAD_ATTACH` ни для одного из существующих потоков. Вызов `DllMain` с этим значением осуществляется, только если DLL проецируется на адресное пространство процесса в момент создания потока.

Обратите также внимание, что система не вызывает функции `DllMain` со значением `DLL_THREAD_ATTACH` и для первичного потока процесса. Любая DLL, проецируемая на адресное пространство процесса в момент его создания, получает уведомление `DLL_PROCESS_ATTACH`, а не `DLL_THREAD_ATTACH`.

Уведомление `DLL_THREAD_DETACH`

Лучший способ завершить поток — дождаться возврата из его стартовой функции, после чего система вызовет `ExitThread` и закроет поток. Эта функция лишь сообщает системе о том, что поток хочет завершиться, но система не уничтожает его немедленно. Сначала она просматривает все проекции DLL, находящиеся в данный момент в адресном пространстве процесса, и заставляет завершающий поток вызвать `DllMain` в каждой из этих DLL со значением `DLL_THREAD_DETACH`. Тем самым она уведомляет DLL-модули о необходимости очистки, связанной с данным потоком. Например, DLL-версия библиотеки C/C++ освобождает блок данных, используемый для управления многопоточными приложениями.

Заметьте, что DLL может не дать потоку завершиться. Например, такое возможно, когда функция `DllMain`, получив уведомление `DLL_THREAD_DETACH`, входит в бесконечный цикл. А операционная система закрывает поток только после того, как все DLL заканчивают обработку этого уведомления.



Если поток завершается из-за того, что другой поток вызвал для него `TerminateThread`, система *не* вызывает `DllMain` со значением `DLL_THREAD_DETACH`. Следовательно, ни одна DLL, спроектированная на адресное пространство процесса, не получит шанса на выполнение очистки до завершения потока, что может привести к потере данных. Поэтому `TerminateThread`, как и `TerminateProcess`, можно использовать лишь в самом крайнем случае!

Если при отключении DLL еще выполняются какие-то потоки, то для них `DllMain` не вызывается со значением `DLL_THREAD_DETACH`. Вы можете проверить это при обработке `DLL_PROCESS_DETACH` и провести необходимую очистку.

Ввиду упомянутых выше правил не исключена такая ситуация: поток вызывает `LoadLibrary` для загрузки DLL, в результате чего система вызывает из этой библиотеки `DllMain` со значением `DLL_PROCESS_ATTACH`. (В этом случае уведомление `DLL_THREAD_ATTACH` не посыпается.) Затем поток, загрузивший DLL, завершается, что приводит к новому вызову `DllMain` — на этот раз со значением `DLL_THREAD_DETACH`. Библиотека уведомляется о завершении потока, хотя она не получала `DLL_THREAD_ATTACH`, уведомляющего о его подключении. Поэтому будьте крайне осторожны при выполнении любой очистки, связанной с конкретным потоком. К счастью, большинство программ пишется так, что `LoadLibrary` и `FreeLibrary` вызываются одним потоком.

Как система упорядочивает вызовы *DllMain*

Система упорядочивает вызовы функции *DllMain*. Чтобы понять, что я имею в виду, рассмотрим следующий сценарий. Процесс А имеет два потока: А и В. На его адресное пространство проецируется DLL-модуль SomeDLL.dll. Оба потока собираются вызывать *CreateThread*, чтобы создать еще два потока: С и Д.

Когда поток А вызывает для создания потока С функцию *CreateThread*, система обращается к *DllMain* из SomeDLL.dll со значением *DLL_THREAD_ATTACH*. Пока поток С исполняет код *DllMain*, поток В вызывает *CreateThread* для создания потока D. Системе нужно вновь обратиться к *DllMain* со значением *DLL_THREAD_ATTACH*, и на этот раз код функции должен выполнять поток D. Но система упорядочивает вызовы *DllMain*, и поэтому приостановит выполнение потока D, пока поток С не завершит обработку кода *DllMain* и не выйдет из этой функции.

Закончив выполнение *DllMain*, поток С может начать выполнение своей функции потока. Теперь система возобновляет поток D и позволяет ему выполнить код *DllMain*, при возврате из которой он начнет обработку собственной функции потока.

Обычно никто и не задумывается над тем, что вызовы *DllMain* упорядочиваются. Но я завел об этом разговор потому, что один мой коллега как-то раз написал код, в котором была ошибка, связанная именно с упорядочиванием вызовов *DllMain*. Его код выглядел примерно так:

```
BOOL WINAPI DllMain(HINSTANCE hinstDll, DWORD fdwReason, PVOID fImpLoad) {

    HANDLE hThread;
    DWORD dwThreadId;

    switch (fdwReason) {
        case DLL_PROCESS_ATTACH:
            // DLL проецируется на адресное пространство процесса

            // создаем поток для выполнения какой-то работы
            hThread = CreateThread(NULL, 0, SomeFunction, NULL, 0, &dwThreadId);

            // задерживаем наш поток до завершения нового потока
            WaitForSingleObject(hThread, INFINITE);

            // доступ к новому потоку больше не нужен
            CloseHandle(hThread);
            break;

        case DLL_THREAD_ATTACH:
            // создается еще один поток
            break;

        case DLL_THREAD_DETACH:
            // поток завершается корректно
            break;

        case DLL_PROCESS_DETACH:
            // DLL выгружается из адресного пространства процесса
            break;
    }
    return(TRUE);
}
```

Нашли «жучка»? Мы-то его искали несколько часов. Когда *DllMain* получает уведомление DLL_PROCESS_ATTACH, создается новый поток. Системе нужно вновь вызвать эту же *DllMain* со значением DLL_THREAD_ATTACH. Но выполнение нового потока приостанавливается — ведь поток, из-за которого в *DllMain* было отправлено уведомление DLL_PROCESS_ATTACH, свою работу еще не закончил. Проблема кроется в вызове *WaitForSingleObject*. Она приостанавливает выполнение текущего потока до тех пор, пока не завершится новый. Однако у нового потока нет ни единого шанса не только на завершение, но и на выполнение хоть какого-нибудь кода — он приостановлен в ожидании того, когда текущий поток выйдет из *DllMain*. Вот Вам и взаимная блокировка — выполнение обоих потоков задержано навеки!

Впервые начав размышлять над этой проблемой, я обнаружил функцию *DisableThreadLibraryCalls*:

```
BOOL DisableThreadLibraryCalls(HINSTANCE hinstDll);
```

Вызывая ее, Вы сообщаете системе, что уведомления DLL_THREAD_ATTACH и DLL_THREAD_DETACH не должны посыпаться *DllMain* той библиотеки, которая указана в вызове. Мне показалось логичным, что взаимной блокировки не будет, если система не станет посылать DLL уведомления. Но, проверив свое решение (см. ниже), я убедился, что это не выход.

```
BOOL WINAPI DllMain(HINSTANCE hinstDll, DWORD fdwReason, PVOID fImpLoad) {  
  
    HANDLE hThread;  
    DWORD dwThreadId;  
  
    switch (fdwReason) {  
        case DLL_PROCESS_ATTACH:  
            // DLL проецируется на адресное пространство процесса  
  
            // предотвращаем вызов DllMain при создании  
            // или завершении потока  
            DisableThreadLibraryCalls(hinstDll);  
  
            // создаем поток для выполнения какой-то работы  
            hThread = CreateThread(NULL, 0, SomeFunction, NULL, 0, &dwThreadId);  
  
            // задерживаем наш поток до завершения нового потока  
            WaitForSingleObject(hThread, INFINITE);  
  
            // доступ к новому потоку больше не нужен  
            CloseHandle(hThread);  
            break;  
  
        case DLL_THREAD_ATTACH:  
            // создается еще один поток  
            break;  
  
        case DLL_THREAD_DETACH:  
            // поток завершается корректно  
            break;  
  
        case DLL_PROCESS_DETACH:  
            // DLL выгружается из адресного пространства процесса  
            break;  
    }  
}
```

```

        break;
    }
    return(TRUE);
}

```

Потом я понял, в чем дело. Создавая процесс, система создает и объект-мьютекс. У каждого процесса свой объект-мьютекс — он не разделяется между несколькими процессами. Его назначение — синхронизация всех потоков процесса при вызове ими функций *DllMain* из DLL, спроектированных на адресное пространство данного процесса.

Когда вызывается *CreateThread*, система создает сначала объект ядра «поток» и стек потока, затем обращается к *WaitForSingleObject*, передавая ей описатель объекта-мьютекса данного процесса. Как только поток захватит этот мьютекс, система заставит его вызвать *DllMain* из каждой DLL со значением *DLL_THREAD_ATTACH*. И лишь тогда система вызовет *ReleaseMutex*, чтобы освободить объект-мьютекс. Вот из-за того, что система работает именно так, дополнительный вызов *DisableThreadLibraryCalls* и не предотвращает взаимной блокировки потоков. Единственное, что я смог придумать, — переделать эту часть исходного кода так, чтобы ни одна *DllMain* не вызывала *WaitForSingleObject*.

Функция *DllMain* и библиотека С/C++

Рассматривая функцию *DllMain* в предыдущих разделах, я подразумевал, что для сборки DLL Вы используете компилятор Microsoft Visual C++. Весьма вероятно, что при написании DLL Вам понадобится поддержка со стороны стартового кода из библиотеки С/C++. Например, в DLL есть глобальная переменная — экземпляр какого-то С++-класса. Прежде чем DLL сможет безопасно ее использовать, для переменной нужно вызвать ее конструктор, а это работа стартового кода.

При сборке DLL компоновщик встраивает в конечный файл адрес DLL-функции входа/выхода. Вы задаете этот адрес компоновщику ключом /ENTRY. Если у Вас компоновщик Microsoft и Вы указали ключ /DLL, то по умолчанию он считает, что функция входа/выхода называется *_DllMainCRTStartup*. Эта функция содержится в библиотеке С/C++ и при компоновке статически подключается к Вашей DLL — даже если Вы используете DLL-версию библиотеки С/C++.

Когда DLL проецируется на адресное пространство процесса, система на самом деле вызывает именно *_DllMainCRTStartup*, а не Вашу функцию *DllMain*. Получив уведомление *DLL_PROCESS_ATTACH*, функция *_DllMainCRTStartup* инициализирует библиотеку С/C++ и конструирует все глобальные и статические С++-объекты. Закончив, *_DllMainCRTStartup* вызывает Вашу *DllMain*.

Как только DLL получает уведомление *DLL_PROCESS_DETACH*, система вновь вызывает *_DllMainCRTStartup*, которая теперь обращается к Вашей функции *DllMain*, и, когда та вернет управление, *_DllMainCRTStartup* вызовет деструкторы для всех глобальных и статических С++-объектов. Получив уведомление *DLL_THREAD_ATTACH*, функция *_DllMainCRTStartup* не делает ничего особенного. Но в случае уведомления *DLL_THREAD_DETACH*, она освобождает в потоке блок памяти *tiddata*, если он к тому времени еще не удален. Обычно в корректно написанной функции потока этот блок отсутствует, потому что она возвращает управление в *_threadstartex* из библиотеки С/C++ (см. главу 6). Функция *_threadstartex* сама вызывает *_endthreadex*, которая освобождает блок *tiddata* до того, как поток обращается к *ExitThread*.

Но представьте, что приложение, написанное на Паскале, вызывает функции из DLL, написанной на С/C++. В этом случае оно создаст поток, не прибегая к *_begin-*

threadex, и такой поток никогда не узнает о библиотеке C/C++. Далее поток вызовет функцию из DLL, которая в свою очередь обратится к библиотечной С-функции. Как Вы помните, подобные функции «на лету» создают блок *tiddata* и сопоставляют его с вызывающим потоком. Получается, что приложение, написанное на Паскале, может создавать потоки, способные без проблем обращаться к функциям из библиотеки C. Когда его функция потока возвращает управление, вызывается *ExitThread*, а библиотека C/C++ получает уведомление *DLL_THREAD_DETACH* и освобождает блок памяти *tiddata*, так что никакой утечки памяти не происходит. Здорово придумано, да?

Я уже говорил, что реализовать в коде Вашей DLL функцию *DllMain* не обязательно. Если у Вас нет этой функции, библиотека C/C++ использует свою реализацию *DllMain*, которая выглядит примерно так (если Вы связываете DLL со статической библиотекой C/C++):

```
BOOL WINAPI DllMain(HINSTANCE hinstDll, DWORD fdwReason, PVOID fImpLoad) {  
  
    if (fdwReason == DLL_PROCESS_ATTACH)  
        DisableThreadLibraryCalls(hinstDll);  
    return(TRUE);  
}
```

При сборке DLL компоновщик, не найдя в Ваших OBJ-файлах функцию *DllMain*, подключит *DllMain* из библиотеки C/C++. Если Вы не предоставили свою версию функции *DllMain*, библиотека C/C++ вполне справедливо будет считать, что Вас не интересуют уведомления *DLL_THREAD_ATTACH* и *DLL_THREAD_DETACH*. Функция *DisableThreadLibraryCalls* вызывается для ускорения создания и разрушения потоков.

Отложенная загрузка DLL

Microsoft Visual C++ 6.0 поддерживает отложенную загрузку DLL — новую, просто фантастическую функциональность, которая значительно упрощает работу с библиотеками. DLL отложенной загрузки (delay-load DLL) — это неявно связываемая DLL, которая не загружается до тех пор, пока Ваш код не обратится к какому-нибудь экспортируемому из нее идентификатору. Такие DLL могут быть полезны в следующих ситуациях.

- Если Ваше приложение использует несколько DLL, его инициализация может занимать длительное время, потому что загрузчику приходится проецировать их на адресное пространство процесса. Один из способов снять остроту этой проблемы — распределить загрузку DLL в ходе выполнения приложения. DLL отложенной загрузки позволяют легко решить эту задачу.
- Если приложение использует какую-то новую функцию и Вы пытаетесь запустить его в более старой версии операционной системы, в которой нет такой функции, загрузчик сообщает об ошибке и не дает запустить приложение. Вам нужно как-то обойти этот механизм и уже в период выполнения, выяснив, что приложение работает в старой версии системы, не вызывать новую функцию. Например, Ваша программа в Windows 2000 должна использовать функции PSAPI, а в Windows 98 — ToolHelp-функции (вроде *Process32Next*). При инициализации программа должна вызвать *GetVersionEx*, чтобы определить версию текущей операционной системы, и после этого обращаться к соответствующим функциям. Попытка запуска этой программы в Windows 98 приведет к тому, что загрузчик сообщит об ошибке, поскольку в этой системе нет модуля PSAPI.dll. Так вот, и эта проблема легко решается за счет DLL отложенной загрузки.

Я довольно долго экспериментировал с DLL отложенной загрузки в Visual C++ 6.0 и должен сказать, что Microsoft прекрасно справилась со своей задачей. DLL отложенной загрузки открывают массу дополнительных возможностей и корректно работают как в Windows 98, так и в Windows 2000.

Давайте начнем с простого: попробуем воспользоваться механизмом поддержки DLL отложенной загрузки. Для этого создайте, как обычно, свою DLL. Точно так же создайте и EXE-модуль, но потом Вы должны поменять пару ключей компоновщика и повторить сборку исполняемого файла. Вот эти ключи:

```
/Lib:DelayImp.lib
/DelayLoad:MyDll.dll
```

Первый ключ заставляет компоновщик внедрить в EXE-модуль специальную функцию, `__delayLoadHelper`, а второй — выполнить следующие операции:

- удалить MyDll.dll из раздела импорта исполняемого модуля, чтобы при инициализации процесса загрузчик операционной системы не пытался неявно связывать эту библиотеку с EXE-модулем;
- встроить в EXE-файл новый раздел отложенного импорта (.didat) со списком функций, импортируемых из MyDll.dll;
- привести вызовы функций из DLL отложенной загрузки к вызовам `__delayLoadHelper`.

При выполнении приложения вызов функции из DLL отложенной загрузки (далее для краткости — DLL-функции) фактически переадресуется к `__delayLoadHelper`. Последняя, просмотрев раздел отложенного импорта, знает, что нужно вызывать `LoadLibrary`, а затем `GetProcAddress`. Получив адрес DLL-функции, `__delayLoadHelper` делает так, чтобы в дальнейшем эта DLL-функция вызывалась напрямую. Обратите внимание, что каждая функция в DLL настраивается индивидуально при первом ее вызове. Ключ `/DelayLoad` компоновщика указывается для каждой DLL, загрузку которой требуется отложить.

Вот собственно, и все. Как видите, ничего сложного здесь нет. Однако следует учесть некоторые тонкости. Загружая Ваш EXE-файл, загрузчик операционной системы обычно пытается подключить требуемые DLL и при неудаче сообщает об ошибке. Но при инициализации процесса наличие DLL отложенной загрузки не проверяется. И если функция `__delayLoadHelper` уже в период выполнения не найдет нужную DLL, она возбудит программное исключение. Вы можете перехватить его, используя SEH, и как-то обработать. Если же Вы этого не сделаете, Ваш процесс будет закрыт. (О структурной обработке исключений см. главы 23, 24 и 25.)

Еще одна проблема может возникнуть, когда `__delayLoadHelper`, найдя Вашу DLL, не обнаружит в ней вызываемую функцию (например, загрузчик нашел старую версию DLL). В этом случае `__delayLoadHelper` также возбудит программное исключение, и все пойдет по уже описанной схеме. В программе-примере, которая представлена в следующем разделе, я покажу, как написать SEH-код, обрабатывающий подобные ошибки. В ней же Вы увидите и массу другого кода, не имеющего никакого отношения к SEH и обработке ошибок. Он использует дополнительные возможности (о них — чуть позже), предоставляемые механизмом поддержки DLL отложенной загрузки. Если эта более «продвинутая» функциональность Вас не интересует, просто удалите дополнительный код.

Разработчики Visual C++ определили два кода программных исключений: `VcppException(ERROR_SEVERITY_ERROR, ERROR_MOD_NOT_FOUND)` и `VcppException(ERROR_SEVE-`

RITY_ERROR, ERROR_PROC_NOT_FOUND). Они уведомляют соответственно об отсутствии DLL и DLL-функции. Моя функция фильтра исключений *DelayLoadDllExceptionFilter* реагирует на оба кода. При возникновении любого другого исключения она, как и положено корректно написанному фильтру, возвращает *EXCEPTION_CONTINUE_SEARCH*. (Программа не должна «глотать» исключения, которые не умеет обрабатывать.) Однако, если генерируется один из приведенных выше кодов, функция *_delayLoadHelper* предоставляет указатель на структуру *DelayLoadInfo*, содержащую некоторую дополнительную информацию. Она определена в заголовочном файле *DelayImp.h*, поставляемом с Visual C++.

```
typedef struct DelayLoadInfo {
    DWORD          cb;           // размер структуры
    PCImgDelayDescr pidd;        // "сырые" данные (все, что пока не обработано)
    FARPROC *      ppfn;         // указатель на адрес функции, которую надо загрузить
    LPCSTR          szDll;        // имя DLL
    DelayLoadProc   dlp;          // имя или порядковый номер процедуры
    HMODULE         hmodCur;      // hInstance загруженной библиотеки
    FARPROC          pfnCur;        // функция, которая будет вызвана на самом деле
    DWORD           dwLastError; // код ошибки
} DelayLoadInfo, * PDelayLoadInfo;
```

Экземпляр этой структуры данных создается и инициализируется функцией *_delayLoadHelper*, а ее элементы заполняются по мере выполнения задачи, связанной с динамической загрузкой DLL. Внутри Вашего SEH-фильтра элемент *szDll* указывает на имя загружаемой DLL, а элемент *dlp* — на имя нужной DLL-функции. Поскольку искать функцию можно как по порядковому номеру, так и по имени, *dlp* представляет собой следующее.

```
typedef struct DelayLoadProc {
    BOOL fImportByName;
    union {
        LPCSTR szProcName;
        DWORD  dwOrdinal;
    };
} DelayLoadProc;
```

Если DLL загружается, но требуемой функции в ней нет, Вы можете проверить элемент *hmodCur*, в котором содержится адрес проекции этой DLL, и элемент *dwLastError*, в который помещается код ошибки, вызвавшей исключение. Однако для фильтра исключения код ошибки, видимо, не понадобится, поскольку код исключения и так информирует о том, что произошло. Элемент *pfnCur* содержит адрес DLL-функции, и фильтр исключения устанавливает его в NULL, так как само исключение говорит о том, что *_delayLoadHelper* не смогла найти этот адрес.

Что касается остальных элементов, то *cb* служит для определения версии системы, *pidd* указывает на раздел, встроенный в модуль и содержащий список DLL отложенной загрузки, а *ppfn* — это адрес, по которому вызывается функция, если она найдена в DLL. Последние два параметра используются внутри *_delayLoadHelper* и рассчитаны на очень «продвинутое» применение — крайне маловероятно, что они Вам когда-нибудь понадобятся.

Итак, самое главное о том, как использовать DLL отложенной загрузки, я рассказал. Но это лишь видимая часть айсберга — их возможности гораздо шире. В частности, Вы можете еще и выгружать эти DLL. Допустим, что для распечатки документа Вашему приложению нужна специальная DLL. Такая DLL — подходящий кандидат на

отложенную загрузку, поскольку она требуется только на время печати документа. Когда пользователь выбирает команду Print, приложение обращается к соответствующей функции Вашей DLL, и та автоматически загружается. Все отлично, но, напечатав документ, пользователь вряд ли станет сразу же печатать что-то еще, а значит, Вы можете выгрузить свою DLL и освободить системные ресурсы. Потом, когда пользователь решит напечатать другой документ, DLL вновь будет загружена в адресное пространство Вашего процесса.

Чтобы DLL отложенной загрузки можно было выгружать, Вы должны сделать две вещи. Во-первых, при сборке исполняемого файла задать ключ /Delay:unload компоновщика. А во-вторых, немного изменить исходный код и поместить в точке выгрузки DLL вызов функции `_FUnloadDelayLoadedDLL`:

```
BOOL _FUnloadDelayLoadedDLL(PCSTR szDll);
```

Ключ /Delay:unload заставляет компоновщик создать в файле дополнительный раздел. В нем хранится информация, необходимая для сброса уже вызывавшихся DLL-функций, чтобы к ним снова можно было обратиться через `_delayLoadHelper`. Вызывая `_FUnloadDelayLoadedDLL`, Вы передаете имя выгружаемой DLL. После этого она просматривает раздел выгрузки (unload section) и сбрасывает адреса всех DLL-функций. И, наконец, `_FUnloadDelayLoadedDLL` вызывает `FreeLibrary`, чтобы выгрузить эту DLL.

Обратите внимание на несколько важных моментов. Во-первых, ни при каких условиях не вызывайте сами `FreeLibrary` для выгрузки DLL, иначе сброса адреса DLL-функции не произойдет, и впоследствии любое обращение к ней приведет к нарушению доступа. Во-вторых, при вызове `_FUnloadDelayLoadedDLL` в имени DLL нельзя указывать путь, а регистры всех букв должны быть точно такими же, как и при передаче компоновщику в ключе /DelayLoad; в ином случае вызов `_FUnloadDelayLoadedDLL` закончится неудачно. В-третьих, если Вы вообще не собираетесь выгружать DLL отложенной загрузки, не задавайте ключ /Delay:unload — тогда Вы уменьшите размер своего исполняемого файла. И, наконец, если Вы вызовете `_FUnloadDelayLoadedDLL` из модуля, собранного без ключа /Delay:unload, ничего страшного не случится: `_FUnloadDelayLoadedDLL` проигнорирует вызов и просто вернет FALSE.

Другая особенность DLL отложенной загрузки в том, что вызываемые Вами функции по умолчанию связываются с адресами памяти, по которым они, как считает система, будут находиться в адресном пространстве процесса. (О связывании мы поговорим чуть позже.) Поскольку связываемые разделы DLL отложенной загрузки увеличивают размер исполняемого файла, Вы можете запретить их создание, указав ключ /Delay:nobind компоновщика. Однако связывание, как правило, предпочтительнее, поэтому при сборке большинства приложений этот ключ использовать не следует.

И последняя особенность DLL отложенной загрузки. Она, кстати, наглядно демонстрирует характерное для Microsoft внимание к деталям. Функция `_delayLoadHelper` может вызывать предоставленные Вами функции-ловушки (hook functions), и они будут получать уведомления о том, как идет выполнение `_delayLoadHelper`, а также уведомления об ошибках. Кроме того, они позволяют изменять порядок загрузки DLL и формирования виртуального адреса DLL-функций.

Чтобы получать уведомления или изменить поведение `_delayLoadHelper`, нужно внести два изменения в свой исходный код. Во-первых, Вы должны написать функцию-ловушку по образу и подобию `DliHook`, код которой показан на рис. 20-6. Моя функция `DliHook` не влияет на характер работы `_delayLoadHelper`. Если Вы хотите изменить поведение `_delayLoadHelper`, начните с `DliHook` и модифицируйте ее код так, как Вам требуется. Потом передайте ее адрес функции `_delayLoadHelper`.

В статически подключаемой библиотеке DelayImp.lib определены две глобальные переменные типа *PfnDliHook*: *_pfnDliNotifyHook* и *_pfnDliFailureHook*:

```
typedef FARPROC (WINAPI *PfnDliHook)(  
    unsigned dliNotify,  
    PDelayLoadInfo pdli);
```

Как видите, это тип данных, соответствующий функции, и он совпадает с прототипом моей *DliHook*. В DelayImp.lib эти две переменные инициализируются значением NULL, которое сообщает *_delayLoadHelper*, что никаких функций-ловушек вызывать не требуется. Чтобы Ваша функция-ловушка все же вызывалась, Вы должны присвоить ее адрес одной из этих переменных. В своей программе я просто добавил на глобальном уровне две строки:

```
PfnDliHook __pfnDliNotifyHook = DliHook;  
PfnDliHook __pfnDliFailureHook = DliHook;
```

Так что *_delayLoadHelper* фактически работает с двумя функциями обратного вызова: одна вызывается для уведомлений, другая — для сообщений об ошибках. Поскольку их прототипы идентичны, а первый параметр, *dliNotify*, сообщает о причине вызова функции, я всегда упрощаю себе жизнь, создавая одну функцию и настраивая на нее обе переменные.

Механизм отложенной загрузки DLL, введенный в Visual C++ 6.0, — вещь весьма интересная, и я знаю многих разработчиков, которые давно мечтали о нем. Он будет полезен в очень большом числе приложений (особенно от Microsoft).

Программа-пример DelayLoadApp

Эта программа, «20 DelayLoadApp.exe» (см. листинг на рис. 20-6), показывает, как использовать все преимущества DLL отложенной загрузки. Для демонстрации нам понадобится небольшой DLL-файл; он находится в каталоге 20-DelayLoadLib на компакт-dиске, прилагаемом к книге.

Так как программа загружает модуль «20 DelayLoadLib» с задержкой, загрузчик не проецирует его на адресное пространство процесса при запуске. Периодически вызывая функцию *IsModuleLoaded*, программа выводит окно, которое информирует, загружен ли модуль в адресное пространство процесса. При первом запуске модуль «20 DelayLoadLib» не загружается, о чем и сообщается в окне (рис. 20-4).



Рис. 20-4. DelayLoadApp сообщает, что модуль «20 DelayLoadLib» не загружен

Далее программа вызывает функцию, импортируемую из DLL, и это заставляет *_delayLoadHelper* автоматически загрузить нужную DLL. Когда функция вернет управление, программа выведет окно, показанное на рис. 20-5.



Рис. 20-5. DelayLoadApp сообщает, что модуль «20 DelayLoadLib» загружен

Когда пользователь закроет это окно, будет вызвана другая функция из той же DLL. В этом случае DLL не перезагружается в адресное пространство, но перед вызовом новой функции придется определять ее адрес.

Далее вызывается `_FUnloadDelayLoadedDLL`, и модуль «20 DelayLoadLib» выгружается из памяти. После очередного вызова `IsModuleLoaded` на экране появляется окно, показанное на рис. 20-4. Наконец, вновь вызывается импортируемая функция, что приводит к повторной загрузке модуля «20 DelayLoadLib», а `IsModuleLoaded` открывает окно, как на рис. 20-5.

Если все нормально, то программа будет работать, как я только что рассказал. Однако, если перед запуском программы Вы удалите модуль «20 DelayLoadLib» или если в этом модуле не окажется одной из импортируемых функций, будет возбуждено исключение. Из моего кода видно, как корректно выйти из такой ситуации.

Наконец, эта программа демонстрирует, как настроить функцию-ловушку из DLL отложенной загрузки. Моя схематическая функция `DliHook` не делает ничего интересного. Тем не менее она перехватывает различные уведомления и показывает их Вам.



DelayLoadApp.cpp

```

/*
Модуль: DelayLoadApp.cpp
Автор: Copyright (c) 2000, Джейфри Рихтер (Jeffrey Richter)
*/

#include "..\CmnHdr.h"      /* см. приложение A */
#include <Windowsx.h>
#include <tchar.h>

/////////////////////////////// 

#include <Delayimp.h>      // для обработки ошибок и доступа
                           // к дополнительной функциональности
#include "..\20-DelayLoadLib\DelayLoadLib.h" // прототипы функций в моей DLL

/////////////////////////////// 

// статическое связывание с __delayLoadHelper и __FUnloadDelayLoadedDLL
#pragma comment(lib, "Delayimp.lib")

// сообщаем компоновщику, что загрузка DLL должна быть отложена;
// обратите внимание на две группы символов ("\"), введенных
// из-за пробела в имени файла
#pragma comment(linker, "/DelayLoad:\"20 DelayLoadLib.dll\"")

// сообщаем компоновщику, что DLL должна при необходимости выгружаться
#pragma comment(linker, "/Delay:unload")

// сообщаем компоновщику, что связывание DLL отложенной загрузки не требуется;
// но обычно это все же требуется, поэтому я закомментировал следующую строку
// #pragma comment(linker, "/Delay:nobind")

```

Рис. 20-6. Программа-пример *DelayLoadApp*

см. след. стр.

Рис. 20-6. продолжение

```
// имя модуля отложенной загрузки (используется только этой программой)
TCHAR g_szDelayLoadModuleName[] = TEXT("20 DelayLoadLib");

//////////////////////////////



// упреждающий прототип функции
LONG WINAPI DelayLoadD1lExceptionFilter(PEXCEPTION_POINTERS pep);

//////////////////////////////


void IsModuleLoaded(PCTSTR pszModuleName) {

    HMODULE hmod = GetModuleHandle(pszModuleName);
    char sz[100];
#ifdef UNICODE
    wsprintfA(sz, "Module \"%S\" is %Sloaded.",
              pszModuleName, (hmod == NULL) ? L"not " : L"");
#else
    wsprintfA(sz, "Module \"%s\" is %sloaded.",
              pszModuleName, (hmod == NULL) ? "not " : "");
#endif
    chMB(sz);
}

//////////////////////////////


int _tWinMain(HINSTANCE hinstExe, HINSTANCE, PTSTR pszCmdLine, int) {

    // помещаем все вызовы функций DLL отложенной загрузки в SEH-фрейм
    __try {
        int x = 0;

        // если Вы запустили программу под отладчиком, попробуйте выбрать
        // из меню Debug новую команду Modules, и Вы убедитесь, что
        // до выполнения следующей строки наша DLL еще не загружалась
        IsModuleLoaded(g_szDelayLoadModuleName);

        x = fnLib(); // пытаемся вызвать DLL-функцию

        // выберите из меню Debug команду Modules, чтобы убедиться: DLL загружена
        IsModuleLoaded(g_szDelayLoadModuleName);

        x = fnLib2(); // пытаемся вызвать DLL-функцию

        // Выгружаем DLL отложенной загрузки.
        // Примечание: имя должно быть идентично указанному в /DelayLoad.
        __FUnloadDelayedDLL("20 DelayLoadLib.dll");

        // выберите из меню Debug команду Modules, чтобы убедиться: DLL выгружена
        IsModuleLoaded(g_szDelayLoadModuleName);

        x = fnLib(); // пытаемся вызвать DLL-функцию
    }
}
```

Рис. 20-6. продолжение

```

// выберите из меню Debug команду Modules, чтобы убедиться:
// DLL снова загружена
IsModuleLoaded(g_szDelayLoadModuleName);
}

__except (DelayLoadDllExceptionFilter(GetExceptionInformation())) {
    // здесь нам делать нечего, поток продолжает работу
}

// сюда можно добавить еще чуточку кода...

return(0);
}

///////////////////////////////



LONG WINAPI DelayLoadDllExceptionFilter(PEXCEPTION_POINTERS pep) {

// предполагаем, что мы распознаем это исключение
LONG lDisposition = EXCEPTION_EXECUTE_HANDLER;

// если проблема связана с отложенной загрузкой, ExceptionInformation[0]
// указывает на структуру DelayLoadInfo, в которой содержится
// подробная информация об ошибке
PDelayLoadInfo pdli =
    PDelayLoadInfo(pep->ExceptionRecord->ExceptionInformation[0]);

// создаем буфер, в котором мы будем формировать сообщения об ошибках
char sz[500] = { 0 };

switch (pep->ExceptionRecord->ExceptionCode) {
case VcppException(ERROR_SEVERITY_ERROR, ERROR_MOD_NOT_FOUND):
    // DLL найти не удалось
    wsprintfA(sz, "Dll not found: %s", pdli->szDll);
    break;

case VcppException(ERROR_SEVERITY_ERROR, ERROR_PROC_NOT_FOUND):
    // DLL найдена, но нужной функции в ней нет
    if (pdli->dlp.fImportByName) {
        wsprintfA(sz, "Function %s was not found in %s",
                  pdli->dlp.szProcName, pdli->szDll);
    } else {
        wsprintfA(sz, "Function ordinal %d was not found in %s",
                  pdli->dlp.dwOrdinal, pdli->szDll);
    }
    break;

default:
    // это исключение мы не распознаем
    lDisposition = EXCEPTION_CONTINUE_SEARCH;
    break;
}
}

```

см. след. стр.

Рис. 20-6. продолжение

```
if (lDisposition == EXCEPTION_EXECUTE_HANDLER) {  
    // мы распознали ошибку и сформировали сообщение; показываем его  
    chMB(sz);  
}  
  
return(lDisposition);  
}  
  
//////////  
  
// схематическая функция DliHook, которая не делает ничего интересного  
FARPROC WINAPI DliHook(unsigned dliNotify, PDelayLoadInfo pdli) {  
  
    FARPROC fp = NULL; // значение, возвращаемое по умолчанию  
  
    // Примечание: элементы структуры DelayLoadInfo, на которые указывает pdli,  
    // содержат сведения о ходе выполнения работы  
  
    Switch (dliNotify) {  
        Case dliStartProcessing:  
            // Вызывается, когда __delayLoadHelper пытается найти DLL или функцию.  
            // Возвращаем 0, если изменять поведение не требуется, и ненулевое  
            // значение, если изменить поведение все же необходимо (Вы все равно  
            // получите dliNoteEndProcessing).  
            break;  
  
        case dliNotePreLoadLibrary:  
            // Вызывается непосредственно перед вызовом LoadLibrary.  
            // Возвращаем NULL, чтобы __delayLoadHelper вызвала LoadLibrary.  
            // Но Вы можете сами вызвать LoadLibrary и вернуть HMODULE.  
            fp = (FARPROC) (HMODULE) NULL;  
            break;  
  
        case dliFailLoadLib:  
            // Вызывается, если LoadLibrary терпит неудачу.  
            // И на этот раз Вы можете сами вызвать LoadLibrary и вернуть HMODULE.  
            // Если Вы вернете NULL, __delayLoadHelper возбудит исключение  
            // ERROR_MOD_NOT_FOUND.  
            fp = (FARPROC) (HMODULE) NULL;  
            break;  
  
        case dliNotePreGetProcAddress:  
            // Вызывается непосредственно перед вызовом GetProcAddress.  
            // Возвращаем NULL, чтобы __delayLoadHelper вызвала GetProcAddress.  
            // Но Вы можете сами вызвать GetProcAddress и вернуть адрес.  
            fp = (FARPROC) NULL;  
            break;  
  
        case dliFailGetProc:  
            // Вызывается, если GetProcAddress терпит неудачу.  
            // Вы можете сами вызвать GetProcAddress и вернуть адрес.  
            // Если Вы вернете NULL, __delayLoadHelper возбудит исключение
```

Рис. 20-6. продолжение

```

// ERROR_PROC_NOT_FOUND.
fp = (FARPROC) NULL;
break;

case dliNoteEndProcessing:
// Простое уведомление об окончании работы __delayLoadHelper.
// Вы можете пользоваться элементами структуры DelayLoadInfo,
// на которую указывает pdli, и при необходимости возбудить исключение.
break;
}

return(fp);
}

/////////////////////////////// Конец файла ///////////////////////////////

```

// сообщаем __delayLoadHelper вызвать мою функцию-ловушку

PfnDliHook __pfnDliNotifyHook = DliHook;

PfnDliHook __pfnDliFailureHook = DliHook;

/////////////////////////////// Конец файла ///////////////////////////////

DelayLoadLib.cpp

```

*****  

Модуль: DelayLoadLib.cpp  

Автор: Copyright (c) 2000, Джейфри Рихтер (Jeffrey Richter)  

*****  

#include "..\CmnHdr.h"      /* см. приложение A */  

#include <Windowsx.h>  

#include <tchar.h>  

/////////////////////////////// Конец файла ///////////////////////////////  

#define DELAYLOADLIBAPI extern "C" __declspec(dllexport)  

#include "DelayLoadLib.h"  

/////////////////////////////// Конец файла ///////////////////////////////  

int fnLib() {
    return(321);
}  

/////////////////////////////// Конец файла ///////////////////////////////  

int fnLib2() {
    return(123);
}  

/////////////////////////////// Конец файла ///////////////////////////////

```

см. след. стр.

Рис. 20-6. продолжение**DelayLoadLib.h**

```
/*****************************************************************************  
Модуль: DelayLoadLib.h  
Автор: Copyright (c) 2000, Джейфри Рихтер (Jeffrey Richter)  
*****/  
  
#ifndef DELAYLOADLIBAPI  
#define DELAYLOADLIBAPI extern "C" __declspec(dllimport)  
#endif  
  
////////////////////////////////////////////////////////////////////////  
  
DELAYLOADLIBAPI int fnLib();  
DELAYLOADLIBAPI int fnLib2();  
  
//////////////////////////////////////////////////////////////////////// Конец файла //////////////////////////////////////////////////////////////////////
```

Переадресация вызовов функций

Запись о переадресации вызова функции (function forwarder) — это строка в разделе экспорта DLL, которая перенаправляет вызов к другой функции, находящейся в другой DLL. Например, запустив утилиту DumpBin из Visual C++ для Kernel32.dll в Windows 2000, Вы среди прочей информации увидите и следующее.

```
C:\winnt\system32>DumpBin -Exports Kernel32.dll  
      (часть вывода опущена)  
360 167 HeapAlloc (forwarded to NTDLL.RtlAllocateHeap)  
361 168 HeapCompact (000128D9)  
362 169 HeapCreate (000126EF)  
363 16A HeapCreateTagsW (0001279E)  
364 16B HeapDestroy (00012750)  
365 16C HeapExtend (00012773)  
366 16D HeapFree (forwarded to NTDLL.RtlFreeHeap)  
367 16E HeapLock (000128ED)  
368 16F HeapQueryTagW (000127B8)  
369 170 HeapReAlloc (forwarded to NTDLL.RtlReAllocateHeap)  
370 171 HeapSize (forwarded to NTDLL.RtlSizeHeap)  
      (остальное тоже опущено)
```

Здесь есть четыре переадресованные функции. Всякий раз, когда Ваше приложение вызывает *HeapAlloc*, *HeapFree*, *HeapReAlloc* или *HeapSize*, его EXE-модуль динамически связывается с Kernel32.dll. При запуске EXE-модуля загрузчик загружает Kernel32.dll и, обнаружив, что переадресуемые функции на самом деле находятся в NTDLL.dll, загружает и эту DLL. Обращаясь к *HeapAlloc*, программа фактически вызывает функцию *RtlAllocateHeap* из NTDLL.dll. А функции *HeapAlloc* вообще нет!

При вызове *HeapAlloc* (см. ниже) функция *GetProcAddress* просмотрит раздел экспорта Kernel32.dll и, выяснив, что *HeapAlloc* — переадресуемая функция, рекурсивно вызовет сама себя для поиска *RtlAllocateHeap* в разделе экспорта NTDLL.dll.

```
GetProcAddress(GetModuleHandle("Kernel32"), "HeapAlloc");
```

Вы тоже можете применять переадресацию вызовов функций в своих DLL. Самый простой способ — воспользоваться директивой *pragma*:

```
// переадресация к функции из DllWork
#pragma comment(linker, "/export:SomeFunc=DllWork.SomeOtherFunc")
```

Эта директива сообщает компоновщику, что DLL должна экспортировать функцию *SomeFunc*, которая на самом деле реализована как функция *SomeOtherFunc* в модуле *DllWork.dll*. Такая запись нужна для каждой переадресуемой функции.

Известные DLL

Некоторые DLL, поставляемые с операционной системой, обрабатываются по-особому. Они называются *известными DLL* (known DLLs) и ведут себя точно так же, как и любые другие DLL с тем исключением, что система всегда ищет их в одном и том же каталоге. В реестре есть раздел:

```
HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\
    Session Manager\KnownDLLs
```

Содержимое этого раздела может выглядеть примерно так, как показано ниже (при просмотре реестра с помощью утилиты RegEdit.exe).

The screenshot shows the Windows Registry Editor window. The left pane displays a tree view of registry keys under 'Session Manager'. One of the keys is 'KnownDLLs', which is expanded to show a list of DLL names and their types. The right pane is a table with three columns: 'Name', 'Type', and 'Data'. The 'Name' column lists the DLL names, many of which start with an 'a' followed by a number. The 'Type' column shows them as REG_SZ. The 'Data' column contains the paths to the DLL files, such as 'advapi32.dll' and 'kernel32.dll'. The status bar at the bottom of the editor window shows the path: 'My Computer\HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Session Manager\KnownDLLs'.

Name	Type	Data
a0 (Default)	REG_SZ	{value not set}
a0 advapi32	REG_SZ	advapi32.dll
a0 comdlg32	REG_SZ	comdlg32.dll
a0 DLLDirectory	REG_EXPAND_SZ	%SystemRoot%\system32
a0 gd32	REG_SZ	gd32.dll
a0 imagehlp	REG_SZ	imagehlp.dll
a0 kernel32	REG_SZ	kernel32.dll
a0 lz32	REG_SZ	lz32.dll
a0 ole32	REG_SZ	ole32.dll
a0 oleaut32	REG_SZ	oleaut32.dll
a0 olecli32	REG_SZ	olecli32.dll
a0 olecnv32	REG_SZ	olecnv32.dll
a0 olesvr32	REG_SZ	olesvr32.dll
a0 olethk32	REG_SZ	olethk32.dll
a0 rpcrt4	REG_SZ	rpcrt4.dll
a0 shell32	REG_SZ	shell32.dll
a0 url	REG_SZ	url.dll
a0 urlmon	REG_SZ	urlmon.dll
a0 user32	REG_SZ	user32.dll
a0 version	REG_SZ	version.dll
a0 wininet	REG_SZ	wininet.dll
a0 wldap32	REG_SZ	wldap32.dll

Как видите, здесь содержится набор параметров, имена которых совпадают с именами известных DLL. Значения этих параметров представляют собой строки, идентичные именам параметров, но дополненные расширением .dll. (Впрочем, это не всегда так, и Вы сами убедитесь в этом на следующем примере.) Когда Вы вызываете *LoadLibrary* или *LoadLibraryEx*, каждая из них сначала проверяет, указано ли имя DLL вместе с расширением .dll. Если нет, поиск DLL ведется по обычным правилам.

Если же расширение .dll указано, функция его отбрасывает и ищет в разделе реестра *KnownDLLs* параметр, имя которого совпадает с именем DLL. Если его нет, вновь применяются обычные правила поиска. А если он есть, система считывает значение этого параметра и пытается загрузить заданную в нем DLL. При этом система ищет

DLL в каталоге, на который указывает значение, связанное с параметром реестра *DllDirectory*. По умолчанию в Windows 2000 параметру *DllDirectory* присваивается значение %SystemRoot%\System32.

А теперь допустим, что мы добавили в раздел реестра *KnownDLLs* такой параметр:

Имя параметра: SomeLib

Значение параметра: SomeOtherLib.dll

Когда мы вызовем следующую функцию, система будет искать файл по обычным правилам.

```
LoadLibrary("SomeLib");
```

Но если мы вызовем ее так, как показано ниже, система увидит, что в реестре есть параметр с идентичным именем (не забудьте: она отбрасывает расширение .dll).

```
LoadLibrary("SomeLib.dll");
```

Таким образом, система попытается загрузить SomeOtherLib.dll вместо SomeLib.dll. При этом она будет сначала искать SomeOtherLib.dll в каталоге %SystemRoot%\System32. Если нужный файл в этом каталоге есть, будет загружен именно он. Нет — *LoadLibrary(Ex)* вернет NULL, а *GetLastError* — ERROR_FILE_NOT_FOUND (2).

Перенаправление DLL

WINDOWS 98 Windows 98 не поддерживает перенаправление DLL.

Когда разрабатывались первые версии Windows, оперативная память и дисковое пространство были крайне дефицитным ресурсом, так что Windows была рассчитана на предельно экономное их использование — с максимальным разделением между потребителями. В связи с этим Microsoft рекомендовала размещать все модули, используемые многими приложениями (например, библиотеку C/C++ и DLL, относящиеся к MFC) в системном каталоге Windows, где их можно было легко найти.

Однако со временем это вылилось в серьезную проблему: программы установки приложений то и дело перезаписывали новые системные файлы старыми или не полностью совместимыми. Из-за этого уже установленные приложения переставали работать. Но сегодня жесткие диски стали очень емкими и недорогими, оперативная память тоже значительно подешевела. Поэтому Microsoft сменила свою позицию на прямо противоположную: теперь она настоятельно рекомендует размещать все файлы приложения в своем каталоге и ничего не трогать в системном каталоге Windows. Тогда Ваше приложение не нарушит работу других программ, и наоборот.

С той же целью Microsoft ввела в Windows 2000 поддержку перенаправления DLL (DLL redirection). Она заставляет загрузчик операционной системы загружать модули сначала из каталога Вашего приложения и, только если их там нет, искать в других каталогах.

Чтобы загрузчик всегда проверял сначала каталог приложения, нужно всего лишь поместить туда специальный файл. Его содержимое не имеет значения и игнорируется — важно только его имя: оно должно быть в виде *AppName.local*. Так, если исполняемый файл Вашего приложения — *SuperApp.exe*, присвойте перенаправляющему файлу имя *SuperApp.exe.local*.

Функция *LoadLibrary(Ex)* проверяет наличие этого файла и, если он есть, загружает модуль из каталога приложения; в ином случае *LoadLibrary(Ex)* работает так же, как и раньше.

Перенаправление DLL исключительно полезно для работы с зарегистрированными СОМ-объектами. Оно позволяет приложению размещать DLL с СОМ-объектами в своем каталоге, и другие программы, регистрирующие те же объекты, не будут мешать его нормальной работе.

Модификация базовых адресов модулей

У каждого EXE и DLL-модуля есть *предпочтительный базовый адрес* (preferred base address) — идеальный адрес, по которому он должен проецироваться на адресное пространство процесса. Для EXE-модуля компоновщик выбирает в качестве такого адреса значение 0x00400000, а для DLL-модуля — 0x10000000. Выяснить этот адрес позволяет утилита DumpBin с ключом /Headers. Вот какую информацию сообщает DumpBin о самой себе:

```
C:\>DUMPBIN /headers dumpbin.exe
```

```
Microsoft (R) COFF Binary File Dumper Version 6.00.8168
Copyright (C) Microsoft Corp 1992-1998. All rights reserved.
```

```
Dump of file dumpbin.exe
```

```
PE signature found
```

```
File Type: EXECUTABLE IMAGE
```

FILE HEADER VALUES

```
14C machine (i386)
3 number of sections
3588004A time date stamp Wed Jun 17 10:43:38 1998
0 file pointer to symbol table
0 number of symbols
E0 size of optional header
10F characteristics
    Relocations stripped
    Executable
    Line numbers stripped
    Symbols stripped
    32 bit word machine
```

OPTIONAL HEADER VALUES

```
10B magic #
6.00 linker version
1000 size of code
2000 size of initialized data
    0 size of uninitialized data
1320 RVA of entry point
1000 base of code
2000 base of data
400000 image base      <-- предпочтительный базовый адрес модуля
1000 section alignment
```

см. след. стр.

```
1000 file alignment
4.00 operating system version
0.00 image version
4.00 subsystem version
    0 Win32 version
4000 size of image
1000 size of headers
127E2 checksum
    3 subsystem (Windows CUI)
    0 DLL characteristics
100000 size of stack reserve
    1000 size of stack commit

:
```

При запуске исполняемого модуля загрузчик операционной системы создает виртуальное адресное пространство нового процесса и проецирует этот модуль по адресу 0x00400000, а DLL-модуль — по адресу 0x10000000. Почему так важен предпочтительный базовый адрес? Взгляните на следующий фрагмент кода:

```
int g_x;

void Func() {
    g_x = 5; // нас интересует эта строка
}
```

После обработки функции *Func* компилятором и компоновщиком полученный машинный код будет выглядеть приблизительно так:

```
MOV [0x00414540], 5
```

Иначе говоря, компилятор и компоновщик «жестко зашили» в машинный код адрес переменной *g_x* в адресном пространстве процесса (0x00414540). Но, конечно, этот адрес корректен, только если исполняемый модуль будет загружен по базовому адресу 0x00400000.

А что получится, если тот же исходный код будет помещен в DLL? Тогда машинный код будет иметь такой вид:

```
MOV [0x10014540], 5
```

Заметьте, что и на этот раз виртуальный адрес переменной *g_x* «жестко зашип» в машинный код. И опять же этот адрес будет правилен только при том условии, что DLL загрузится по своему базовому адресу.

О'кэй, а теперь представьте, что Вы создали приложение с двумя DLL. По умолчанию компоновщик установит для EXE-модуля предпочтительный базовый адрес 0x00400000, а для обеих DLL — 0x10000000. Если Вы затем попытаетесь запустить исполняемый файл, загрузчик создаст виртуальное адресное пространство и спроектирует EXE-модуль по адресу 0x00400000. Далее первая DLL будет спроектирована по адресу 0x10000000, но загрузить вторую DLL по предпочтительному базовому адресу не удастся — ее придется проецировать по какому-то другому адресу.

Переадресация (relocation) в EXE- или DLL-модуле — операция просто ужасающая, и Вы должны сделать все, чтобы избежать ее. Почему? Допустим, загрузчик переместил вторую DLL по адресу 0x20000000. Тогда код, который присваивает переменной *g_x* значение 5, должен измениться на:

```
MOV [0x20014540], 5
```

Но в образе файла код остался прежним:

```
MOV [0x10014540], 5
```

Если будет выполнен именно этот код, он перезапишет какое-то 4-байтовое значение в первой DLL значением 5. Но, по идеи, такого не должно случиться. Загрузчик исправит этот код. Дело в том, что, создавая модуль, компоновщик встраивает в конечный файл раздел переадресации (relocation section) со списком байтовых смещений. Эти смещения идентифицируют адреса памяти, используемые инструкциями машинного кода. Если загрузчику удастся спроектировать модуль по его предпочтительному базовому адресу, раздел переадресации не понадобится. Именно этого мы и хотим.

С другой стороны, если модуль не удастся спроектировать по базовому адресу, загрузчик обратится к разделу переадресации и последовательно обработает все его записи. Для каждой записи загрузчик обращается к странице памяти, где содержится машинная команда, которую надо модифицировать, получает используемый ею на данный момент адрес и добавляет к нему разницу между предпочтительным базовым адресом модуля и его фактическим адресом.

В предыдущем примере вторая DLL была спроектирована по адресу 0x20000000, тогда как ее предпочтительный базовый адрес — 0x10000000. Получаем разницу (0x10000000), добавляем ее к адресу в машинной команде и получаем:

```
MOV [0x20014540], 5
```

Теперь и вторая DLL корректно ссылается на переменную *g_x*.

Невозможность загрузить модуль по предпочтительному базовому адресу создает две крупные проблемы.

- Загрузчику приходится обрабатывать все записи раздела переадресации и модифицировать уйму кода в модуле. Это сильнейшим образом оказывается на быстродействии и может резко увеличить время инициализации приложения.
- Из-за того что загрузчик модифицирует в оперативной памяти страницы с кодом модуля, системный механизм копирования при записи создает их копии в страничном файле.

Вторая проблема особенно неприятна, поскольку теперь страницы с кодом модуля больше нельзя выгружать из памяти и перезагружать из его файла на диске. Вместо этого страницы будут постоянно сбрасываться в страничный файл и подгружаться из него. Это тоже отрицательно скажется на производительности. Но и это еще не все. Поскольку все страницы с кодом модуля размещаются в страничном файле, в системе сокращается объем общей памяти, доступной другим процессам, а это ограничивает размер электронных таблиц, документов текстовых процессоров, чертежей CAD, растровых изображений и т. д.

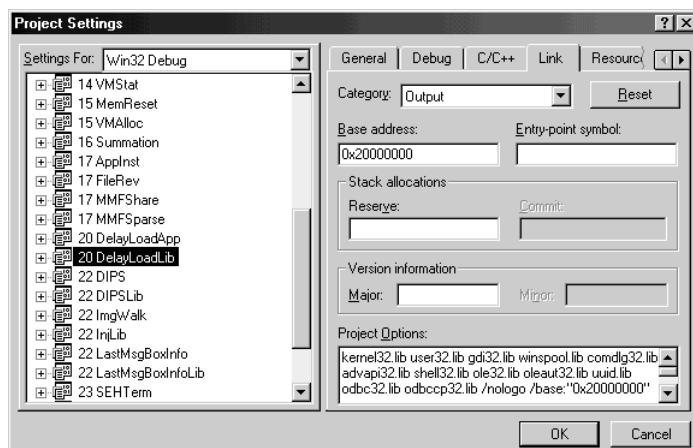
Кстати, Вы можете создать EXE- или DLL-модуль без раздела переадресации, указав при сборке ключ /FIXED компоновщика. Тогда у модуля будет меньший размер, но загрузить его по другому базовому адресу, кроме предпочтительного, уже не удастся. Если загрузчику понадобится модифицировать адреса в модуле, в котором нет раздела переадресации, он уничтожит весь процесс, и пользователь увидит сообщение «Abnormal Process Termination» («аварийное завершение процесса»).

Для DLL, содержащей только ресурсы, это тоже проблема. Хотя в ней нет машинного кода, отсутствие раздела переадресации не позволит загрузить ее по базовому

адресу, отличному от предпочтительного. Просто нелепо. Но, к счастью, компоновщик может встроить в заголовок модуля информацию о том, что в модуле нет разделяемой переадресации, так как он вообще не нужен. А загрузчик Windows 2000, обнаружив эту информацию, может загрузить DLL, которая содержит только ресурсы, без дополнительной нагрузки на страничный файл.

Для создания файла с немодифицируемыми адресами предназначен ключ /SUBSYSTEM:WINDOWS, 5.0 или /SUBSYSTEM:CONSOLE, 5.0; ключ /FIXED при этом не нужен. Если компоновщик определяет, что модификация адресов в модуле не понадобится, он опускает раздел переадресации и сбрасывает в заголовке специальный флаг IMAGE_FILE_RELOCS_STRIPPED. Тогда Windows 2000 увидит, что данный модуль можно загружать по базовому адресу, отличному от предпочтительного, и что ему не требуется модификация адресов. Но все, о чем я только что рассказал, поддерживается лишь в Windows 2000 (вот почему в ключе /SUBSYSTEM указывается значение 5.0).

Теперь Вы понимаете, насколько важен предпочтительный базовый адрес. Загружая несколько модулей в одно адресное пространство, для каждого из них приходится выбирать свои базовые адреса. Диалоговое окно Project Settings в среде Microsoft Visual Studio значительно упрощает решение этой задачи. Вам нужно лишь открыть вкладку Link, в списке Category указать Output, а в поле Base Address ввести предпочтительный адрес. Например, на следующей иллюстрации для DLL установлен базовый адрес 0x20000000.



Кстати, всегда загружайте DLL, начиная со старших адресов; это позволяет уменьшить фрагментацию адресного пространства.



Предпочтительные базовые адреса должны быть кратны гранулярности выделения памяти (64 Кб на всех современных платформах). В будущем эта цифра может измениться. Подробнее о гранулярности выделения памяти см. главу 13.

О'кэй, все это просто замечательно, но что делать, если понадобится загрузить кучу модулей в одно адресное пространство? Было бы неплохо «одним махом» задать правильные базовые адреса для всех модулей. К счастью, такой способ есть.

В Visual Studio есть утилита Rebase.exe. Запустив ее без ключей в командной строке, Вы получите информацию о том, как ею пользоваться. Она описана в документации Platform SDK, и я не буду ее здесь детально рассматривать. Добавлю лишь, что в ней нет ничего сверхъестественного: она просто вызывает функцию *ReBaseImage* для каждого указанного файла. Вот что представляет собой эта функция:

```

BOOL ReBaseImage(
    PSTR CurrentImageName, // полное имя обрабатываемого файла
    PSTR SymbolPath, // символьный путь к файлу (необходим для
                     // корректности отладочной информации)
    BOOL fRebase, // TRUE = выполнить реальную модификацию адреса;
                  // FALSE = имитировать такую модификацию
    BOOL fRebaseSysFileOk, // FALSE = не модифицировать адреса системных файлов
    BOOL fGoingDown, // TRUE = модифицировать адрес модуля,
                     // продвигаясь в сторону уменьшения адресов
    ULONG CheckImageSize, // ограничение на размер получаемого в итоге модуля
    ULONG* pOldImageSize, // исходный размер модуля
    ULONG* pOldImageBase, // исходный базовый адрес модуля
    ULONG* pNewImageSize, // новый размер модуля
    ULONG* pNewImageBase, // новый базовый адрес модуля
    ULONG TimeStamp); // новая временная метка модуля

```

Когда Вы запускаете утилиту Rebase, указывая ей несколько файлов, она выполняет следующие операции.

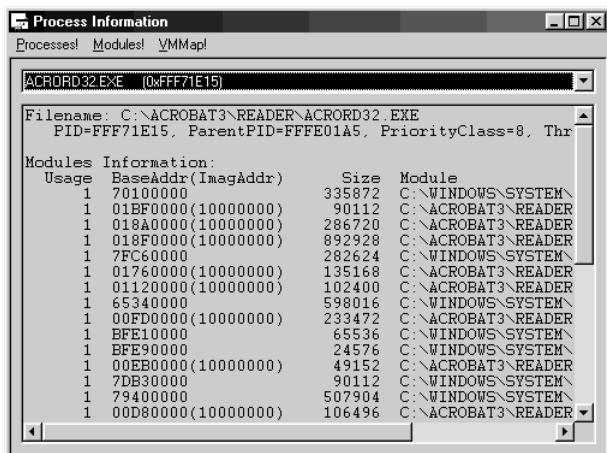
1. Моделирует создание адресного пространства процесса.
2. Открывает все модули, которые загружались бы в это адресное пространство, и получает предпочтительный базовый адрес и размер каждого модуля.
3. Моделирует переадресацию модулей в адресном пространстве, добиваясь того, чтобы модули не перекрывались.
4. В каждом модуле анализирует раздел переадресации и соответственно изменяет код в файле модуля на диске.
5. Записывает новый базовый адрес в заголовок файла.

Rebase — отличная утилита, и я настоятельно рекомендую Вам пользоваться ею. Вы должны запускать ее ближе к концу цикла сборки, когда уже созданы все модули приложения. Кроме того, применения утилиту Rebase, можно проигнорировать настройку базового адреса в диалоговом окне Project Settings. Она автоматически изменит базовый адрес 0x10000000 для DLL, задаваемый компоновщиком по умолчанию.

Но ни при каких обстоятельствах не модифицируйте базовые адреса системных модулей. Их адреса уже оптимизированы Microsoft, так что при загрузке в одно адресное пространство системные модули не перекрываются.

Я, кстати, добавил специальный инструмент в свою программу ProcessInfo.exe (см. главу 4). Он показывает список всех модулей, находящихся в адресном пространстве процесса. В колонке BaseAddr сообщается виртуальный адрес, по которому загружен модуль. Справа от BaseAddr расположена колонка ImagAddr. Обычно она пуста, указывая, что соответствующий модуль загружен по его предпочтительному базовому адресу. Так и должно быть для всех модулей. Однако, если в этой колонке существует адрес в скобках, значит, модуль загружен не по предпочтительному базовому адресу, и в колонке ImagAddr показывается базовый адрес, взятый из заголовка его файла на диске.

Ниже приведена информация о процессе Acrord32.exe, предоставленная моей программой ProcessInfo. Обратите внимание, что часть модулей загружена по предпочтительным базовым адресам, а часть — нет. Для последних сообщается один и тот же базовый адрес, 0x10000000; значит, автор этих DLL не подумал о проблемах модификации базовых адресов — пусть ему будет стыдно.



Связывание модулей

Модификация базовых адресов действительно очень важна и позволяет существенно повысить производительность всей системы. Но Вы можете сделать еще больше. Допустим, Вы должным образом модифицировали базовые адреса всех модулей своего приложения. Вспомните из главы 19, как загрузчик определяет адреса импортируемых идентификаторов: он записывает виртуальные адреса идентификаторов в раздел импорта EXE-модуля. Это позволяет, ссылаясь на импортируемые идентификаторы, обращаться к нужным участкам в памяти.

Давайте поразмыслим. Сохраняя виртуальные адреса импортируемых идентификаторов в разделе импорта EXE-модуля, загрузчик записывает их на те страницы памяти, где содержится этот раздел. Здесь включается в работу механизм копирования при записи, и их копии попадают в страничный файл. И у нас опять та же проблема, что и при модификации базовых адресов: отдельные части проекции модуля периодически сбрасываются в страничный файл и вновь подгружаются из него. Кроме того, загрузчику приходится преобразовывать адреса всех импортируемых идентификаторов (для каждого модуля), на что может понадобиться немалое время.

Для ускорения инициализации и сокращения объема памяти, занимаемого Вашим приложением, можно применить связывание модулей (*module binding*). Суть этой операции в том, что в раздел импорта модуля помещаются виртуальные адреса всех импортируемых идентификаторов. Естественно, она имеет смысл, только если проводится до загрузки модуля.

В Visual Studio есть еще одна утилита, Bind.exe. Информацию о том, как ею пользоваться, Вы получите, запустив Bind.exe без ключей в командной строке. Она описана в документации Platform SDK, и я не буду ее здесь детально рассматривать. Добавлю лишь, что в ней, как и в утилите Rebase, тоже нет ничего сверхъестественного: она просто вызывает функцию *BindImageEx* для каждого указанного файла. Вот что представляет собой эта функция:

```
BOOL BindImageEx(
    DWORD dwFlags,           // управляющие флаги
    PSTR pszImageName,       // полное имя обрабатываемого файла
    PSTR pszDllPath,         // путь для поиска образов файлов
    PSTR pszSymbolPath,      // путь для поиска отладочной информации
    PIMAGEHLP_STATUS_ROUTINE StatusRoutine); // функция обратного вызова
```

Последний параметр, *StatusRoutine*, — адрес функции обратного вызова, к которой периодически обращается *BindImageEx*, позволяя отслеживать процесс связывания. Прототип функции обратного вызова должен выглядеть так:

```
BOOL WINAPI StatusRoutine(
    IMAGEHLP_STATUS_REASON Reason, // причина неудачи
    PSTR pszImageName,          // полное имя обрабатываемого файла
    PSTR pszDllName,            // полное имя DLL
    ULONG_PTR VA,               // вычисленный виртуальный адрес
    ULONG_PTR Parameter);      // дополнительные сведения (зависят от значения Reason)
```

Когда Вы запускаете утилиту Bind, указывая ей нужный файл, она выполняет следующие операции.

1. Открывает раздел импорта указанного файла.
2. Открывает каждую DLL, указанную в разделе импорта, и просматривает ее заголовок, чтобы определить предпочтительный базовый адрес.
3. Отыскивает все импортируемые идентификаторы в разделе экспорта DLL.
4. Получает RVA (относительный виртуальный адрес) идентификатора, суммирует его с предпочтительным базовым адресом модуля и записывает полученное значение в раздел импорта обрабатываемого файла.
5. Вносит в раздел импорта модуля некоторую дополнительную информацию, включая имена всех DLL, с которыми связывается файл, и их временные метки.

В главе 19 мы исследовали раздел импорта Calc.exe с помощью утилиты DumpBin. В конце выведенного ею текста можно заметить информацию о связывании, добавленную при операции по п. 5. Вот эти строки:

```
Header contains the following bound import information:
Bound to SHELL32.dll [36E449E0] Mon Mar 08 14:06:24 1999
Bound to MSVCRT.dll [36BB8379] Fri Feb 05 15:49:13 1999
Bound to ADVAPI32.dll [36E449E1] Mon Mar 08 14:06:25 1999
Bound to KERNEL32.dll [36DDAD55] Wed Mar 03 13:44:53 1999
Bound to GDI32.dll [36E449E0] Mon Mar 08 14:06:24 1999
Bound to USER32.dll [36E449E0] Mon Mar 08 14:06:24 1999
```

Здесь видно, с какими модулями связан файл Calc.exe, а номер в квадратных скобках идентифицирует время создания каждого DLL-модуля. Это 32-разрядное значение расшифровывается и отображается за квадратными скобками в более привычном нам виде.

Утилита Bind использует два важных правила.

- При инициализации процесса все необходимые DLL действительно загружаются по своим предпочтительным базовым адресам. Вы можете соблюсти это правило, применив утилиту Rebase.
- Адреса идентификаторов в разделе экспорта остаются неизменными со временем последнего связывания. Загрузчик проверяет это, сравнивая временную метку каждой DLL со значением, сохраненным при операции по п. 5.

Конечно, если загрузчик обнаружит, что нарушено хотя бы одно из правил, он решит, что Bind не справилась со своей задачей, и самостоятельно модифицирует раздел импорта исполняемого модуля (по обычной процедуре). Но если загрузчик увидит, что модуль связан, нужные DLL загружены по предпочтительным базовым

адресам и временные метки корректны, он фактически ничего делать не будет, и приложение сможет немедленно начать свою работу!

Кроме того, приложение не потребует лишнего места в страничном файле. И очень жаль, что многие коммерческие приложения поставляются без должной модификации базовых адресов и связывания.

О'кэй, теперь Вы знаете, что все модули приложения нужно связывать. Но вот вопрос: когда? Если Вы свяжете модули в своей системе, Вы привяжете их к системным DLL, установленным на Вашем компьютере, а у пользователя могут быть установлены другие версии DLL. Поскольку Вам заранее не известно, в какой операционной системе (Windows 98, Windows NT или Windows 2000) будет запускаться Ваше приложение и какие сервисные пакеты в ней установлены, связывание нужно проводить в процессе установки приложения.

Естественно, если пользователь применяет конфигурацию с альтернативной загрузкой Windows 98 и Windows 2000, то для одной из операционных систем модули будут связаны неправильно. Тот же эффект даст и обновление операционной системы установкой в ней сервисного пакета. Эту проблему ни Вам, ни тем более пользователю решить не удастся. Microsoft следовало бы поставлять с операционной системой утилиту, которая автоматически проводила бы повторное связывание всех модулей после обновления системы. Но, увы, такой утилиты нет.

Локальная память потока

Иногда данные удобно связывать с экземпляром какого-либо объекта. Например, чтобы сопоставить какие-то дополнительные данные с окном, применяют функции *SetWindowWord* и *SetWindowLong*. Локальная память потока (thread-local storage, TLS) позволяет связать данные и с определенным потоком (скажем, сопоставить с ним время его создания), а по завершении этого потока вычислить время его жизни.

TLS также используется в библиотеке C/C++. Но эту библиотеку разработали задолго до появления многопоточных приложений, и большая часть содержащихся в ней функций рассчитана на однопоточные программы. Наглядный пример — функция *strtok*. При первом вызове она получает адрес строки и запоминает его в собственной статической переменной. Когда при следующих вызовах *strtok* Вы передаете ей NULL, она оперирует с адресом, записанным в своей переменной.

В многопоточной среде вероятна такая ситуация: один поток вызывает *strtok*, и, не успел он вызвать ее повторно, как к ней уже обращается другой. Тогда второй поток заставит функцию занести в статическую переменную новый адрес, неизвестный первому. И в дальнейшем первый поток, вызывая *strtok*, будет использовать строку, принадлежащую второму. Вот Вам и «жучок», найти который очень трудно.

Чтобы устраниТЬ эту проблему, в библиотеке C/C++ теперь применяется механизм локальной памяти потока: за каждым потоком закрепляется свой строковый указатель, зарезервированный для *strtok*. Аналогичный механизм действует и для других библиотечных функций, в том числе *asctime* и *gmtime*.

Локальная память потока может быть той соломинкой, за которую придется ухватиться, если Ваша программа интенсивно использует глобальные или статические переменные. К счастью, сейчас наметилась тенденция отхода от применения таких переменных и перехода к автоматическим (размещаемым в стеке) переменным и передаче данных через параметры функций. И правильно: ведь расположенные в стеке переменные всегда связаны только с конкретным потоком.

Стандартная библиотека С существует уже долгие годы — это и хорошо, и плохо. Ее переделывали под многие компиляторы, и ни один из них без нее не стоил бы ломаного гроша. Программисты пользовались и будут пользоваться ею, а значит, прототипы и поведение функций вроде *strtok* останутся прежними. Но если бы эту библиотеку взялись перерабатывать сегодня, ее построили бы с учетом многопоточности и уж точно не стали бы применять глобальные и статические переменные.

В своих программах я стараюсь избегать глобальных переменных. Если же Вы используете глобальные и статические переменные, советую проанализировать каждую из них и подумать, нельзя ли заменить ее переменной, размещаемой в стеке. Усилия окупятся сторицей, когда Вы решите создать в программе дополнительные потоки; впрочем, и однопоточное приложение лишь выиграет от этого.

Хотя два вида TLS-памяти, рассматриваемые в этой главе, применимы как в приложениях, так и в DLL, они все же полезнее при разработке DLL, поскольку именно в

в этом случае Вам не известна структура программы, с которой они будут связаны. Если же Вы пишете приложение, то обычно знаете, сколько потоков оно создаст и для чего. Поэтому здесь еще можно как-то вывернуться. Но разработчик DLL ничего этого не знает. Чтобы помочь ему, и был создан механизм локальной памяти потока. Однако сведения, изложенные в этой главе, пригодятся и разработчику приложений.

Динамическая локальная память потока

Приложение работает с динамической локальной памятью потока, оперируя набором из четырех функций. Правда, чаще с ними работают DLL-, а не EXE-модули. На рис. 21-1 показаны внутренние структуры данных, используемые для управления TLS в Windows.

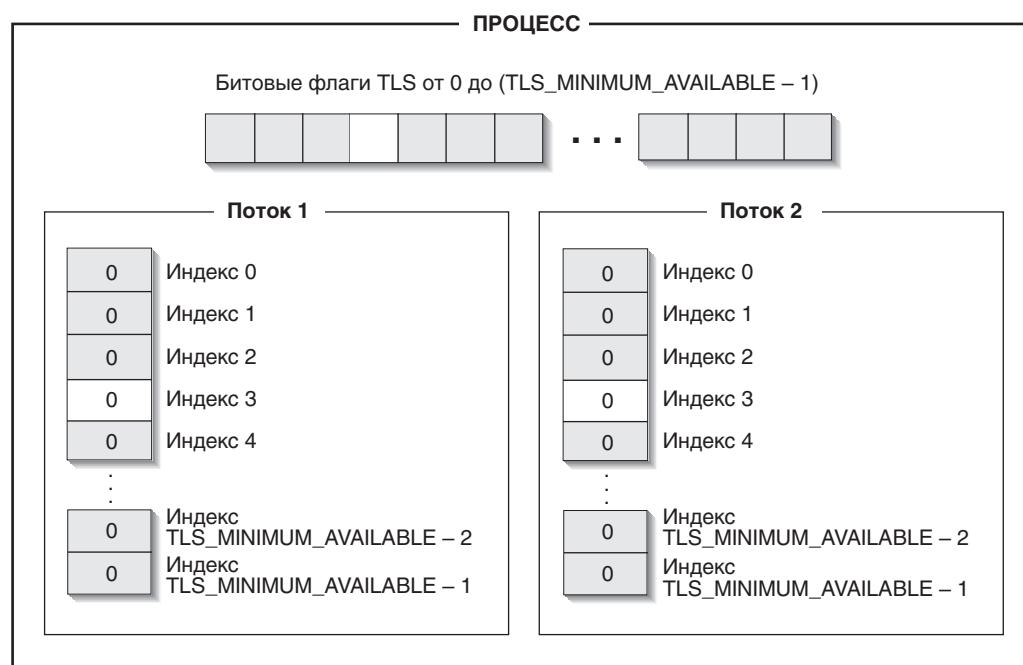


Рис. 21-1. Внутренние структуры данных, предназначенные для управления локальной памятью потока

Каждый флаг выполняемого в системе процесса может находиться в состоянии FREE или INUSE, указывая, свободна или занята данная область локальной памяти потока (TLS-область). Microsoft гарантирует доступность по крайней мере TLS_MINIMUM_AVAILABLE битовых флагов. Идентификатор TLS_MINIMUM_AVAILABLE определен в файле WinNT.h как 64. Но в Windows 2000 этот флаговый массив вмещает свыше 1000 элементов! Этого более чем достаточно для любого приложения.

Чтобы воспользоваться динамической TLS, вызовите сначала функцию *TlsAlloc*:

```
DWORD TlsAlloc();
```

Она заставляет систему сканировать битовые флаги в текущем процессе и искать флаг FREE. Отыскав, система меняет его на INUSE, а *TlsAlloc* возвращает индекс флага в битовом массиве. DLL (или приложение) обычно сохраняет этот индекс в глобальной переменной. Не найдя в списке флаг FREE, *TlsAlloc* возвращает код TLS_OUT_OF_INDEXES (определенный в файле WinBase.h как 0xFFFFFFFF).

Когда *TlsAlloc* вызывается впервые, система узнает, что первый флаг — FREE, и немедленно меняет его на INUSE, а *TlsAlloc* возвращает 0. Вот 99 процентов того, что делает *TlsAlloc*. Об оставшемся одном проценте мы поговорим позже.

Создавая поток, система создает и массив из *TLS_MINIMUM_AVAILABLE* элементов — значений типа *PVOID*; она инициализирует его нулями и сопоставляет с потоком. Таким массивом (элементы которого могут принимать любые значения) располагает каждый поток (рис. 21-1).

Прежде чем сохранить что-то в *PVOID*-массиве потока, выясните, какой индекс в нем доступен, — этой цели и служит предварительный вызов *TlsAlloc*. Фактически она резервирует какой-то элемент этого массива. Скажем, если возвращено значение 3, то в Вашем распоряжении третий элемент *PVOID*-массива в каждом потоке данного процесса — не только в выполняемых сейчас, но и в тех, которые могут быть созданы в будущем.

Чтобы занести в массив потока значение, вызовите функцию *TlsSetValue*:

```
BOOL TlsSetValue(
    DWORD dwTlsIndex,
    PVOID pvTlsValue);
```

Она помещает в элемент массива, индекс которого определяется параметром *dwTlsIndex*, значение типа *PVOID*, содержащееся в параметре *pvTlsValue*. Содержимое *pvTlsValue* сопоставляется с потоком, вызвавшим *TlsSetValue*. В случае успеха возвращается TRUE.

Обращаясь к *TlsSetValue*, поток изменяет только свой *PVOID*-массив. Он не может что-то изменить в локальной памяти другого потока. Лично мне хотелось бы видеть какую-нибудь TLS-функцию, которая позволила бы одному потоку записывать данные в массив другого потока, но такой нет. Сейчас единственный способ пересылки каких-либо данных от одного потока другому — передать единственное значение через *CreateThread* или *_beginthreadex*. Те в свою очередь передают это значение функции потока.

Вызывая *TlsSetValue*, будьте осторожны и передавайте только тот индекс, который получен предыдущим вызовом *TlsAlloc*. Чтобы максимально увеличить быстродействие этих функций, Microsoft отказалась от контроля ошибок. Если Вы передадите индекс, не зарезервированный ранее *TlsAlloc*, система все равно запишет в соответствующий элемент массива значение, и тогда ждите неприятностей.

Для чтения значений из массива потока служит функция *TlsGetValue*:

```
PVOID TlsGetValue(DWORD dwTlsIndex);
```

Она возвращает значение, сопоставленное с TLS-областью под индексом *dwTlsIndex*. Как и *TlsSetValue*, функция *TlsGetValue* обращается только к массиву, который принадлежит вызывающему потоку. Она тоже не контролирует допустимость передаваемого индекса.

Когда необходимость в TLS-области у всех потоков в процессе отпадет, вызовите *TlsFree*:

```
BOOL TlsFree(DWORD dwTlsIndex);
```

Эта функция просто сообщит системе, что данная область больше не нужна. Флаг INUSE, управляемый массивом битовых флагов процесса, установится как FREE, и в будущем, когда поток еще раз вызовет *TlsAlloc*, этот участок памяти окажется вновь доступен. *TlsFree* возвращает TRUE, если вызов успешен. Попытка освобождения невыделенной TLS-области даст ошибку.

Использование динамической TLS

Обычно, когда в DLL применяется механизм TLS-памяти, вызов *DllMain* со значением *DLL_PROCESS_ATTACH* заставляет DLL обратиться к *TlsAlloc*, а вызов *DllMain* со значением *DLL_PROCESS_DETACH* — к *TlsFree*. Вызовы *TlsSetValue* и *TlsGetValue* чаще всего происходят при обращении к функциям, содержащимся в DLL.

Вот один из способов работы с TLS-памятью: Вы создаете ее только по необходимости. Например, в DLL может быть функция, работающая аналогично *strtok*. При первом ее вызове поток передает этой функции указатель на 40-байтовую структуру, которую надо сохранить, чтобы ссылаться на нее при последующих вызовах. Поэтому Вы пишете свою функцию, скажем, так:

```
DWORD g_dwTlsIndex; // считаем, что эта переменная инициализируется
                     // в результате вызова функции TlsAlloc
:
void MyFunction(PSOMESTRUCT pSomeStruct) {
    if (pSomeStruct != NULL) {
        // вызывающий поток передает в функцию какие-то данные

        // проверяем, не выделена ли уже область для хранения этих данных
        if (TlsGetValue(g_dwTlsIndex) == NULL) {
            // еще не выделена; функция вызывается этим потоком впервые
            TlsSetValue(g_dwTlsIndex, HeapAlloc(GetProcessHeap(), 0,
                                              sizeof(*pSomeStruct)));
        }
        // память уже выделена; сохраняем только что переданные значения
        memcpy(TlsGetValue(g_dwTlsIndex), pSomeStruct, sizeof(*pSomeStruct));
    }
    // вызывающий код уже передал функции данные;
    // теперь что-то делаем с ними

    // получаем адрес записанных данных
    pSomeStruct = (PSOMESTRUCT) TlsGetValue(g_dwTlsIndex);

    // на эти данные указывает pSomeStruct; используем ее
    :
}
}
```

Если поток приложения никогда не вызовет *MyFunction*, то и блок памяти никогда не будет выделен.

Если Вам показалось, что 64 TLS-области — слишком много, напомню: приложение может динамически подключать несколько DLL. Одна DLL займет, допустим, 10 TLS-индексов, вторая — 5 и т. д. Так что это вовсе не много — напротив, стремитесь к тому, чтобы DLL использовала минимальное число TLS-индексов. И для этого лучше всего применять метод, показанный на примере функции *MyFunction*. Конечно, я могу сохранить 40-байтовую структуру в 10 TLS-индексах, но тогда не только будет попусту расходоваться TLS-массив, но и затруднится работа с данными. Гораздо эффективнее выделить отдельный блок памяти для данных, сохранив указатель на него в одном TLS-индексе, — именно так и делается в *MyFunction*. Как я уже упомянул, в Windows 2000 количество TLS-областей увеличено до более чем 1000. Microsoft пошла на

это из-за того, что многие разработчики слишком бесцеремонно использовали TLS-области и их не хватало другим DLL.

Теперь вернемся к тому единственному проценту, о котором я обещал рассказать, рассматривая *TlsAlloc*. Взгляните на фрагмент кода:

```
DWORD dwTlsIndex;
PVOID pvSomeValue;
:
dwTlsIndex = TlsAlloc();
TlsSetValue(dwTlsIndex, (PVOID) 12345);
TlsFree(dwTlsIndex);

// допустим, значение dwTlsIndex, возвращенное после этого вызова TlsAlloc,
// идентично индексу, полученному при предыдущем вызове TlsAlloc
dwTlsIndex = TlsAlloc();

pvSomeValue = TlsGetValue(dwTlsIndex);
```

Как Вы думаете, что содержится в *pvSomeValue* после выполнения этого кода? 12345? Нет — нуль. Прежде чем вернуть управление, *TlsAlloc* «проходит» по всем потокам в процессе и заносит 0 по только что выделенному индексу в массив каждого потока. И прекрасно! Ведь не исключено, что приложение вызовет *LoadLibrary*, чтобы загрузить DLL, а последняя — *TlsAlloc*, чтобы зарезервировать какой-то индекс. Далее поток может обратиться к *FreeLibrary* и удалить DLL. Последняя должна освободить выделенный ей индекс, вызвав *TlsFree*, но кто знает, какие значения код DLL занес в тот или иной TLS-массив? В следующее мгновение поток вновь вызывает *LoadLibrary* и загружает другую DLL, которая тоже обращается к *TlsAlloc* и получает тот же индекс, что и предыдущая DLL. И если бы *TlsAlloc* не делала того, о чём я упомянул в самом начале, поток мог бы получить старое значение элемента, и программа стала бы работать некорректно.

Допустим, DLL, загруженная второй, решила проверить, выделена ли какому-то потоку локальная память, и вызвала *TlsGetValue*, как в предыдущем фрагменте кода. Если бы *TlsAlloc* не очищала соответствующий элемент в массиве каждого потока, то в этих элементах оставались бы старые данные от первой DLL. И тогда было бы вот что. Поток обращается к *MyFunction*, а та — в полной уверенности, что блок памяти уже выделен, — вызывает *memcp* и таким образом копирует новые данные в ту область, которая, как ей кажется, и является выделенным блоком. Результат мог бы быть катастрофическим. К счастью, *TlsAlloc* инициализирует элементы массива, и такое просто немыслимо.

Статическая локальная память потока

Статическая локальная память потока основана на той же концепции, что и динамическая, — она предназначена для того, чтобы с потоком можно было сопоставить те или иные данные. Однако статическую TLS использовать гораздо проще, так как при этом не нужно обращаться к каким-либо функциям.

Возьмем такой пример: Вы хотите сопоставлять стартовое время с каждым потоком, создаваемым программой. В этом случае нужно лишь объявить переменную для хранения стартового времени:

```
_declspec(thread) DWORD gt_dwStartTime = 0;
```

Префикс `_declspec(thread)` — модификатор, поддерживаемый компилятором Microsoft Visual C++. Он сообщает компилятору, что соответствующую переменную следует поместить в отдельный раздел EXE- или DLL-файла. Переменная, указываемая за `_declspec(thread)`, должна быть либо глобальной, либо статической внутри (или вне) функции. Локальную переменную с модификатором `_declspec(thread)` объявить нельзя. Но это не должно Вас беспокоить, ведь локальные переменные и так связаны с конкретным потоком. Кстати, глобальные TLS-переменные я помечаю префиксом `gt_`, а статические — `st_`.

Обрабатывая программу, компилятор выносит все TLS-переменные в отдельный раздел, и Вы вряд ли удивитесь, что этому разделу присваивается имя `.tls`. Компоновщик объединяет эти разделы из разных объектных модулей и создает в итоге один большой раздел `.tls`, помещаемый в конечный EXE- или DLL-файл.

Работа статической TLS строится на тесном взаимодействии с операционной системой. Загружая приложение в память, система отыскивает в EXE-файле раздел `.tls` и динамически выделяет блок памяти для хранения всех статических TLS-переменных. Всякий раз, когда Ваша программа ссылается на одну из таких переменных, ссылка переадресуется к участку, расположенному в выделенном блоке памяти. В итоге компилятору приходится генерировать дополнительный код для ссылок на статические TLS-переменные, что увеличивает размер приложения и замедляет скорость его работы. В частности, на процессорах `x86` каждая ссылка на статическую TLS-переменную заставляет генерировать три дополнительные машинные команды.

Если в процессе создается другой поток, система выделяет еще один блок памяти для хранения статических переменных нового потока. Только что созданный поток имеет доступ лишь к своим статическим TLS-переменным, и не может обратиться к TLS-переменным любого другого потока.

Вот так в общих чертах и работает статическая TLS-память. Теперь посмотрим, что происходит при участии DLL. Ведь скорее всего Ваша программа, использующая статические TLS-переменные, связывается с какой-нибудь DLL, в которой тоже применяются переменные этого типа. Загружая такую программу, система сначала определяет объем ее раздела `.tls`, а затем добавляет эту величину к сумме размеров всех разделов `.tls`, содержащихся в DLL, которые связаны с Вашей программой. При создании потоков система автоматически выделяет блок памяти, достаточно большой, чтобы в нем уместились все TLS-переменные, необходимые как приложению, так и неявно связываемым с ней DLL. Все так хорошо, что даже не верится!

И не верьте! Подумайте, что будет, если приложение вызовет `LoadLibrary` и подключит DLL, тоже содержащую статические TLS-переменные. Системе придется проверить потоки, уже существующие в процессе, и увеличить их блоки TLS-памяти, чтобы подогнать эти блоки под дополнительные требования, предъявляемые новой DLL. Ну а если Вы вызовете `FreeLibrary` для выгрузки DLL со статическими TLS-переменными, системе придется ужать блоки памяти, сопоставленные с потоками в данном процессе.

Это слишком большая нагрузка на операционную систему. Кроме того, допуская явную загрузку DLL, содержащих статические TLS-переменные, система не в состоянии должным образом инициализировать TLS-данные, что при попытке обращения к ним может вызвать нарушение доступа. Это, пожалуй, единственный недостаток статической TLS; при использовании динамической TLS такой проблемы нет. DLL, работающие с динамической TLS, могут загружаться и выгружаться из выполняемой программы в любой момент и без всяких проблем.

Внедрение DLL и перехват API-вызовов

В среде Windows каждый процесс получает свое адресное пространство. Указатели, используемые Вами для ссылки на определенные участки памяти, — это адреса в адресном пространстве Вашего процесса, и в нем нельзя создать указатель, ссылающийся на память, принадлежащую другому процессу. Так, если в Вашей программе есть «жучок», из-за которого происходит запись по случайному адресу, он не разрушит содержимое памяти, отведенной другим процессам.

WINDOWS 98 В Windows 98 процессы фактически совместно используют 2 Гб адресного пространства (от 0x80000000 до 0xFFFFFFFF). На этот регион отображаются только системные компоненты и файлы, проецируемые в память (подробнее на эту тему см. главы 13, 14 и 17).

Раздельные адресные пространства очень выгодны и разработчикам, и пользователям. Первым важно, что Windows перехватывает обращения к памяти по случайным адресам, вторым — что операционная система более устойчива и сбой одного приложения не приведет к краху другого или самой системы. Но, конечно, за надежность приходится платить: написать программу, способную взаимодействовать с другими программами или манипулировать другими процессами, теперь гораздо сложнее.

Вот ситуации, в которых требуется прорыв за границы процессов и доступ к адресному пространству другого процесса:

- создание подкласса окна, порожденного другим процессом;
- получение информации для отладки (например, чтобы определить, какие DLL используются другим процессом);
- установка ловушек (hooks) в других процессах.

В этой главе я расскажу о нескольких механизмах, позволяющих внедрить (*inject*) какую-либо DLL в адресное пространство другого процесса. Ваш код, попав в чужое адресное пространство, может устроить в нем настоящий хаос, поэтому хорошенько взвесьте, так ли Вам необходимо это внедрение.

Пример внедрения DLL

Допустим, Вы хотите создать подкласс от экземпляра окна, порожденного другим процессом. Это, как Вы помните, позволит изменять поведение окна. Все, что от Вас для этого требуется, — вызвать функцию *SetWindowLongPtr*, чтобы заменить адрес оконной процедуры в блоке памяти, принадлежащем окну, новым — указывающим на Вашу функцию *WndProc*. В документации Platform SDK утверждается, что приложение

не может создать подкласс окна другого процесса. Это не совсем верно. Проблема создания подкласса окна из другого процесса на самом деле сводится к преодолению границ адресного пространства.

Вызывая `SetWindowLongPtr` для создания подкласса окна (как показано ниже), Вы говорите системе, что все сообщения окну, на которое указывает `hwnd`, следует направлять не обычной оконной процедуре, а функции `MySubclassProc`.

```
SetWindowLongPtr(hwnd, GWLP_WNDPROC, MySubclassProc);
```

Иными словами, когда системе надо передать сообщение процедуре `WndProc` указанного окна, она находит ее адрес и вызывает напрямую. В нашем примере система видит, что с окном сопоставлен адрес функции `MySubclassProc`, и поэтому вызывает именно ее, а не исходную оконную процедуру.

Проблема с созданием подкласса окна, принадлежащего другому процессу, состоит в том, что процедура подкласса находится в чужом адресном пространстве. Упрощенная схема приема сообщений оконной процедурой представлена на рис. 22-1. Процесс А создает окно. На адресное пространство этого процесса проецируется файл User32.dll. Эта проекция User32.dll отвечает за прием и диспетчеризацию сообщений (синхронных и асинхронных), направляемых любому из окон, созданных потоками процесса А. Обнаружив какое-то сообщение, она определяет адрес процедуры `WndProc` окна и вызывает ее, передавая описатель окна, сообщение и параметры `wParam` и `lParam`. Когда `WndProc` обработает сообщение, User32.dll вернется в начало цикла и будет ждать следующее оконное сообщение.

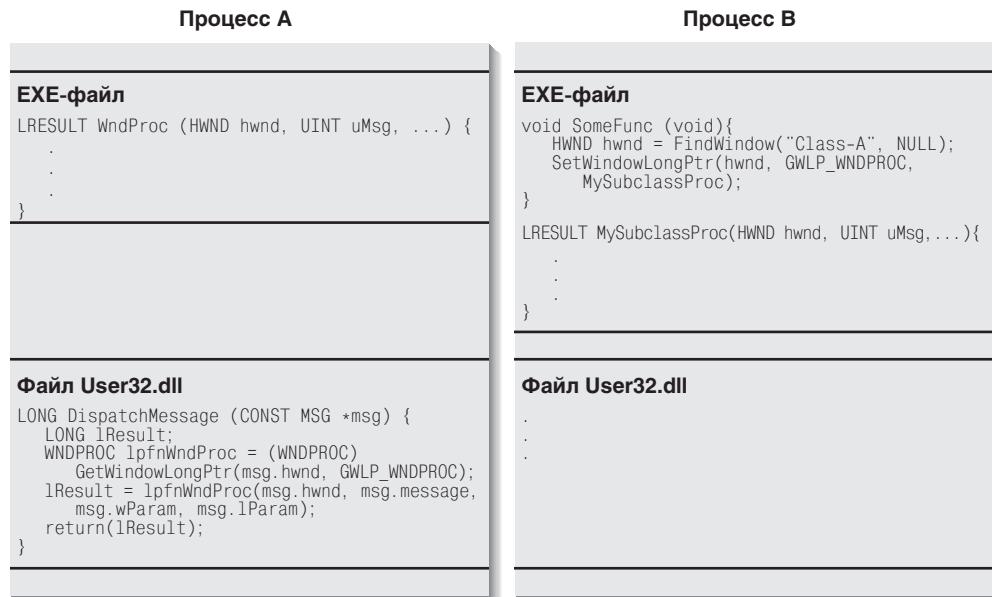


Рис. 22-1. Поток процесса В пытается создать подкласс окна, сформированного потоком процесса А

Теперь допустим, что процесс В хочет создать подкласс окна, порожденного одним из потоков процесса А. Сначала код процесса В должен определить описатель этого окна, что можно сделать самыми разными способами. В примере на рис. 22-1 поток процесса В просто вызывает `FindWindow`, затем — `SetWindowLongPtr`, пытаясь изменить адрес процедуры `WndProc` окна. Обратите внимание: *пытаясь*. Этот вызов не даст ничего, кроме `NULL`. Функция `SetWindowLongPtr` просто проверяет, не хочет

ли процесс изменить адрес *WndProc* окна, созданного другим процессом, и, если да, игнорирует вызов.

А если бы функция *SetWindowLongPtr* могла изменить адрес *WndProc*? Система тогда связала бы адрес процедуры *MySubclassProc* с указанным окном. Затем при посылке сообщения этому окну код User32 в процессе А извлек бы данное сообщение, получил адрес *MySubclassProc* и попытался бы вызвать процедуру по этому адресу. Но это привело бы к крупным неприятностям, так как *MySubclassProc* находится в адресном пространстве процесса В, а активен — процесс А. Очевидно, если бы User32 обратился по данному адресу, то на самом деле он обратился бы к какому-то участку памяти в адресном пространстве процесса А, что, естественно, привело бы к нарушению доступа к памяти.

Чтобы избежать этого, было бы неплохо сообщить системе, что *MySubclassProc* находится в адресном пространстве процесса В, и тогда она переключила бы контекст перед вызовом процедуры подкласса. Увы, по ряду причин такая функциональность в системе не реализована.

- Подклассы окон, созданных потоками других процессов, порождаются весьма редко. Большинство приложений делает это лишь применительно к собственным окнам, и архитектура памяти в Windows этому не препятствует.
- Переключение активных процессов отнимает слишком много процессорного времени.
- Код *MySubclassProc* должен был бы выполняться потоком процесса В, но каким именно — новым или одним из существующих?
- Как User32.dll узнает, с каким процессом связан адрес оконной процедуры?

Поскольку удачных решений этих проблем нет, Microsoft предпочла запретить функции *SetWindowLongPtr* замену процедуры окна, созданного другим процессом.

Тем не менее порождение подкласса окна, созданного чужим процессом, возможно: нужно просто пойти другим путем. Ведь на самом деле проблема не столько в создании подкласса, сколько в закрытости адресного пространства процесса. Если бы Вы могли как-то поместить код своей оконной процедуры в адресное пространство процесса А, это позволило бы вызвать *SetWindowLongPtr* и передать ей адрес *MySubclassProc* в процессе А. Я называю такой прием внедрением (*injecting*) DLL в адресное пространство процесса. Мне известно несколько способов подобного внедрения. Рассмотрим их по порядку, начиная с простейшего.

Внедрение DLL с использованием реестра

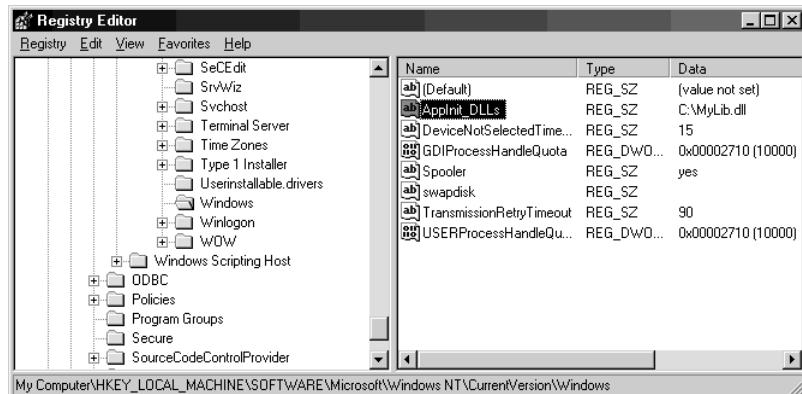
Если Вы уже работали с Windows, то знаете, что такое реестр. В нем хранится конфигурация всей системы, и, модифицируя в реестре те или иные параметры, можно изменить поведение системы. Я намерен поговорить о параметре реестра:

```
HKEY_LOCAL_MACHINE\Software\Microsoft
    \Windows NT\CurrentVersion\Windows\AppInit_DLLs
```

WINDOWS 98 Windows 98 игнорирует этот параметр реестра, поэтому для нее такой способ внедрения DLL не сработает.

Список параметров в разделе реестра, где находится AppInit_DLLs, можно просмотреть с помощью программы Registry Editor (Редактор реестра). Значением параметра AppInit_DLLs может быть как имя одной DLL (с указанием пути доступа), так и имена

нескольких DLL, разделенных пробелами или запятыми. Поскольку пробел используется здесь в качестве разделителя, в именах файлов не должно быть пробелов. Система считывает путь только первой DLL в списке — пути остальных DLL игнорируются, поэтому лучше размещать свои DLL в системном каталоге Windows, чтобы не указывать пути. Как видите, я указал в параметре AppInit_DLLs только одну DLL и задал путь к ней: C:\MyLib.dll.



При следующей перезагрузке компьютера Windows сохранит значение этого параметра. Далее, когда User32.dll будет спроектирован на адресное пространство процесса, этот модуль получит уведомление DLL_PROCESS_ATTACH и после его обработки вызовет *LoadLibrary* для всех DLL, указанных в параметре AppInit_DLLs. В момент загрузки каждая DLL инициализируется вызовом ее функции *DllMain* с параметром *dwReason*, равным DLL_PROCESS_ATTACH. Поскольку внедряемая DLL загружается на такой ранней стадии создания процесса, будьте особенно осторожны при вызове функций. Проблем с вызовом функций Kernel32.dll не должно быть, но в случае других DLL они вполне вероятны — User32.dll не проверяет, успешно ли загружены и инициализированы эти DLL. Правда, в Windows 2000 модуль User32.dll ведет себя несколько иначе, но об этом — чуть позже.

Это простейший способ внедрения DLL. Все, что от Вас требуется, — добавить значение в уже существующий параметр реестра. Однако он не лишен недостатков.

- Так как система считывает значение параметра при инициализации, после его изменения придется перезагружать компьютер. Выход и повторный вход в систему не сработает — Вы должны перезагрузить компьютер. Впрочем, сканное относится лишь к Windows NT версии 4.0 (или ниже). В Windows 2000 модуль User32.dll повторно считывает параметр реестра AppInit_DLLs при каждой загрузке в процесс, и перезапуска системы не требуется.
- Ваша DLL проецируется на адресные пространства только тех процессов, на которые спроектирован и модуль User32.dll. Его используют все GUI-приложения, но большинство программ консольного типа — нет. Поэтому такой метод не годится для внедрения DLL, например, в компилятор или компоновщик.
- Ваша DLL проецируется на адресные пространства всех GUI-процессов. Но Вам-то почти наверняка надо внедрить DLL только в один или несколько определенных процессов. Чем больше процессов попадет «под тень» такой DLL, тем выше вероятность аварийной ситуации. Ведь теперь Ваш код выполняется потоками этих процессов, и, если он зациклится или некорректно обратится к памяти, Вы повлияете на поведение и устойчивость соответствующих про-

цессов. Поэтому лучше внедрять свою DLL в как можно меньшее число процессов.

- Ваша DLL проецируется на адресное пространство каждого GUI-процесса в течение всей его жизни. Тут есть некоторое сходство с предыдущей проблемой. Желательно не только внедрять DLL в минимальное число процессов, но и проецировать ее на эти процессы как можно меньшее время. Допустим, Вы хотите создать подкласс главного окна WordPad в тот момент, когда пользователь запускает Ваше приложение. Естественно, пока пользователь не откроет Ваше приложение, внедрять DLL в адресное пространство WordPad не требуется. Когда пользователь закроет Ваше приложение, целесообразно отменить переопределение оконной процедуры WordPad. И в этом случае DLL тоже не засчет «держать» в адресном пространстве WordPad. Так что лучшее решение — внедрять DLL только на то время, в течение которого она действительно нужна конкретной программе.

Внедрение DLL с помощью ловушек

Внедрение DLL в адресное пространство процесса возможно и с применением ловушек. Чтобы они работали так же, как и в 16-разрядной Windows, Microsoft пришлось создать механизм, позволяющий внедрять DLL в адресное пространство другого процесса. Рассмотрим его на примере.

Процесс A (вроде утилиты Spy++) устанавливает ловушку WH_GETMESSAGE и наблюдает за сообщениями, которые обрабатываются окнами в системе. Ловушка устанавливается вызовом *SetWindowsHookEx*:

```
HHOOK hHook = SetWindowsHookEx(WH_GETMESSAGE, GetMsgProc, hinstDll, 0);
```

Аргумент *WH_GETMESSAGE* определяет тип ловушки, а параметр *GetMsgProc* — адрес функции (в адресном пространстве Вашего процесса), которую система должна вызывать всякий раз, когда окно собирается обработать сообщение. Параметр *hinstDll* идентифицирует DLL, содержащую функцию *GetMsgProc*. В Windows значение *hinstDll* для DLL фактически задает адрес в виртуальной памяти, по которому DLL спроектирована на адресное пространство процесса. И, наконец, последний аргумент, 0, указывает поток, для которого предназначена ловушка. Поток может вызвать *SetWindowsHookEx* и передать ей идентификатор другого потока в системе. Передавая 0, мы сообщаем системе, что ставим ловушку для всех существующих в ней GUI-потоков.

Теперь посмотрим, как все это действует:

1. Поток процесса B собирается направить сообщение какому-либо окну.
2. Система проверяет, не установлена ли для данного потока ловушка *WH_GETMESSAGE*.
3. Затем выясняет, спроектирована ли DLL, содержащая функцию *GetMsgProc*, на адресное пространство процесса B.
4. Если указанная DLL еще не спроектирована, система отображает ее на адресное пространство процесса B и увеличивает счетчик блокировок (lock count) проекции DLL в процессе B на 1.
5. Система проверяет, не совпадают ли значения *hinstDll* этой DLL, относящиеся к процессам A и B. Если *hinstDll* в обоих процессах одинаковы, то и адрес *GetMsgProc* в этих процессах тоже одинаков. Тогда система может просто вызвать *GetMsgProc* в адресном пространстве процесса A. Если же *hinstDll* различ-

ны, система определяет адрес функции *GetMsgProc* в адресном пространстве процесса В по формуле:

$$\text{GetMsgProc B} = \text{hinstDl}l\ A + (\text{GetMsgProc A} - \text{hinstDl}l\ A)$$

Вычитая *hinstDl}l\ A* из *GetMsgProc A*, Вы получаете смещение (в байтах) адреса функции *GetMsgProc*. Добавляя это смещение к *hinstDl}l\ B*, Вы получаете адрес *GetMsgProc*, соответствующий проекции DLL в адресном пространстве процесса В.

6. Счетчик блокировок проекции DLL в процессе В увеличивается на 1.
7. Вызывается *GetMsgProc* в адресном пространстве процесса В.
8. После возврата из *GetMsgProc* счетчик блокировок проекции DLL в адресном пространстве процесса В уменьшается на 1.

Кстати, когда система внедряет или проецирует DLL, содержащую функцию фильтра ловушки, проецируется вся DLL, а не только эта функция. А значит, потокам, выполняемым в контексте процесса В, теперь доступны все функции такой DLL.

Итак, чтобы создать подкласс окна, сформированного потоком другого процесса, можно сначала установить ловушку *WH_GETMESSAGE* для этого потока, а затем — когда будет вызвана функция *GetMsgProc* — обратиться к *SetWindowLongPtr* и создать подкласс. Разумеется, процедура подкласса должна быть в той же DLL, что и *GetMsgProc*.

В отличие от внедрения DLL с помощью реестра этот способ позволяет в любой момент отключить DLL от адресного пространства процесса, для чего достаточно вызвать:

```
BOOL UnhookWindowsHookEx(HHOOK hHook);
```

Когда поток обращается к этой функции, система просматривает внутренний список процессов, в которые ей пришлось внедрить данную DLL, и уменьшает счетчик ее блокировок на 1. Как только этот счетчик обнуляется, DLL автоматически выгружается. Вспомните: система увеличивает его непосредственно перед вызовом *GetMsgProc* (см. выше п. 6). Это позволяет избежать нарушения доступа к памяти. Если бы счетчик не увеличивался, то другой поток мог бы вызвать *UnhookWindowsHookEx* в тот момент, когда поток процесса В пытается выполнить код *GetMsgProc*.

Все это означает, что нельзя создать подкласс окна и тут же убрать ловушку — она должна действовать в течение всей жизни подкласса.

Утилита для сохранения позиций элементов на рабочем столе

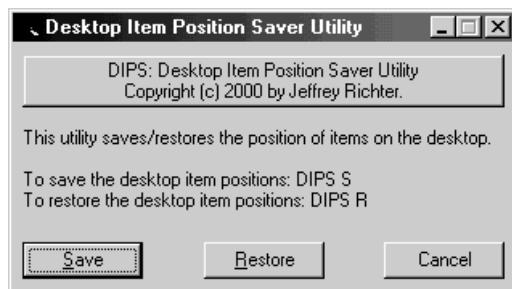
Эта утилита, «22 DIPS.exe» (см. листинг на рис. 22-2), использует ловушки окон для внедрения DLL в адресное пространство Explorer.exe. Файлы исходного кода и ресурсов этой программы и DLL находятся в каталогах 22-DIPS и 22-DIPSLib на компакт-dиске, прилагаемом к книге.

Компьютер я использую в основном для работы, и, на мой взгляд, самое оптимальное в этом случае разрешение экрана — 1152 × 864. Иногда я запускаю на своем компьютере кое-какие игры, но большинство из них рассчитано на разрешение 640 × 480. Когда у меня появляется настроение поиграть, приходится открывать апплет Display в Control Panel и устанавливать разрешение 640 × 480, а закончив игру, вновь возвращаться в Display и восстанавливать разрешение 1152 × 864.

Возможность изменять экранное разрешение «на лету» — очень удобная функция Windows. Единственное, что мне не по душе, — при смене экранного разрешения не сохраняются позиции ярлыков на рабочем столе. У меня на рабочем столе масса ярлыков для быстрого доступа к часто используемым программам и файлам. Стоит мне

сменить разрешение, размеры рабочего стола изменяются, и ярлыки перестраиваются так, что уже ничего не найдешь. А когда я восстанавливаю прежнее разрешение, ярлыки так и остаются вперемешку. Чтобы навести порядок, приходится вручную перемещать каждый ярлык на свое место — очень интересное занятие!

В общем, мне это так осточертело, что я придумал утилиту, сохраняющую позиции элементов на экране (Desktop Item Position Saver, DIPS). DIPS состоит из крошечного исполняемого файла и компактной DLL. После запуска исполняемого файла появляется следующее окно.



В этом окне поясняется, как работать с утилитой. Когда она запускается с ключом *S* в командной строке, в реестре создается подраздел:

```
HKEY_CURRENT_USER\Software\Richter\Desktop Item Position Saver
```

куда добавляется по одному параметру на каждый ярлык, расположенный на Вашем рабочем столе. Значение каждого параметра — позиция соответствующего ярлыка. Утилиту DIPS следует запускать перед установкой более низкого экранного разрешения. Всласть наигравшись и восстановив нормальное разрешение, вновь запустите DIPS — на этот раз с ключом *R*. Тогда DIPS откроет соответствующий подраздел реестра и восстановит для каждого объекта рабочего стола его исходную позицию.

На первый взгляд утилита DIPS тривиальна и очень проста в реализации. Бродя только и надо, что получить описатель элемента управления ListView рабочего стола, заставить его (послав соответствующие сообщения) перечислить все ярлыки и определить их координаты, а потом сохранить полученные данные в реестре. Но попробуйте, и Вы убедитесь, что все не так просто. Проблема в том, что большинство оконных сообщений для стандартных элементов управления (например, LVM_GETITEM и LVM_GETITEMPOSITION) не может преодолеть границы процессов. Почему?

Сообщение LVM_GETITEM требует, чтобы Вы передали в параметре *lParam* адрес структуры LV_ITEM. Поскольку ее адрес имеет смысл лишь в адресном пространстве процесса — отправителя сообщения, процесс-приемник не может безопасно использовать его. Поэтому, чтобы DIPS работала так, как было обещано, в Explorer.exe надо внедрить код, посылающий сообщения LVM_GETITEM и LVM_GETITEMPOSITION элементу управления ListView рабочего стола.



В отличие от новых стандартных элементов управления встроенные (кнопки, поля, метки, списки, комбинированные списки и т. д.) позволяют передавать оконные сообщения через границы процессов. Например, окну списка, созданному каким-нибудь потоком другого процесса, можно послать сообщение LB_GETTEXT, чей параметр *lParam* указывает на строковый буфер в адресном пространстве процесса-отправителя. Это срабатывает, потому что операционная система специально проверяет, не отправлено ли сообщение LB_GETTEXT,

см. след. стр.

и, если да, сама создает проецируемый в память файл и копирует строковые данные из адресного пространства одного процесса в адресное пространство другого.

Почему Microsoft решила по-разному обрабатывать встроенные и новые элементы управления? Дело в том, что в 16-разрядной Windows, в которой все приложения выполняются в едином адресном пространстве, любая программа могла послать сообщение LB_GETTEXT окну, созданному другой программой. Чтобы упростить перенос таких приложений в Win32, Microsoft и пошла на эти ухищрения. А поскольку в 16-разрядной Windows нет новых элементов управления, то проблемы их переноса тоже нет, и Microsoft ничего подобного для них делать не стала.

Сразу после запуска DIPS получает описатель окна элемента управления ListView рабочего стола:

```
// окно ListView рабочего стола – "внук" окна ProgMan
hwndLV = GetFirstChild(GetFirstChild(FindWindow(_T("ProgMan"), NULL)));
```

Этот код сначала ищет окно класса ProgMan. Даже несмотря на то что никакой Program Manager не запускается, новая оболочка по-прежнему создает окно этого класса — для совместимости с приложениями, рассчитанными на старые версии Windows. У окна ProgMan единственное дочернее окно класса SHELLDLL_DefView, у которого тоже одно дочернее окно — класса SysListView32. Оно-то и служит элементом управления ListView рабочего стола. (Кстати, всю эту информацию я выудил благодаря Spy++)

Получив описатель окна ListView, я определяю идентификатор создавшего его потока, для чего вызываю *GetWindowThreadProcessId*. Этот идентификатор я передаю функции *SetDIPSHook*, реализованной в DIPSLib.cpp. Последняя функция устанавливает ловушку WH_GETMESSAGE для данного потока и вызывает:

```
PostThreadMessage(dwThreadId, WM_NULL, 0, 0);
```

чтобы разбудить поток Windows Explorer. Поскольку для него установлена ловушка WH_GETMESSAGE, операционная система автоматически внедряет мою DIPSLib.dll в адресное пространство Explorer и вызывает мою функцию *GetMsgProc*. Та сначала проверяет, впервые ли она вызвана, и, если да, создает скрытое окно с заголовком «Richter DIPS». Возьмите на заметку, что это окно создается потоком, принадлежащим Explorer. Пока окно создается, поток DIPS.exe возвращается из функции *SetDIPSHook* и вызывает:

```
GetMessage(&msg, NULL, 0, 0);
```

Этот вызов «усыпляет» поток до появления в очереди какого-нибудь сообщения. Хотя DIPS.exe сам не создает ни одного окна, у него все же есть очередь сообщений, и они помещаются туда исключительно в результате вызовов *PostThreadMessage*. Взглядите на код *GetMsgProc* в DIPSLib.cpp: сразу после обращения к *CreateDialog* стоит вызов *PostThreadMessage*, который вновь пробуждает поток DIPS.exe. Идентификатор потока сохраняется в разделяемой переменной внутри функции *SetDIPSHook*.

Очередь сообщений я использую для синхронизации потоков. В этом нет ничего противозаконного, и иногда гораздо проще синхронизировать потоки именно так, не прибегая к объектам ядра — мьютексам, семафорам, событиям и т. д. (В Windows очень богатый API; пользуйтесь этим.)

Когда поток DIPS.exe пробуждается, он узнает, что серверное диалоговое окно уже создано, и обращается к *FindWindow*, чтобы получить его описатель. С этого момента для организации взаимодействия между клиентом (утилитой DIPS) и сервером (скрытым диалоговым окном) можно использовать механизм оконных сообщений. Поскольку это диалоговое окно создано потоком, выполняемым в контексте процесса Explorer, нас мало что ограничивает в действиях с Explorer.

Чтобы сообщить своему диалоговому окну сохранить или восстановить позиции ярлыков на экране, достаточно послать сообщение:

```
// сообщаем окну DIPS, с каким окном ListView работать
// и что делать: сохранять или восстанавливать позиции ярлыков
SendMessage(hwndDIPS, WM_APP, (WPARAM) hwndLV, fSave);
```

Процедура диалогового окна проверяет сообщение WM_APP. Когда она принимает это сообщение, параметр *wParam* содержит описатель нужного элемента управления ListView, а *lParam* — булево значение, определяющее, сохранять текущие позиции ярлыков в реестре или восстанавливать.

Так как здесь используется *SendMessage*, а не *PostMessage*, управление не передается до завершения операции. Если хотите, определите дополнительные сообщения для процедуры диалогового окна — это расширит возможности программы в управлении Explorer. Закончив, я завершаю работу сервера, для чего посылаю ему сообщение WM_CLOSE, которое говорит диалоговому окну о необходимости самоуничтожения.

Наконец, перед своим завершением DIPS вновь вызывает *SetDIPSHook*, но на этот раз в качестве идентификатора потока передается 0. Получив нулевое значение, функция снимает ловушку WH_GETMESSAGE. А когда ловушка удаляется, операционная система автоматически выгружает DIPSLib.dll из адресного пространства процесса Explorer, и это означает, что теперь процедура диалогового окна больше не принадлежит данному адресному пространству. Поэтому важно уничтожить диалоговое окно заранее — до снятия ловушки. Иначе очередное сообщение, направленное диалоговому окну, вызовет нарушение доступа. И тогда Explorer будет аварийно завершен операционной системой — с внедрением DLL шутки плохи!

```


Dips.cpp

/*
Modуль: DIPS.cpp
Автор: Copyright (c) 2000, Джейфри Рихтер (Jeffrey Richter)
*/

#include "..\CmnHdr.h"      /* см. приложение A */
#include <WindowsX.h>
#include <tchar.h>
#include "Resource.h"
#include "..\22-DIPSLib\DIPLib.h"

///////////

BOOL Dlg_OnInitDialog(HWND hwnd, HWND hwndFocus, LPARAM lParam) {

```

Рис. 22-2. Утилита DIPS

см. след. стр.

Рис. 22-2. продолжение

```
chSETDLGICONS(hwnd, IDI_DIPS);
return(TRUE);
}

//////////////////////////////



void Dlg_OnCommand(HWND hwnd, int id, HWND hwndCtl, UINT codeNotify) {

    switch (id) {
        case IDC_SAVE:
        case IDC_RESTORE:
        case IDCANCEL:
            EndDialog(hwnd, id);
            break;
    }
}

//////////////////////////////



BOOL WINAPI Dlg_Proc(HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam) {

    switch (uMsg) {
        chHANDLE_DLGMMSG(hwnd, WM_INITDIALOG, Dlg_OnInitDialog);
        chHANDLE_DLGMMSG(hwnd, WM_COMMAND, Dlg_OnCommand);
    }
    return(FALSE);
}

//////////////////////////////



int WINAPI _tWinMain(HINSTANCE hinstExe, HINSTANCE, PTSTR pszCmdLine, int) {

    // преобразуем символ в командной строке в верхний регистр
    CharUpperBuff(pszCmdLine, 1);
    TCHAR cWhatToDo = pszCmdLine[0];

    if ((cWhatToDo != TEXT('S')) && (cWhatToDo != TEXT('R'))) {

        // неверный аргумент в командной строке; сообщаем пользователю
        cWhatToDo = 0;
    }

    if (cWhatToDo == 0) {
        // аргумента в командной строке нет;
        // выводим диалоговое окно с инструкциями
        switch (DialogBox(hinstExe, MAKEINTRESOURCE(IDD_DIPS), NULL, Dlg_Proc)) {
            case IDC_SAVE:
                cWhatToDo = TEXT('S');
                break;
        }
    }
}
```

Рис. 22-2. продолжение

```

        case IDC_RESTORE:
            cWhatToDo = TEXT('R');
            break;
    }
}

if (cWhatToDo == 0) {
    // пользователь не хочет ничего делать
    return(0);
}

// окно ListView рабочего стола - "внук" окна ProgMan
HWND hwndLV = GetFirstChild(GetFirstChild(
    FindWindow(TEXT("ProgMan"), NULL)));
chASSERT(IsWindow(hwndLV));

// Устанавливаем ловушку, внедряющую нашу DLL в адресное пространство
// Explorer. После этого создаем скрытое немодальное диалоговое окно.
// Мы посылаем ему сообщения, которые говорят, что делать.
chVERIFY(SetDIPSHook(GetWindowThreadProcessId(hwndLV, NULL)));

// ждем создания серверного окна DIPS
MSG msg;
GetMessage(&msg, NULL, 0, 0);

// получаем описатель скрытого диалогового окна
HWND hwndDIPS = FindWindow(NULL, TEXT("Richter DIPS"));

// проверяем, действительно ли это окно создано
ChASSERT(IsWindow(hwndDIPS));

// сообщаем окну DIPS, с каким окном ListView следует работать,
// а также что делать: сохранять или восстанавливать элементы
SendMessage(hwndDIPS, WM_APP, (WPARAM) hwndLV, (cWhatToDo == TEXT('S')));

// Сообщаем окну DIPS о необходимости самоуничтожения. Вместо
// PostMessage используем SendMessage, чтобы уничтожить окно
// до снятия ловушки.
SendMessage(hwndDIPS, WM_CLOSE, 0, 0);

// проверяем, действительно ли окно уничтожено
chASSERT(!IsWindow(hwndDIPS));

// убираем ловушку, удаляя процедуру диалогового окна DIPS
// из адресного пространства процесса Explorer
SetDIPSHook(0);

return(0);
}

/////////////////////////////// Конец файла ///////////////////////////////

```

см. след. стр.

Рис. 22-2. продолжение**DIPSLib.cpp**

```
*****  
Модуль: DIPSLib.cpp  
Автор: Copyright (c) 2000, Джейфри Рихтер (Jeffrey Richter)  
*****  
  
#include "..\CmnHdr.h"      /* см. приложение A */  
#include <WindowsX.h>  
#include <CommCtrl.h>  
  
#define DIPSLIBAPI __declspec(dllexport)  
#include "DIPSLib.h"  
#include "Resource.h"  
  
//////////  
  
#ifdef _DEBUG  
// эта функция активизирует отладчик  
void ForceDebugBreak() {  
    _try { DebugBreak(); }  
    _except(UnhandledExceptionFilter(GetExceptionInformation())) { }  
}  
#else  
#define ForceDebugBreak()  
#endif  
  
//////////  
  
// упреждающие ссылки  
HRESULT WINAPI GetMsgProc(int nCode, WPARAM wParam, LPARAM lParam);  
  
INT_PTR WINAPI Dlg_Proc(HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam);  
  
//////////  
  
// требуем от компилятора разместить переменную g_hhook в отдельном разделе  
// данных Shared, а компоновщику сообщаем, что данные из этого раздела должны  
// быть доступны всем экземплярам приложения  
#pragma data_seg("Shared")  
HHOOK g_hhook = NULL;  
DWORD g_dwThreadIdDIPS = 0;  
#pragma data_seg()  
  
// сообщаем компоновщику, что раздел Shared должен быть общим  
// и доступным для чтения и записи  
#pragma comment(linker, "/section:Shared,rws")  
  
//////////  
  
// неразделяемые переменные  
HINSTANCE g_hinstDll = NULL;
```

Рис. 22-2. продолжение

```
//////////  

BOOL WINAPI DllMain(HINSTANCE hinstDll, DWORD fdwReason, PVOID fImpLoad) {  

    switch (fdwReason) {  

        case DLL_PROCESS_ATTACH:  

            // к адресному пространству текущего процесса подключается DLL  

            g_hinstDll = hinstDll;  

            break;  

        case DLL_THREAD_ATTACH:  

            // в текущем процессе создается поток  

            break;  

        case DLL_THREAD_DETACH:  

            // поток корректно завершается  

            break;  

        case DLL_PROCESS_DETACH:  

            // вызывающий процесс отключает DLL от своего  

            // адресного пространства  

            break;  

    }  

    return(TRUE);  

}  

//////////  

BOOL WINAPI SetDIPSHook(DWORD dwThreadId) {  

    BOOL fOk = FALSE;  

    if (dwThreadId != 0) {  

        // убеждаемся, что ловушка еще не установлена  

        chASSERT(g_hhook == NULL);  

        // сохраняем идентификатор потока в разделяемой переменной, чтобы после создания  

        // окна сервера наша функция GetMsgProc могла послать асинхронный ответ потоку  

        g_dwThreadIdDIPS = GetCurrentThreadId();  

        // устанавливаем ловушку для указанного потока  

        g_hhook = SetWindowsHookEx(WH_GETMESSAGE, GetMsgProc, g_hinstDll,  

            dwThreadId);  

        fOk = (g_hhook != NULL);  

        if (fOk) {  

            // ловушка установлена; отправляем сообщение в очередь  

            // потока, провоцируя вызов функции ловушки  

            fOk = PostThreadMessage(dwThreadId, WM_NULL, 0, 0);  

        }  

    }  

}
```

см. след. стр.

Рис. 22-2. продолжение

```

} else {

    // убеждаемся, что ловушка действительно была установлена
    chASSERT(g_hhook != NULL);
    fOk = UnhookWindowsHookEx(g_hhook);
    g_hhook = NULL;
}
return(fOk);
}

///////////////////////////////



LRESULT WINAPI GetMsgProc(int nCode, WPARAM wParam, LPARAM lParam) {

static BOOL fFirstTime = TRUE;

if (fFirstTime) {
    // DLL только что внедрена
    fFirstTime = FALSE;

    // раскомментируйте следующую строку, чтобы вызвать отладчик
    // для процесса, в который только что внедрена DLL
    // ForceDebugBreak();

    // создаем окно сервера DIPS, обрабатывающее клиентские запросы
    CreateDialog(g_hinstDll, MAKEINTRESOURCE(IDD_DIPS), NULL, Dlg_Proc);

    // сообщаем утилите DIPS, что сервер готов к обработке запросов
    PostThreadMessage(g_dwThreadIdDIPS, WM_NULL, 0, 0);
}
return(CallNextHookEx(g_hhook, nCode, wParam, lParam));
}

///////////////////////////////



void Dlg_OnClose(HWND hwnd) {
    DestroyWindow(hwnd);
}

///////////////////////////////



static const TCHAR g_szRegSubKey[ ] =
    TEXT("Software\\Richter\\Desktop Item Position Saver");

///////////////////////////////



void SaveListViewItemPositions(HWND hwndLV) {

int nMaxItems = ListView_GetItemCount(hwndLV);

    // сохраняя новые позиции, удаляем из реестра информацию о старых
    LONG l = RegDeleteKey(HKEY_CURRENT_USER, g_szRegSubKey);
}

```

Рис. 22-2. продолжение

```

// создаем нужный раздел реестра
HKEY hkey;
l = RegCreateKeyEx(HKEY_CURRENT_USER, g_szRegSubKey, 0, NULL,
    REG_OPTION_NON_VOLATILE, KEY_SET_VALUE, NULL, &hkey, NULL);
chASSERT(l == ERROR_SUCCESS);

for (int nItem = 0; nItem < nMaxItems; nItem++) {

    // получаем имя и позицию элемента в ListView
    TCHAR szName[MAX_PATH];
    ListView_GetItemText(hwndLV, nItem, 0, szName, chDIMOF(szName));

    POINT pt;
    ListView_GetItemPosition(hwndLV, nItem, &pt);

    // сохраняем полученные данные в реестре
    l = RegSetValueEx(hkey, szName, 0, REG_BINARY, (PBYTE) &pt, sizeof(pt));
    chASSERT(l == ERROR_SUCCESS);
}
RegCloseKey(hkey);
}

///////////////////////////////
void RestoreListViewItemPositions(HWND hwndLV) {

    HKEY hkey;
    LONG l = RegOpenKeyEx(HKEY_CURRENT_USER, g_szRegSubKey,
        0, KEY_QUERY_VALUE, &hkey);
    if (l == ERROR_SUCCESS) {

        // если в ListView установлен режим автоупорядочения (Auto Arrange),
        // временно отключаем его
        DWORD dwStyle = GetWindowStyle(hwndLV);
        if (dwStyle & LVS_AUTOARRANGE)
            SetWindowLong(hwndLV, GWL_STYLE, dwStyle & ~LVS_AUTOARRANGE);

        l = NO_ERROR;
        for (int nIndex = 0; l != ERROR_NO_MORE_ITEMS; nIndex++) {
            TCHAR szName[MAX_PATH];
            DWORD cbValueName = chDIMOF(szName);

            POINT pt;
            DWORD cbData = sizeof(pt), nItem;

            // считываем данные из реестра
            DWORD dwType;
            l = RegEnumValue(hkey, nIndex, szName, &cbValueName,
                NULL, &dwType, (PBYTE) &pt, &cbData);

            if (l == ERROR_NO_MORE_ITEMS)

```

см. след. стр.

Рис. 22-2. продолжение

```

        continue;

    if ((dwType == REG_BINARY) && (cbData == sizeof(pt))) {
        // если данный параметр нам известен, пытаемся найти
        // одноименный элемент в ListView
        LV_FINDINFO lvfi;
        lvfi.flags = LVFI_STRING;
        lvfi.psz = szName;
        nItem = ListView_FindItem(hwndLV, -1, &lvfi);
        if (nItem != -1) {
            // элемент найден; изменяем его позицию
            ListView_SetItemPosition(hwndLV, nItem, pt.x, pt.y);
        }
    }
    // если раньше действовал режим автоупорядочения,
    // восстанавливаем его
    SetWindowLong(hwndLV, GWL_STYLE, dwStyle);
    RegCloseKey(hkey);
}

/////////////////////////////// Конец файла //////////////////////

INT_PTR WINAPI Dlg_Proc(HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam) {

    switch (uMsg) {
        chHANDLE_DLGMMSG(hwnd, WM_CLOSE, Dlg_OnClose);

        case WM_APP:
            // раскомментируйте следующую строку, чтобы вызвать отладчик
            // для процесса, в который только что внедрена DLL
            // ForceDebugBreak();

            if (lParam)
                SaveListViewItemPositions((HWND) wParam);
            else
                RestoreListViewItemPositions((HWND) wParam);
            break;
    }
    return(FALSE);
}

/////////////////////////////// Конец файла /////////////////////

```

DIPSLib.h

```

*****
* Модуль: DIPSLib.h
* Автор: Copyright (c) 2000, Джейфри Рихтер (Jeffrey Richter)
*****

```

Рис. 22-2. продолжение

```
#if !defined(DIPSLIBAPI)
#define DIPSLIBAPI __declspec(dllexport)
#endif

/////////////////////////////// Конец файла ///////////////////////////////
```

// прототипы внешних функций
DIPSLIBAPI BOOL WINAPI SetDIPSHook(DWORD dwThreadId);

Внедрение DLL с помощью удаленных потоков

Третий способ внедрения DLL — самый гибкий. В нем используются многие особенности Windows: процессы, потоки, синхронизация потоков, управление виртуальной памятью, поддержка DLL и Unicode. (Если Вы плаваете в каких-то из этих тем, прочтите сначала соответствующие главы книги.) Большинство Windows-функций позволяет процессу управлять лишь самим собой, исключая тем самым риск повреждения одного процесса другим. Однако есть и такие функции, которые дают возможность управлять чужим процессом. Изначально многие из них были рассчитаны на применение в отладчиках и других инструментальных средствах. Но ничто не мешает использовать их и в обычном приложении.

Внедрение DLL этим способом предполагает вызов функции *LoadLibrary* потоком целевого процесса для загрузки нужной DLL. Так как управление потоками чужого процесса сильно затруднено, Вы должны создать в нем свой поток. К счастью, Windows-функция *CreateRemoteThread* делает эту задачу несложной:

```
HANDLE CreateRemoteThread(
    HANDLE hProcess,
    PSECURITY_ATTRIBUTES psa,
    DWORD dwStackSize,
    PTHREAD_START_ROUTINE pfnStartAddr,
    PVOID pvParam,
    DWORD fdwCreate,
    PDWORD pdwThreadId);
```

Она идентична *CreateThread*, но имеет дополнительный параметр *hProcess*, идентифицирующий процесс, которому будет принадлежать новый поток. Параметр *pfnStartAddr* определяет адрес функции потока. Этот адрес, разумеется, относится к удаленному процессу — функция потока не может находиться в адресном пространстве Вашего процесса.



В Windows 2000 чаще используемая функция *CreateThread*, между прочим, реализована через вызов *CreateRemoteThread*:

```
HANDLE CreateThread(PSECURITY_ATTRIBUTES psa, DWORD dwStackSize,
    PTHREAD_START_ROUTINE pfnStartAddr, PVOID pvParam,
    DWORD fdwCreate, PDWORD pdwThreadID) {

    return (CreateRemoteThread(GetCurrentProcess(), psa, dwStackSize,
        pfnStartAddr, pvParam, fdwCreate, pdwThreadID));
}
```

WINDOWS 98 В Windows 98 функция *CreateRemoteThread* определена, но не реализована и просто возвращает FALSE; последующий вызов *GetLastError* дает код ERROR_CALL_NOT_IMPLEMENTED. (Но функция *CreateThread*, которая создает поток в вызывающем процессе, реализована полностью.) Так что описываемый здесь метод внедрения DLL в Windows 98 не работает.

О'кэй, теперь Вы знаете, как создать поток в другом процессе. Но как заставить этот поток загрузить нашу DLL? Ответ прост: нужно, чтобы он вызвал функцию *LoadLibrary*:

```
HINSTANCE LoadLibrary(PCTSTR pszLibFile);
```

Заглянув в заголовочный файл WinBase.h, Вы увидите, что для *LoadLibrary* там есть такие строки:

```
HINSTANCE WINAPI LoadLibraryA(LPCSTR pszLibFileName);
HINSTANCE WINAPI LoadLibraryW(LPCWSTR pszLibFileName);
#ifndef UNICODE
#define LoadLibrary LoadLibraryW
#else
#define LoadLibrary LoadLibraryA
#endif // !UNICODE
```

В действительности существует две функции *LoadLibrary*: *LoadLibraryA* и *LoadLibraryW*. Они различаются только типом передаваемого параметра. Если имя файла библиотеки хранится как ANSI-строка, вызывайте *LoadLibraryA*; если же имя файла представлено Unicode-строкой — *LoadLibraryW*. Самой функции *LoadLibrary* нет. В большинстве программ макрос *LoadLibrary* раскрывается в *LoadLibraryA*.

К счастью, прототипы *LoadLibrary* и функции потока идентичны. Вот как выглядит прототип функции потока:

```
DWORD WINAPI ThreadFunc(PVOID pvParam);
```

О'кэй, не идентичны, но очень похожи друг на друга. Обе функции принимают единственный параметр и возвращают некое значение. Кроме того, обе используют одни и те же правила вызова — WINAPI. Это крайне удачное стечание обстоятельств, потому что нам как раз и нужно создать новый поток, адрес функции которого является адресом *LoadLibraryA* или *LoadLibraryW*. По сути, требуется выполнить примерно такую строку кода:

```
HANDLE hThread = CreateRemoteThread(hProcessRemote, NULL, 0,
    LoadLibraryA, "C:\\MyLib.dll", 0, NULL);
```

Или, если Вы предпочитаете Unicode:

```
HANDLE hThread = CreateRemoteThread(hProcessRemote, NULL, 0,
    LoadLibraryW, L"C:\\\\MyLib.dll", 0, NULL);
```

Новый поток в удаленном процессе немедленно вызывает *LoadLibraryA* (или *LoadLibraryW*), передавая ей адрес полного имени DLL. Все просто. Однако Вас ждут две проблемы.

Первая в том, что нельзя вот так запросто, как я показал выше, передать *LoadLibraryA* или *LoadLibraryW* в четвертом параметре функции *CreateRemoteThread*. Причина этого весьма неочевидна. При сборке программы в конечный двоичный файл помещается раздел импорта (описанный в главе 19). Этот раздел состоит из серии шлю-

зов к импортируемым функциям. Так что, когда Ваш код вызывает функцию вроде *LoadLibraryA*, в разделе импорта модуля генерируется вызов соответствующего шлюза. А уже от шлюза происходит переход к реальной функции.

Следовательно, прямая ссылка на *LoadLibraryA* в вызове *CreateRemoteThread* преобразуется в обращение к шлюзу *LoadLibraryA* в разделе импорта Вашего модуля. Передача адреса шлюза в качестве стартового адреса удаленного потока заставит этот поток выполнить неизвестно что. И скорее всего это закончится нарушением доступа. Чтобы напрямую вызывать *LoadLibraryA*, минуя шлюз, Вы должны выяснить ее точный адрес в памяти с помощью *GetProcAddress*.

Вызов *CreateRemoteThread* предполагает, что Kernel32.dll спроектирована в локальном процессе на ту же область памяти, что и в удаленном. Kernel32.dll используется всеми приложениями, и, как показывает опыт, система проектирует эту DLL в каждом процессе по одному и тому же адресу. Так что *CreateRemoteThread* надо вызвать так:

```
// получаем истинный адрес LoadLibraryA в Kernel32.dll
PTHREAD_START_ROUTINE pfnThreadRtn = (PTHREAD_START_ROUTINE)
    GetProcAddress(GetModuleHandle(TEXT("Kernel32")), "LoadLibraryA");
```

```
HANDLE hThread = CreateRemoteThread(hProcessRemote, NULL, 0,
    pfnThreadRtn, "C:\\MyLib.dll", 0, NULL);
```

Или, если Вы предпочитаете Unicode:

```
// получаем истинный адрес LoadLibraryA в Kernel32.dll
PTHREAD_START_ROUTINE pfnThreadRtn = (PTHREAD_START_ROUTINE)
    GetProcAddress(GetModuleHandle(TEXT("Kernel32")), "LoadLibraryW");

HANDLE hThread = CreateRemoteThread(hProcessRemote, NULL, 0,
    pfnThreadRtn, L"C:\\MyLib.dll", 0, NULL);
```

Отлично, одну проблему мы решили. Но я говорил, что их две. Вторая связана со строкой, в которой содержится полное имя файла DLL. Стока «C:\\MyLib.dll» находится в адресном пространстве вызывающего процесса. Ее адрес передается только что созданному потоку, который в свою очередь передает его в *LoadLibraryA*. Но, когда *LoadLibraryA* будет проводить разыменование (dereferencing) этого адреса, она не найдет по нему строку с полным именем файла DLL и скорее всего вызовет нарушение доступа в потоке удаленного процесса; пользователь увидит сообщение о необрабатываемом исключении, и удаленный процесс будет закрыт. Все верно: Вы благополучно угробили чужой процесс, сохранив свой в целости и сохранности!

Эта проблема решается размещением строки с полным именем файла DLL в адресном пространстве удаленного процесса. Впоследствии, вызывая *CreateRemoteThread*, мы передадим ее адрес (в удаленном процессе). На этот случай в Windows предусмотрена функция *VirtualAllocEx*, которая позволяет процессу выделять память в чужом адресном пространстве:

```
VOID VirtualAllocEx(
    HANDLE hProcess,
    PVOID pvAddress,
    SIZE_T dwSize,
    DWORD  flAllocationType,
    DWORD  flProtect);
```

А освободить эту память можно с помощью функции *VirtualFreeEx*.

```
BOOL VirtualFreeEx(  
    HANDLE hProcess,  
    PVOID pvAddress,  
    SIZE_T dwSize,  
    DWORD dwFreeType);
```

Обе функции аналогичны своим версиям без суффикса *Ex* в конце (о них я рассказывал в главе 15). Единственная разница между ними в том, что эти две функции требуют передачи в первом параметре описателя удаленного процесса.

Выделив память, мы должны каким-то образом скопировать строку из локального адресного пространства в удаленное. Для этого в Windows есть две функции:

```
BOOL ReadProcessMemory(  
    HANDLE hProcess,  
    PVOID pvAddressRemote,  
    PVOID pvBufferLocal,  
    DWORD dwSize,  
    PDWORD pdwNumBytesRead);
```

```
BOOL WriteProcessMemory(  
    HANDLE hProcess,  
    PVOID pvAddressRemote,  
    PVOID pvBufferLocal,  
    DWORD dwSize,  
    PDWORD pdwNumBytesWritten);
```

Параметр *hProcess* идентифицирует удаленный процесс, *pvAddressRemote* и *pvBufferLocal* определяют адреса в адресных пространствах удаленного и локального процесса, а *dwSize* — число передаваемых байтов. По адресу, на который указывает параметр *pdwNumBytesRead* или *pdwNumBytesWritten*, возвращается число фактически считанных или записанных байтов.

Теперь, когда Вы понимаете, что я пытаюсь сделать, давайте суммируем все сказанное и запишем это в виде последовательности операций, которые Вам надо будет выполнить.

1. Выделите блок памяти в адресном пространстве удаленного процесса через *VirtualAllocEx*.
2. Вызвав *WriteProcessMemory*, скопируйте строку с полным именем файла DLL в блок памяти, выделенный в п. 1.
3. Используя *GetProcAddress*, получите истинный адрес функции *LoadLibraryA* или *LoadLibraryW* внутри Kernel32.dll.
4. Вызвав *CreateRemoteThread*, создайте поток в удаленном процессе, который вызовет соответствующую функцию *LoadLibrary*, передав ей адрес блока памяти, выделенного в п. 1.

На этом этапе DLL внедрена в удаленный процесс, а ее функция *DllMain* получила уведомление *DLL_PROCESS_ATTACH* и может приступить к выполнению нужного кода. Когда *DllMain* вернет управление, удаленный поток выйдет из *LoadLibrary* и вернется в функцию *BaseThreadStart* (см. главу 6), которая в свою очередь вызовет *ExitThread* и завершит этот поток.

Теперь в удаленном процессе имеется блок памяти, выделенный в п. 1, и DLL, все еще «сидящая» в его адресном пространстве. Для очистки после завершения удаленного потока потребуется несколько дополнительных операций.

5. Вызовом *VirtualFreeEx* освободите блок памяти, выделенный в п. 1.
6. С помощью *GetProcAddress* определите истинный адрес функции *FreeLibrary* внутри Kernel32.dll.
7. Используя *CreateRemoteThread*, создайте в удаленном процессе поток, который вызовет *FreeLibrary* с передачей HINSTANCE внедренной DLL.

Вот, собственно, и все. Единственный недостаток этого метода внедрения DLL (самого универсального из уже рассмотренных) — многие нужные функции в Windows 98 не поддерживаются. Так что данный метод применим только в Windows 2000.

Программа-пример InjLib

Эта программа, «22 InjLib.exe» (см. листинг на рис. 22-3), внедряет DLL с помощью функции *CreateRemoteThread*. Файлы исходного кода и ресурсов этой программы и DLL находятся в каталогах 22-InjLib и 22-ImgWalk на компакт-диске, прилагаемом к книге. После запуска InjLib на экране появляется диалоговое окно для ввода идентификатора выполняемого процесса, в который будет внедрена DLL.



Вы можете выяснить этот идентификатор через Task Manager. Получив его, программа попытается открыть описатель этого процесса, вызвав *OpenProcess* и запросив соответствующие права доступа.

```
hProcess = OpenProcess(
    PROCESS_CREATE_THREAD | // для CreateRemoteThread
    PROCESS_VM_OPERATION | // для VirtualAllocEx/VirtualFreeEx
    PROCESS_VM_WRITE,      // для WriteProcessMemory
    FALSE, dwProcessId);
```

Если *OpenProcess* вернет NULL, значит, программа выполняется в контексте защищины, в котором открытие описателя этого процесса не разрешено. Некоторые процессы вроде WinLogon, SvcHost и Csrss выполняются по локальной системной учетной записи, которую зарегистрированный пользователь не имеет права изменять. Описатель такого процесса можно открыть, только если Вы получили полномочия на отладку этих процессов. Программа ProcessInfo из главы 4 демонстрирует, как это делается.

При успешном выполнении *OpenProcess* записывает в буфер полное имя внедряемой DLL. Далее программа вызывает *InjectLib* и передает ей описатель удаленного процесса. И, наконец, после возврата из *InjectLib* программа выводит окно, где сообщает, успешно ли внедрена DLL, а потом закрывает описатель процесса.

Наверное, Вы заметили, что я специально проверяю, не равен ли идентификатор процесса нулю. Если да, то вместо идентификатора удаленного процесса я передаю идентификатор процесса самой InjLib.exe, получаемый вызовом *GetCurrentProcessId*. Тогда при вызове *InjectLib* библиотека внедряется в адресное пространство процесса InjLib. Я сделал это для упрощения отладки. Сами понимаете, при возникновении ошибки иногда трудно определить, в каком процессе она находится: локальном или удаленном. Поначалу я отлаживал код с помощью двух отладчиков: один наблюдал за InjLib, другой — за удаленным процессом. Это оказалось страшно неудобно. Потом меня осенило, что InjLib способна внедрить DLL и в себя, т. е. в адресное пространство вызывающего процесса. И это сразу упростило отладку.

Просмотрев начало исходного кода модуля, Вы увидите, что *InjectLib* — на самом деле макрос, заменяемый на *InjectLibA* или *InjectLibW* в зависимости от того, как компилируется исходный код. В исходном коде достаточно комментариев, и я добавлю лишь одно. Функция *InjectLibA* весьма компактна. Она просто преобразует полное имя DLL из ANSI в Unicode и вызывает *InjectLibW*, которая и делает всю работу. Тут я придерживаюсь того подхода, который я рекомендовал в главе 2.



InjLib.cpp

```
*****  
Модуль: InjLib.cpp  
Автор: Copyright (c) 2000, Джейфри Рихтер (Jeffrey Richter)  
*****  
  
#include "..\CmnHdr.h"      /* см. приложение A */  
#include <windowsx.h>  
#include <stdio.h>  
#include <tchar.h>  
#include <malloc.h>          // для доступа к alloca  
#include <TlHelp32.h>  
#include "Resource.h"  
  
//////////  
  
#ifdef UNICODE  
#define InjectLib InjectLibW  
#define EjectLib  EjectLibW  
#else  
#define InjectLib InjectLibA  
#define EjectLib  EjectLibA  
#endif // !UNICODE  
  
//////////  
  
BOOL WINAPI InjectLibW(DWORD dwProcessId, PCWSTR pszLibFile) {  
  
    BOOL fOk = FALSE; // считаем, что функция потерпит неудачу  
    HANDLE hProcess = NULL, hThread = NULL;  
    PWSTR pszLibFileRemote = NULL;  
  
    __try {  
        // получаем описатель целевого процесса  
        hProcess = OpenProcess(  
            PROCESS_CREATE_THREAD | // для CreateRemoteThread  
            PROCESS_VM_OPERATION | // для VirtualAllocEx/VirtualFreeEx  
            PROCESS_VM_WRITE,       // для WriteProcessMemory  
            FALSE, dwProcessId);  
        if (hProcess == NULL) __leave;  
    }
```

Рис. 22-3. Программа-пример *InjLib*

Рис. 22-3. продолжение

```

// определяем, сколько байтов нужно для строки с полным именем DLL
int cch = 1 + lstrlenW(pszLibFile);
int cb = cch * sizeof(WCHAR);

// выделяем блок памяти под эту строку
pszLibFileRemote = (PWSTR)
    VirtualAllocEx(hProcess, NULL, cb, MEM_COMMIT, PAGE_READWRITE);
if (pszLibFileRemote == NULL) __leave;

// копируем эту строку в адресное пространство удаленного процесса
if (!WriteProcessMemory(hProcess, pszLibFileRemote,
    (PVOID) pszLibFile, cb, NULL)) __leave;

// получаем истинный адрес LoadLibraryW в Kernel32.dll
PTHREAD_START_ROUTINE pfnThreadRtn = (PTHREAD_START_ROUTINE)
    GetProcAddress(GetModuleHandle(TEXT("Kernel32")), "LoadLibraryW");
if (pfnThreadRtn == NULL) __leave;

// создаем удаленный поток, вызывающий LoadLibraryW
hThread = CreateRemoteThread(hProcess, NULL, 0,
    pfnThreadRtn, pszLibFileRemote, 0, NULL);
if (hThread == NULL) __leave;

// ждем завершения удаленного потока
WaitForSingleObject(hThread, INFINITE);

fOk = TRUE; // все прошло успешно
}

__finally { // проводим очистку

// освобождаем память, выделенную под строку для полного имени DLL
if (pszLibFileRemote != NULL)
    VirtualFreeEx(hProcess, pszLibFileRemote, 0, MEM_RELEASE);

if (hThread != NULL)
    CloseHandle(hThread);

if (hProcess != NULL)
    CloseHandle(hProcess);
}

return(fOk);
}

///////////////////////////////
BOOL WINAPI InjectLibA(DWORD dwProcessId, PCSTR pszLibFile) {

// выделяем буфер (в стеке) для Unicode-строки с полным именем DLL
PWSTR pszLibFileW = (PWSTR)
    _alloca((lstrlenA(pszLibFile) + 1) * sizeof(WCHAR));
}

```

см. след. стр.

Рис. 22-3. продолжение

```
// преобразуем ANSI-строку в Unicode
wsprintfW(pszLibFileW, L"%S", pszLibFile);

// вызываем Unicode-версию функции, которая, собственно, и выполняет работу
return(InjectLibW(dwProcessId, pszLibFileW));
}

///////////////////////////////
BOOL WINAPI EjectLibW(DWORD dwProcessId, PCWSTR pszLibFile) {

    BOOL fOk = FALSE; // считаем, что функция потерпит неудачу
    HANDLE hthSnapshot = NULL;
    HANDLE hProcess = NULL, hThread = NULL;

    __try {
        // получаем новый "моментальный снимок" процесса
        hthSnapshot = CreateToolhelp32Snapshot(TH32CS_SNAPMODULE, dwProcessId);
        if (hthSnapshot == NULL) __leave;

        // получаем HMODULE требуемой DLL
        MODULEENTRY32W me = { sizeof(me) };
        BOOL fFound = FALSE;
        BOOL fMoreMods = Module32FirstW(hthSnapshot, &me);
        for (; fMoreMods; fMoreMods = Module32NextW(hthSnapshot, &me)) {
            fFound = (lstrcmpiW(me.szModule, pszLibFile) == 0) ||
                (lstrcmpiW(me.szExePath, pszLibFile) == 0);
            if (fFound) break;
        }
        if (!fFound) __leave;

        // получаем описатель целевого процесса
        hProcess = OpenProcess(
            PROCESS_CREATE_THREAD | 
            PROCESS_VM_OPERATION, // для CreateRemoteThread
            FALSE, dwProcessId);
        if (hProcess == NULL) __leave;

        // получаем истинный адрес LoadLibraryW в Kernel32.dll
        PTHREAD_START_ROUTINE pfnThreadRtn = (PTHREAD_START_ROUTINE)
            GetProcAddress(GetModuleHandle(TEXT("Kernel32")), "FreeLibrary");
        if (pfnThreadRtn == NULL) __leave;

        // создаем удаленный поток,зывающий LoadLibraryW
        hThread = CreateRemoteThread(hProcess, NULL, 0,
            pfnThreadRtn, me.modBaseAddr, 0, NULL);
        if (hThread == NULL) __leave;

        // ждем завершения удаленного потока
        WaitForSingleObject(hThread, INFINITE);
```

Рис. 22-3. продолжение

```

fOk = TRUE; // все прошло успешно
}
__finally { // проводим очистку

    if (hthSnapshot != NULL)
        CloseHandle(hthSnapshot);

    if (hThread     != NULL)
        CloseHandle(hThread);

    if (hProcess    != NULL)
        CloseHandle(hProcess);
}
return(fOk);
}

///////////////////////////////



BOOL WINAPI EjectLibA(DWORD dwProcessId, PCSTR pszLibFile) {

    // выделяем буфер (в стеке) для Unicode-строки с полным именем DLL
    PWSTR pszLibFileW = (PWSTR)
        _alloca((lstrlenA(pszLibFile) + 1) * sizeof(WCHAR));

    // преобразуем ANSI-строку в Unicode
    wsprintfW(pszLibFileW, L"%S", pszLibFile);

    // вызываем Unicode-версию функции, которая, собственно, и выполняет работу
    return(EjectLibW(dwProcessId, pszLibFileW));
}

///////////////////////////////



BOOL Dlg_OnInitDialog(HWND hwnd, HWND hwndFocus, LPARAM lParam) {

    chSETDLGICONS(hwnd, IDI_INJLIB);
    return(TRUE);
}

///////////////////////////////



void Dlg_OnCommand(HWND hwnd, int id, HWND hwndCtl, UINT codeNotify) {

    switch (id) {
        case IDCANCEL:
            EndDialog(hwnd, id);
            break;

        case IDC_INJECT:
            DWORD dwProcessId = GetDlgItemInt(hwnd, IDC_PROCESSID, NULL, FALSE);
}

```

см. след. стр.

Рис. 22-3. продолжение

```
if (dwProcessId == 0) {
    // если идентификатор процесса равен 0, DLL
    // внедряется в локальный процесс (это облегчает отладку)
    DwProcessId = GetCurrentProcessId();
}

TCHAR szLibFile[MAX_PATH];
GetModuleFileName(NULL, szLibFile, sizeof(szLibFile));
_tcscpy(_tcsrchr(szLibFile, TEXT('\\')) + 1, TEXT("22_ImgWalk.DLL"));
if (InjectLib(dwProcessId, szLibFile)) {
    chVERIFY(EjectLib(dwProcessId, szLibFile));
    chMB("DLL Injection/Ejection successful.");
} else {
    chMB("DLL Injection/Ejection failed.");
}
break;
}

///////////////////////////////
INT_PTR WINAPI Dlg_Proc(HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam) {

switch (uMsg) {
    chHANDLE_DLGMMSG(hwnd, WM_INITDIALOG, Dlg_OnInitDialog);
    chHANDLE_DLGMMSG(hwnd, WM_COMMAND, Dlg_OnCommand);
}
return(FALSE);
}

/////////////////////////////
int WINAPI _tWinMain(HINSTANCE hinstExe, HINSTANCE, PTSTR pszCmdLine, int) {

chWindows9xNotAllowed();
DialogBox(hinstExe, MAKEINTRESOURCE(IDD_INJLIB), NULL, Dlg_Proc);
return(0);
}

/////////////////////////// Конец файла ////////////////////////
```

Библиотека ImgWalk.dll

ImgWalk.dll (см. листинг на рис. 22-4) — это DLL, которая, будучи внедрена в адресное пространство процесса, выдаст список всех DLL, используемых этим процессом. Файлы исходного кода и ресурсов этой DLL находятся в каталоге 22-ImgWalk на компакт-диске, прилагаемом к книге. Если, например, сначала запустить Notepad, а потом InjLib, передав ей идентификатор процесса Notepad, то InjLib внедрит ImgWalk.dll в адресное пространство Notepad. Попав туда, ImgWalk определит, образы каких файлов (EXE и DLL) используются процессом Notepad, и покажет результаты в следующем окне.



Модуль ImgWalk сканирует адресное пространство процесса и ищет спроектированные файлы, вызывая в цикле функцию *VirtualQuery*, которая заполняет структуру *MEMORY_BASIC_INFORMATION*. На каждой итерации цикла ImgWalk проверяет, нет ли строки с полным именем файла, которую можно было бы добавить в список, выводимый на экран.

```
char szBuf[MAX_PATH * 100] = { 0 };

PBYTE pb = NULL;
MEMORY_BASIC_INFORMATION mbi;
while (VirtualQuery(pb, &mbi, sizeof(mbi)) == sizeof(mbi)) {

    int nLen;
    char szModName[MAX_PATH];

    if (mbi.State == MEM_FREE)
        mbi.AllocationBase = mbi.BaseAddress;

    if ((mbi.AllocationBase == hinstDll) ||
        (mbi.AllocationBase != mbi.BaseAddress) ||
        (mbi.AllocationBase == NULL)) {
        // Имя модуля не включается в список, если
        // истинно хотя бы одно из следующих условий:
        // 1. Данный регион содержит нашу DLL.
        // 2. Данный блок НЕ является началом региона.
        // 3. Адрес равен NULL.
        nLen = 0;
    } else {
        nLen = GetModuleFileNameA((HINSTANCE) mbi.AllocationBase,
                                  szModName, chDIMOF(szModName));
    }

    if (nLen > 0) {
        wsprintfA(strchr(szBuf, 0), "\n%08X-%s",
                  mbi.AllocationBase, szModName);
    }
    pb += mbi.RegionSize;
}
chMB(&szBuf[1]);
```

Сначала я проверяю, не совпадает ли базовый адрес региона с базовым адресом внедренной DLL. Если да, я обнуляю *nLen*, чтобы не показывать в окне имя внедренной DLL. Нет — пытаюсь получить имя модуля, загруженного по базовому адресу данного региона. Если значение *nLen* больше 0, система распознает, что указанный адрес идентифицирует загруженный модуль, и помещает в буфер *szModName* полное имя (вместе с путем) этого модуля. Затем я присоединяю HINSTANCE данного модуля (базовый адрес) и его полное имя к строке *szBuf*, которая в конечном счете и появится в окне. Когда цикл заканчивается, DLL открывает на экране окно со списком.

ImgWalk.cpp

```
*****  
Модуль: ImgWalk.cpp  
Автор: Copyright (c) 2000, Джейфри Рихтер (Jeffrey Richter)  
*****  
  
#include "..\CmnHdr.h"      /* см. приложение A */  
#include <tchar.h>  
  
//////////  
  
BOOL WINAPI DllMain(HINSTANCE hinstDll, DWORD fdwReason, PVOID fImpLoad) {  
  
    if (fdwReason == DLL_PROCESS_ATTACH) {  
        char szBuf[MAX_PATH * 100] = { 0 };  
  
        PBYTE pb = NULL;  
        MEMORY_BASIC_INFORMATION mbi;  
        while (VirtualQuery(pb, &mbi, sizeof(mbi)) == sizeof(mbi)) {  
  
            int nLen;  
            char szModName[MAX_PATH];  
  
            if (mbi.State == MEM_FREE)  
                mbi.AllocationBase = mbi.BaseAddress;  
  
            if ((mbi.AllocationBase == hinstDll) ||  
                (mbi.AllocationBase != mbi.BaseAddress) ||  
                (mbi.AllocationBase == NULL)) {  
                // Имя модуля не включается в список, если  
                // истинно хотя бы одно из следующих условий:  
                // 1. Данный регион содержит нашу DLL.  
                // 2. Данный блок НЕ является началом региона.  
                // 3. Адрес равен NULL.  
                nLen = 0;  
            } else {  
                nLen = GetModuleFileNameA((HINSTANCE) mbi.AllocationBase,  
                    szModName, chDIMOF(szModName));  
            }  
  
            if (nLen > 0) {  
                wsprintfA(strchr(szBuf, 0), "\n%p-%s",
```

Рис. 22-4. Исходный код *ImgWalk.dll*

Рис. 22-4. продолжение

```

        mbi.AllocationBase, szModName);
    }

    pb += mbi.RegionSize;
}
chMB(&szBuf[1]);
}

return(TRUE);
}

/////////////////////////////// Конец файла ///////////////////////////////

```

Внедрение троянской DLL

Другой способ внедрения состоит в замене DLL, загружаемой процессом, на другую DLL. Например, зная, что процессу нужна Xyz.dll, Вы можете создать свою DLL и присвоить ей то же имя. Конечно, перед этим Вы должны переименовать исходную Xyz.dll.

В своей Xyz.dll Вам придется экспортировать те же идентификаторы, что и в исходной Xyz.dll. Это несложно, если задействовать механизм переадресации функций (см. главу 20); однако его лучше не применять, иначе Вы окажетесь в зависимости от конкретной версии DLL. Если Вы замените, скажем, системную DLL, а Microsoft потом добавит в нее новые функции, в Вашей версии той же DLL их не будет. А значит, не удастся загрузить приложения, использующие эти новые функции.

Если Вы хотите применить этот метод только для одного приложения, то можете присвоить своей DLL уникальное имя и записать его в раздел импорта исполняемого модуля приложения. Дело в том, что раздел импорта содержит имена всех DLL, нужных EXE-модулю. Вы можете «покопаться» в этом разделе и изменить его так, чтобы загрузчик операционной системы загружал Вашу DLL. Этот прием совсем неплох, но требует глубоких знаний о формате EXE- и DLL-файлов.

Внедрение DLL как отладчика

Отладчик может выполнять особые операции над отлаживаемым процессом. Когда отлаживаемый процесс загружен и его адресное пространство создано, но первичный поток еще не выполняется, система автоматически уведомляет об этом отладчик. В этот момент отладчик может внедрить в него нужный код (используя, например, *WriteProcessMemory*), а затем заставить его первичный поток выполнить внедренный код.

Этот метод требует манипуляций со структурой CONTEXT потока отлаживаемого процесса, а значит, Ваш код будет зависеть от типа процессора, и его придется модифицировать при переносе на другую процессорную платформу. Кроме того, Вам почти наверняка придется вручную корректировать машинный код, который должен быть выполнен отлаживаемым процессом. Не забудьте и о жесткой связи между отладчиком и отлаживаемой программой: как только отладчик закрывается, Windows немедленно закрывает и отлаживаемую программу. Избежать этого нельзя.

Внедрение кода в среде Windows 98 через проецируемый в память файл

Эта задача в Windows 98, по сути, тривиальна. В ней все 32-разрядные приложения делят верхние два гигабайта своих адресных пространств. Выделенный там блок памяти доступен любому приложению. С этой целью Вы должны использовать проецируемые в память файлы (см. главу 17). Сначала Вы создаете проекцию файла, а потом вызываете *MapViewOfFile* и делаете ее видимой. Далее Вы записываете нужную информацию в эту область своего адресного пространства (она одинакова во всех адресных пространствах). Чтобы все это работало, Вам, вероятно, придется вручную писать машинные коды, а это затруднит перенос программы на другую процессорную платформу. Но вряд ли это должно Вас волновать — все равно Windows 98 работает только на процессорах типа x86.

Данный метод тоже довольно труден, потому что Вам нужно будет заставить поток другого процесса выполнять код в проекции файла. Для этого понадобятся какие-то средства управления удаленным потоком. Здесь пригодилась бы функция *CreateRemoteThread*, но Windows 98 ее не поддерживает. Увы, готового решения этой проблемы у меня нет.

Внедрение кода через функцию *CreateProcess*

Если Ваш процесс порождает дочерний, в который надо внедрить какой-то код, то задача значительно упрощается. Родительский процесс может создать новый процесс и сразу же приостановить его. Это позволит изменить состояние дочернего процесса до начала его выполнения. В то же время родительский процесс получает описатель первичного потока дочернего процесса. Зная его, Вы можете модифицировать код, который будет выполняться этим потоком. Тем самым Вы решите проблему, упомянутую в предыдущем разделе: в данном случае нетрудно установить регистр указателя команд, принадлежащий потоку, на код в проекции файла.

Вот один из способов контроля за тем, какой код выполняется первичным потоком дочернего процесса:

1. Создайте дочерний процесс в приостановленном состоянии.
2. Получите стартовый адрес его первичного потока, считав его из заголовка исполняемого модуля.
3. Сохраните где-нибудь машинные команды, находящиеся по этому адресу памяти.
4. Введите на их место свои команды. Этот код должен вызывать *LoadLibrary* для загрузки DLL.
5. Разрешите выполнение первичного потока дочернего процесса.
6. Восстановите ранее сохраненные команды по стартовому адресу первичного потока.
7. Пусть процесс продолжает выполнение со стартового адреса так, будто ничего и не было.

Этапы 6 и 7 довольно трудны, но реализовать их можно — такое уже делалось.

У этого метода масса преимуществ. Во-первых, мы получаем адресное пространство до выполнения приложения. Во-вторых, данный метод применим как в Windows 98, так и в Windows 2000. В третьих, мы можем без проблем отлаживать приложение с внед-

ренной DLL, не пользуясь отладчиком. Наконец, он работает как в консольных, так и в GUI-приложениях.

Однако у него есть и недостатки. Внедрение DLL возможно, только если это делается из родительского процесса. И, конечно, этот метод создает зависимость программы от конкретного процессора; при ее переносе на другую процессорную платформу потребуются определенные изменения в коде.

Перехват API-вызовов: пример

Внедрение DLL в адресное пространство процесса — замечательный способ узнать, что происходит в этом процессе. Однако простое внедрение DLL не дает достаточной информации. Зачастую надо точно знать, как потоки определенного процесса вызывают различные функции, а иногда и изменять поведение той или иной Windows-функции.

Мне известна одна компания, которая выпустила DLL для своего приложения, работающего с базой данных. Эта DLL должна была расширить возможности основного продукта. При закрытии приложения DLL получала уведомление `DLL_PROCESS_DETACH` и только после этого проводила очистку ресурсов. При этом DLL должна была вызывать функции из других DLL для закрытия сокетов, файлов и других ресурсов, но к тому моменту другие DLL тоже получали уведомление `DLL_PROCESS_DETACH`, так что корректно завершить работу никак не удавалось.

Для решения этой проблемы компания наняла меня, и я предложил поставить ловушку на функцию `ExitProcess`. Как Вам известно, вызов `ExitProcess` заставляет систему посыпать библиотекам уведомление `DLL_PROCESS_DETACH`. Перехватывая вызов `ExitProcess`, мы гарантируем своевременное уведомление внедренной DLL о вызове этой функции. Причем уведомление приходит до того, как аналогичные уведомления посыпаются другим DLL. В этот момент внедренная DLL узнаёт о завершении процесса и успевает провести корректную очистку. Далее вызывается функция `ExitProcess`, что приводит к рассылке уведомлений `DLL_PROCESS_DETACH` остальным DLL, и они корректно завершаются. Это же уведомление получает и внедренная DLL, но ничего особенного она не делает, так как уже выполнила свою задачу.

В этом примере внедрение DLL происходило как бы само по себе: приложение было рассчитано на загрузку именно этой DLL. Оказываясь в адресном пространстве процесса, DLL должна была просканировать EXE-модуль и все загружаемые DLL-модули, найти все обращения к `ExitProcess` и заменить их вызовами функции, находящейся во внедренной DLL. (Эта задача не так сложна, как кажется.) Подставная функция (функция ловушки), закончив свою работу, вызывала настоящую функцию `ExitProcess` из `Kernel32.dll`.

Данный пример иллюстрирует типичное применение перехвата API-вызовов, который позволил решить насущную проблему при минимуме дополнительного кода.

Перехват API-вызовов подменой кода

Перехват API-вызовов далеко не новый метод — разработчики пользуются им уже многие годы. Когда сталкиваешься с проблемой, аналогичной той, о которой я только что рассказал, то первое, что приходит в голову, — установить ловушку, подменив часть исходного кода. Вот как это делается.

- Найдите адрес функции, вызов которой Вы хотите перехватывать (например, `ExitProcess` в `Kernel32.dll`).
- Сохраните несколько первых байтов этой функции в другом участке памяти.

3. На их место вставьте машинную команду JUMP для перехода по адресу подставной функции. Естественно, сигнатура Вашей функции должна быть такой же, как и исходной, т. е. все параметры, возвращаемое значение и правила вызова должны совпадать.
4. Теперь, когда поток вызовет перехватываемую функцию, команда JUMP направит его к Вашей функции. На этом этапе Вы можете выполнить любой нужный код.
5. Снимите ловушку, восстановив ранее сохраненные (в п. 2) байты.
6. Если теперь вызвать перехватываемую функцию (таковой больше не являющаяся), она будет работать так, как работала до установки ловушки.
7. После того как она вернет управление, Вы можете выполнить операции 2 и 3 и тем самым вновь поставить ловушку на эту функцию.

Этот метод был очень популярен среди программистов, создававших приложения для 16-разрядной Windows, и отлично работал в этой системе. В современных системах у этого метода возникло несколько серьезных недостатков, и я настоятельно не рекомендую его применять. Во-первых, он создает зависимость от конкретного процессора из-за команды JUMP, и, кроме того, приходится вручную писать машинные коды. Во-вторых, в системе с вытесняющей многозадачностью данный метод вообще не годится. На замену кода в начале функции уходит какое-то время, а в этот момент перехватываемая функция может понадобиться другому потоку. Результаты могут быть просто катастрофическими!

WINDOWS 98 В Windows 98 основные системные DLL (Kernel32, AdvAPI32, User32 и GDI32) защищены так, что приложение не может что-либо изменить на их страницах кода. Это ограничение можно обойти, только написав специальный драйвер виртуального устройства (VxD).

Перехват API-вызовов с использованием раздела импорта

Данный способ API-перехвата решает обе упомянутые мной проблемы. Он прост и довольно надежен. Но для его понимания нужно иметь представление о том, как осуществляется динамическое связывание. В частности, Вы должны разбираться в структуре раздела импорта модуля. В главе 19 я достаточно подробно объяснил, как создается этот раздел и что в нем находится. Читая последующий материал, Вы всегда можете вернуться к этой главе.

Как Вам уже известно, в разделе импорта содержится список DLL, необходимых модулю для нормальной работы. Кроме того, в нем перечислены все идентификаторы, которые модуль импортирует из каждой DLL. Вызывая импортируемую функцию, поток получает ее адрес фактически из раздела импорта.

Поэтому, чтобы перехватить определенную функцию, надо лишь изменить ее адрес в разделе импорта. Все! И никакой зависимости от процессорной платформы. А поскольку Вы ничего не меняете в коде функции, то и о синхронизации потоков можно не беспокоиться.

Вот функция, которая делает эту сказку былью. Она ищет в разделе импорта модуля ссылку на идентификатор по определенному адресу и, найдя ее, подменяет адрес соответствующего идентификатора.

```
void ReplaceIATEntryInOneMod(PCSTR pszCalleeModName,
    PROC pfnCurrent, PROC pfnNew, HMODULE hmodCaller) {
```

```

ULONG ulSize;
PIMAGE_IMPORT_DESCRIPTOR pImportDesc = (PIMAGE_IMPORT_DESCRIPTOR)
    ImageDirectoryEntryToData(hmodCaller, TRUE,
        IMAGE_DIRECTORY_ENTRY_IMPORT, &ulSize);

if (pImportDesc == NULL)
    return; // в этом модуле нет раздела импорта

// находим дескриптор раздела импорта со ссылками
// на функции DLL (вызываемого модуля)
for (; pImportDesc->Name; pImportDesc++) {
    PSTR pszModName = (PSTR)
        ((PBYTE) hmodCaller + pImportDesc->Name);
    if (lstrcmpiA(pszModName, pszCalleeModName) == 0)
        break;
}

if (pImportDesc->Name == 0)
    // этот модуль не импортирует никаких функций из данной DLL
    return;

// получаем таблицу адресов импорта (IAT) для функций DLL
PIMAGE_THUNK_DATA pThunk = (PIMAGE_THUNK_DATA)
    ((PBYTE) hmodCaller + pImportDesc->FirstThunk);

// заменяем адреса исходных функций адресами своих функций
for (; pThunk->u1.Function; pThunk++) {

    // получаем адрес адреса функции
    PROC* ppfn = (PROC*) &pThunk->u1.Function;

    // та ли это функция, которая нас интересует?
    BOOL fFound = (*ppfn == pfnCurrent);

    // см. текст программы-примера, в котором
    // содержится трюковый код для Windows 98

    if (fFound) {
        // адреса сходятся; изменяем адрес в разделе импорта
        WriteProcessMemory(GetCurrentProcess(), ppfn, &pfnNew,
            sizeof(pfnNew), NULL);
        return; // получилось; выходим
    }
}

// если мы попали сюда, значит, в разделе импорта
// нет ссылки на нужную функцию
}

```

Чтобы понять, как вызывать эту функцию, представьте, что у нас есть модуль с именем DataBase.exe. Он вызывает *ExitProcess* из Kernel32.dll, но мы хотим, чтобы он обращался к *MyExitProcess* в нашем модуле DBExtend.dll. Для этого надо вызвать *ReplaceIATEntryInOneMod* следующим образом.

```
PROC pfnOrig = GetProcAddress(GetModuleHandle("Kernel32"), "ExitProcess");
HMODULE hmodCaller = GetModuleHandle("DataBase.exe");

void ReplaceIATEntryInOneMod(
    "Kernel32.dll", // модуль, содержащий ANSI-функцию
    pfnOrig,        // адрес исходной функции в вызываемой DLL
    MyExitProcess, // адрес заменяющей функции
    hmodCaller);   // описатель модуля, из которого надо вызывать новую функцию
```

Первое, что делает *ReplaceIATEntryInOneMod*, — находит в модуле *hmodCaller* раздел импорта. Для этого она вызывает *ImageDirectoryEntryToData* и передает ей IMAGE_DIRECTORY_ENTRY_IMPORT. Если последняя функция возвращает NULL, значит, в модуле DataBase.exe такого раздела нет, и на этом все заканчивается.

Если же в DataBase.exe раздел импорта присутствует, то *ImageDirectoryEntryToData* возвращает его адрес как указатель типа PIMAGE_IMPORT_DESCRIPTOR. Тогда мы должны искать в разделе импорта DLL, содержащую требуемую импортируемую функцию. В данном примере мы ищем идентификаторы, импортируемые из Kernel32.dll (имя которой указывается в первом параметре *ReplaceIATEntryInOneMod*). В цикле *for* сканируются имена DLL. Заметьте, что в разделах импорта все строки имеют формат ANSI (*Unicode* не применяется). Вот почему язываю функцию *lstrcmpA*, а не макрос *lstrcmpi*.

Если программа не найдет никаких ссылок на идентификаторы в Kernel32.dll, то и в этом случае функция просто вернет управление и ничего делать не станет. А если такие ссылки есть, мы получим адрес массива структур IMAGE_THUNK_DATA, в котором содержится информация об импортируемых идентификаторах. Далее в списке из Kernel32.dll ведется поиск идентификатора с адресом, совпадающим с искомым. В данном случае мы ищем адрес, соответствующий адресу функции *ExitProcess*.

Если такого адреса нет, значит, данный модуль не импортирует нужный идентификатор, и *ReplaceIATEntryInOneMod* просто возвращает управление. Но если адрес обнаруживается, мы вызываем *WriteProcessMemory*, чтобы заменить его на адрес подставной функции. Я применяю *WriteProcessMemory*, а не *InterlockedExchangePointer*, потому что она изменяет байты, не обращая внимания на тип защиты страницы памяти, в которой эти байты находятся. Так, если страница имеет атрибут защиты PAGE_READONLY, вызов *InterlockedExchangePointer* приведет к нарушению доступа, а *WriteProcessMemory* сама модифицирует атрибуты защиты и без проблем выполнит свою задачу.

С этого момента любой поток, выполняющий код в модуле DataBase.exe, при обращении к *ExitProcess* будет вызывать нашу функцию. А из нее мы сможем легко получить адрес исходной функции *ExitProcess* в Kernel32.dll и при необходимости вызвать ее.

Обратите внимание, что *ReplaceIATEntryInOneMod* подменяет вызовы функций только в одном модуле. Если в его адресном пространстве присутствует другая DLL, использующая *ExitProcess*, она будет вызывать именно *ExitProcess* из Kernel32.dll.

Если Вы хотите перехватывать обращения к *ExitProcess* из всех модулей, Вам придется вызывать *ReplaceIATEntryInOneMod* для каждого модуля в адресном пространстве процесса. Я, кстати, написал еще одну функцию, *ReplaceIATEntryInAllMods*. С помощью Toolhelp-функций она перечисляет все модули, загруженные в адресное пространство процесса, и для каждого из них вызывает *ReplaceIATEntryInOneMod*, передавая в качестве последнего параметра описатель соответствующего модуля.

Но и в этом случае могут быть проблемы. Например, что получится, если после вызова *ReplaceIATEntryInAllMods* какой-нибудь поток вызовет *LoadLibrary* для загрузки

новой DLL? Если в только что загруженной DLL имеются вызовы *ExitProcess*, она будет обращаться не к Вашей функции, а к исходной. Для решения этой проблемы Вы должны перехватывать функции *LoadLibraryA*, *LoadLibraryW*, *LoadLibraryExA* и *LoadLibraryExW* и вызывать *ReplaceIATEntryInOneMod* для каждого загружаемого модуля.

И, наконец, есть еще одна проблема, связанная с *GetProcAddress*. Допустим, поток выполняет такой код:

```
typedef int (WINAPI *PfnExitProcess)(UINT uExitCode);
PfnExitProcess pfnExitProcess = (PfnExitProcess) GetProcAddress(
    GetModuleHandle("Kernel32"), "ExitProcess");
pfnExitProcess(0);
```

Этот код сообщает системе, что надо получить истинный адрес *ExitProcess* в Kernel32.dll, а затем сделать вызов по этому адресу. Данный код будет выполнен в обход Вашей подставной функции. Проблема решается перехватом обращений к *GetProcAddress*. При ее вызове Вы должны возвращать адрес своей функции.

В следующем разделе я покажу, как на практике реализовать перехват API-вызовов и решить все проблемы, связанные с использованием *LoadLibrary* и *GetProcAddress*.

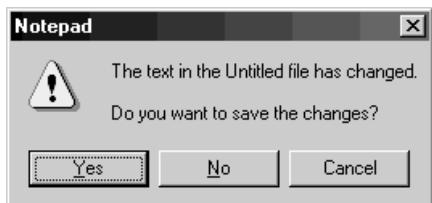
Программа-пример LastMsgBoxInfo

Эта программа, «22 LastMsgBoxInfo.exe» (см. листинг на рис. 22-5), демонстрирует перехват API-вызовов. Она перехватывает все обращения к функции *MessageBox* из User32.dll. Для этого программа внедряет DLL с использованием ловушек. Файлы исходного кода и ресурсов этой программы и DLL находятся в каталогах 22-LastMsgBoxInfo и 22-LastMsgBoxInfoLib на компакт-диске, прилагаемом к книге.*

После запуска LastMsgBoxInfo открывает диалоговое окно, показанное ниже.



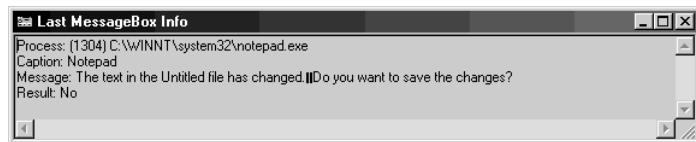
В этот момент программа находится в состоянии ожидания. Запустите какое-нибудь приложение и заставьте его открыть окно с тем или иным сообщением. Тестируя свою программу, я запускал Notepad, набирал произвольный текст, а затем пытался закрыть его окно, не сохранив набранный текст. Это заставляло Notepad выводить вот такое окно с предложением сохранить документ.



* Как сообщил сам автор, эта программа, к сожалению, работает не со всеми приложениями. Чтобы исправить эту ошибку, Вы должны модифицировать метод *CAPIHook::ReplaceIATEntryInOneMod* так, чтобы перед вызовом *WriteProcessMemory* он обращался к *VirtualProtect*:

```
DWORD dwDummy;
VirtualProtect(ppfn, sizeof(ppfn), PAGE_EXECUTE_READWRITE, &dwDummy);
```

После отказа от сохранения документа диалоговое окно LastMsgBoxInfo приобретает следующий вид.



Как видите, LastMsgBoxInfo позволяет наблюдать за вызовами функции *MessageBox* из других процессов.

Код, отвечающий за вывод диалогового окна LastMsgBoxInfo и управление им весьма прост. Трудности начинаются при настройке перехвата API-вызовов. Чтобы упростить эту задачу, я создал C++-класс CAPIHook, определенный в заголовочном файле APIHook.h и реализованный в файле APIHook.cpp. Пользоваться им очень легко, так как в нем лишь несколько открытых функций-членов: конструктор, деструктор и метод, возвращающий адрес исходной функции, на которую Вы ставите ловушку.

Для перехвата вызова какой-либо функции Вы просто создаете экземпляр этого класса:

```
CAPIHook g_MessageBoxA("User32.dll", "MessageBoxA",
    (PROC) Hook_MessageBoxA, TRUE);
```

```
CAPIHook g_MessageBoxW("User32.dll", "MessageBoxW",
    (PROC) Hook_MessageBoxW, TRUE);
```

Мне приходится ставить ловушки на две функции: *MessageBoxA* и *MessageBoxW*. Обе эти функции находятся в User32.dll. Я хочу, чтобы при обращении к *MessageBoxA* вызывалась *Hook_MessageBoxA*, а при вызове *MessageBoxW* — *Hook_MessageBoxW*.

Конструктор класса CAPIHook просто запоминает, какую API-функцию нужно перехватывать, и вызывает *ReplaceIATEntryInAllMods*, которая, собственно, и выполняет эту задачу.

Следующая открытая функция-член — деструктор. Когда объект CAPIHook выходит за пределы области видимости, деструктор вызывает *ReplaceIATEntryInAllMods* для восстановления исходного адреса идентификатора во всех модулях, т. е. для снятия ловушки.

Третий открытый член класса возвращает адрес исходной функции. Эта функция обычно вызывается из подставной функции для обращения к перехватываемой функции. Вот как выглядит код функции *Hook_MessageBoxA*:

```
int WINAPI Hook_MessageBoxA(HWND hWnd, PCSTR pszText,
    PCSTR pszCaption, UINT uType) {

    int nResult = ((PFNMESSAGEBOXA)(PROC) g_MessageBoxA)
        (hWnd, pszText, pszCaption, uType);
    SendLastMsgBoxInfo(FALSE, (PVOID) pszCaption, (PVOID) pszText, nResult);
    return(nResult);
}
```

Этот код ссылается на глобальный объект *g_MessageBoxA* класса CAPIHook. Приведение его к типу PROC заставляет функцию-член вернуть адрес исходной функции *MessageBoxA* в User32.dll.

Всю работу по установке и снятию ловушек этот C++-класс берет на себя. До конца просмотра файла APIHook.cpp, Вы заметите, что мой C++-класс автоматически

создает экземпляры объектов CAPIHook для перехвата вызовов *LoadLibraryA*, *LoadLibraryW*, *LoadLibraryExA*, *LoadLibraryExW* и *GetProcAddress*. Так что он сам справляется с проблемами, о которых я рассказывал в предыдущем разделе.



LastMsgBoxInfo.cpp

```
/****************************************************************************
Modуль: LastMsgBoxInfo.cpp
Автор: Copyright (c) 2000, Джейфри Рихтер (Jeffrey Richter)
****

#include "..\CmnHdr.h"      /* см. приложение A */
#include <windowsx.h>
#include <tchar.h>
#include "Resource.h"
#include "..\22-LastMsgBoxInfoLib\LastMsgBoxInfoLib.h"

///////////

BOOL Dlg_OnInitDialog(HWND hwnd, HWND hwndFocus, LPARAM lParam) {

    chSETDLGICONS(hwnd, IDI_LASTMSGBOXINFO);
    SetDlgItemText(hwnd, IDC_INFO,
        TEXT("Waiting for a Message Box to be dismissed"));
    return(TRUE);
}

///////////

void Dlg_OnSize(HWND hwnd, UINT state, int cx, int cy) {

    SetWindowPos(GetDlgItem(hwnd, IDC_INFO), NULL,
        0, 0, cx, cy, SWP_NOZORDER);
}

///////////

void Dlg_OnCommand(HWND hwnd, int id, HWND hwndCtl, UINT codeNotify) {
    switch (id) {
        case IDCANCEL:
            EndDialog(hwnd, id);
            break;
    }
}

///////////

BOOL Dlg_OnCopyData(HWND hwnd, HWND hwndFrom, PCOPYDATASTRUCT pcds) {

    // какой-то из перехватываемых процессов прислал информацию
    // об открытом им окне с сообщением; выводим ее на экран
}
```

Рис. 22-5. Программа-пример *LastMsgBoxInfo*

см. след. стр.

Рис. 22-5. продолжение

```

SetDlgItemTextA(hwnd, IDC_INFO, (PCSTR) pcds->lpData);
return(TRUE);
}

/////////////////////////////// Конец файла //////////////////////////////

INT_PTR WINAPI Dlg_Proc(HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam) {

    switch (uMsg) {
        chHANDLE_DLGMMSG(hwnd, WM_INITDIALOG, Dlg_OnInitDialog);
        chHANDLE_DLGMMSG(hwnd, WM_SIZE, Dlg_OnSize);
        chHANDLE_DLGMMSG(hwnd, WM_COMMAND, Dlg_OnCommand);
        chHANDLE_DLGMMSG(hwnd, WM_COPYDATA, Dlg_OnCopyData);
    }
    return(FALSE);
}

/////////////////////////////// Конец файла //////////////////////////////

int WINAPI _tWinMain(HINSTANCE hinstExe, HINSTANCE, PTSTR pszCmdLine, int) {

    DWORD dwThreadId = 0;
#ifdef _DEBUG
    HWND hwnd = FindWindow(NULL, TEXT("Untitled - Paint"));
    dwThreadId = GetWindowThreadProcessId(hwnd, NULL);
#endif

    LastMsgBoxInfo_HookAllApps(TRUE, dwThreadId);
    DialogBox(hinstExe, MAKEINTRESOURCE(IDD_LASTMSGBOXINFO), NULL, Dlg_Proc);
    LastMsgBoxInfo_HookAllApps(FALSE, 0);
    return(0);
}

/////////////////////////////// Конец файла //////////////////////////////

```

LastMsgBoxInfoLib.cpp

```

//*****************************************************************************
Modуль: LastMsgBoxInfoLib.cpp
Автор: Copyright (c) 2000, Джейфри Рихтер (Jeffrey Richter)
*****/

#define WINVER      0x0500
#include " \CmnHdr.h"
#include <WindowsX.h>
#include <tchar.h>
#include <stdio.h>
#include "APIHook.h"

#define LASTMSGBOXINFOLIBAPI extern "C" __declspec(dllexport)
#include "LastMsgBoxInfoLib.h"

```

Рис. 22-5. продолжение

```
//////////  

// прототипы перехватываемых функций  

typedef int (WINAPI *PFNMESSAGEBOXA)(HWND hWnd, PCSTR pszText,  

    PCSTR pszCaption, UINT uType);  

typedef int (WINAPI *PFNMESSAGEBOXW)(HWND hWnd, PCWSTR pszText,  

    PCWSTR pszCaption, UINT uType);  

// нам приходится ссылаться на эти переменные до их создания  

extern CAPIHook g_MessageBoxA;  

extern CAPIHook g_MessageBoxW;  

//////////  

// эта функция отсылает сообщение MessageBox в наше диалоговое окно  

void SendLastMsgBoxInfo(BOOL fUnicode,  

    PVOID pvCaption, PVOID pvText, int nResult) {  

    // получаем полное имя процесса, открывшего окно с сообщением  

    char szProcessPathname[MAX_PATH];  

    GetModuleFileNameA(NULL, szProcessPathname, MAX_PATH);  

    // преобразуем возвращенное значение в строку, понятную человеку  

    PCSTR pszResult = "(Unknown)";  

    switch (nResult) {  

        case IDOK:      pszResult = "Ok";           break;  

        case IDCANCEL:   pszResult = "Cancel";       break;  

        case IDABORT:   pszResult = "Abort";        break;  

        case IDRETRY:   pszResult = "Retry";        break;  

        case IDIGNORE:  pszResult = "Ignore";       break;  

        case IDYES:     pszResult = "Yes";          break;  

        case IDNO:      pszResult = "No";           break;  

        case IDCLOSE:   pszResult = "Close";        break;  

        case IDHELP:    pszResult = "Help";         break;  

        case IDTRYAGAIN: pszResult = "Try Again";   break;  

        case IDCONTINUE: pszResult = "Continue";   break;  

    }  

    // формируем строку для отсылки в наше диалоговое окно  

    char sz[2048];  

    wsprintfA(sz, fUnicode  

        ? "Process: (%d) %s\r\nCaption: %S\r\nMessage: %S\r\nResult: %s"  

        : "Process: (%d) %s\r\nCaption: %s\r\nMessage: %s\r\nResult: %s",  

        GetCurrentProcessId(), szProcessPathname,  

        pvCaption, pvText, pszResult);  

    // отсылаем ее в наше диалоговое окно  

    COPYDATASTRUCT cds = { 0, lstrlenA(sz) + 1, sz };  

    FORWARD_WM_COPYDATA(FindWindow(NULL, TEXT("Last MessageBox Info")),  

        NULL, &cds, SendMessage);  

}
```

см. след. стр.

Рис. 22-5. продолжение

```
//////////  
  
// функция, заменяющая MessageBoxW  
int WINAPI Hook_MessageBoxW(HWND hWnd, PCWSTR pszText, LPCWSTR pszCaption,  
    UINT uType) {  
  
    // вызываем исходную функцию MessageBoxW  
    int nResult = ((PFNMESSAGEBOXW)(PROC) g_MessageBoxW)  
        (hWnd, pszText, pszCaption, uType);  
  
    // посылаем полученную информацию в наше диалоговое окно  
    SendLastMsgBoxInfo(TRUE, (PVOID) pszCaption, (PVOID) pszText, nResult);  
  
    // возвращаем результат вызвавшему модулю  
    return(nResult);  
}  
  
//////////  
  
// функция, заменяющая MessageBoxA  
int WINAPI Hook_MessageBoxA(HWND hWnd, PCSTR pszText, PCSTR pszCaption,  
    UINT uType) {  
  
    // вызываем исходную функцию MessageBoxA  
    int nResult = ((PFNMESSAGEBOXA)(PROC) g_MessageBoxA)  
        (hWnd, pszText, pszCaption, uType);  
  
    // посылаем полученную информацию в наше диалоговое окно  
    SendLastMsgBoxInfo(FALSE, (PVOID) pszCaption, (PVOID) pszText, nResult);  
  
    // возвращаем результат вызвавшему модулю  
    return(nResult);  
}  
  
//////////  
  
// ставим ловушки на функции MessageBoxA и MessageBoxW  
CAPIHook g_MessageBoxA("User32.dll", "MessageBoxA",  
    (PROC) Hook_MessageBoxA, TRUE);  
  
CAPIHook g_MessageBoxW("User32.dll", "MessageBoxW",  
    (PROC) Hook_MessageBoxW, TRUE);  
  
// поскольку мы внедряем DLL, пользуясь Windows-ловушками,  
// нам придется сохранить описатель ловушки в общем блоке памяти  
// (кстати, в Windows 2000 этого не требуется)  
#pragma data_seg("Shared")  
HHOOK g_hhook = NULL;  
#pragma data_seg()  
#pragma comment(linker, "/Section:Shared, rws")  
  
//////////
```

Рис. 22-5. продолжение

```

static LRESULT WINAPI GetMsgProc(int code, WPARAM wParam, LPARAM lParam) {

    // Примечание: в Windows 2000 первый параметр CallNextHookEx может
    // принимать значение NULL. В Windows 98 он обязательно должен содержать
    // описатель ловушки.
    return(CallNextHookEx(g_hhook, code, wParam, lParam));
}

/////////////////////////////// Конец файла //////////////////////////////

// возвращает HMODULE, содержащий указанный адрес памяти
static HMODULE ModuleFromAddress(PVOID pv) {

    MEMORY_BASIC_INFORMATION mbi;
    return((VirtualQuery(pv, &mbi, sizeof(mbi)) != 0)
        ? (HMODULE) mbi.AllocationBase : NULL);
}

/////////////////////////////// Конец файла //////////////////////////////

BOOL WINAPI LastMsgBoxInfo_HookAllApps(BOOL fInstall, DWORD dwThreadId) {

    BOOL fOk;

    if (fInstall) {

        chASSERT(g_hhook == NULL); // повторная установка ловушки недопустима
        // устанавливаем ловушку
        g_hhook = SetWindowsHookEx(WH_GETMESSAGE, GetMsgProc,
            ModuleFromAddress(LastMsgBoxInfo_HookAllApps), dwThreadId);
        fOk = (g_hhook != NULL);

    } else {

        chASSERT(g_hhook != NULL); // нельзя снять ловушку, если она не установлена
        fOk = UnhookWindowsHookEx(g_hhook);
        g_hhook = NULL;
    }
    return(fOk);
}

/////////////////////////////// Конец файла //////////////////////////////

```

LastMsgBoxInfoLib.h

```

*****
* Модуль: LastMsgBoxInfoLib.h
* Автор: Copyright (c) 2000, Джейфри Рихтер (Jeffrey Richter)
*****
#ifndef LASTMSGBOXINFOLIBAPI

```

см. след. стр.

Рис. 22-5. продолжение

```
#define LASTMSGBOXINOLIBAPI extern "C" __declspec(dllimport)
#endif

/////////////////////////////// Конец файла //////////////////////////////
```

LASTMSGBOXINOLIBAPI BOOL WINAPI LastMsgBoxInfo_HookAllApps(BOOL fInstall,
 DWORD dwThreadId);

/////////////////////////////// Конец файла //////////////////////////////

APIHook.cpp

```
*****  
Модуль: APIHook.cpp  
Автор: Copyright (c) 2000, Джейфри Рихтер (Jeffrey Richter)  
*****  
  
#include "..\CmnHdr.h"  
#include <ImageHlp.h>  
#pragma comment(lib, "ImageHlp")  
  
#include "APIHook.h"  
#include "..\04-ProcessInfo\Toolhelp.h"  
  
/////////////////////////////// Конец файла //////////////////////////////  
  
// При выполнении приложения в Windows 98 под управлением отладчика  
// последний записывает в раздел импорта модуля указатель на заглушку,  
// которая вызывает желательную функцию. Чтобы учесть это, придется  
// выделять сумасшедшие трюки, и для них понадобятся следующие переменные.  
  
// старший адрес закрытой памяти (только в Windows 98)  
PVOID CAPIHook::sm_pvMaxAppAddr = NULL;  
const BYTE cPushOpCode = 0x68; // код PUSH-инструкции на платформе x86  
  
/////////////////////////////// Конец файла //////////////////////////////  
  
// заголовок связанного списка объектов CAPIHook  
CAPIHook* CAPIHook::sm_pHead = NULL;  
  
/////////////////////////////// Конец файла //////////////////////////////  
  
CAPIHook::CAPIHook(PSTR pszCalleeModName, PSTR pszFuncName, PROC pfnHook,  
    BOOL fExcludeAPIHookMod) {  
  
    if (sm_pvMaxAppAddr == NULL) {  
        // функции с адресами выше lpMaximumApplicationAddress  
        // требуют особой обработки (только в Windows 98)  
        SYSTEM_INFO si;  
        GetSystemInfo(&si);  
        sm_pvMaxAppAddr = si.lpMaximumApplicationAddress;  
    }  
}
```

Рис. 22-5. продолжение

```

m_pNext = sm_pHead; // этот узел был вверху списка
sm_pHead = this; // теперь вверху списка находится этот узел

// сохраняем информацию о перехватываемой функции
m_pszCalleeModName = pszCalleeModName;
m_pszFuncName = pszFuncName;
m_pfnHook = pfnHook;
m_fExcludeAPIHookMod = fExcludeAPIHookMod;
m_pfnOrig = GetProcAddressRaw(
    GetModuleHandleA(pszCalleeModName), m_pszFuncName);
chASSERT(m_pfnOrig != NULL); // такой функции нет

if (m_pfnOrig > sm_pvMaxAppAddr) {
    // адрес находится в общей DLL; его надо подправить
    PBYTE pb = (PBYTE) m_pfnOrig;
    if (pb[0] == cPushOpCode) {
        // пропускаем команду PUSH и получаем истинный адрес
        PVOID pv = * (PVOID*) &pb[1];
        m_pfnOrig = (PROC) pv;
    }
}

// перехватываем эту функцию во всех загруженных модулях
ReplaceIATEEntryInAllMods(m_pszCalleeModName, m_pfnOrig, m_pfnHook,
    m_fExcludeAPIHookMod);
}

///////////////////////////////
CAPIHook::~CAPIHook() {

    // отменяем перехват этой функции во всех модулях
    ReplaceIATEEntryInAllMods(m_pszCalleeModName, m_pfnHook, m_pfnOrig,
        m_fExcludeAPIHookMod);

    // удаляем этот объект из связанного списка
    CAPIHook* p = sm_pHead;
    if (p == this) { // удаляем верхний узел
        sm_pHead = p->m_pNext;
    } else {

        BOOL fFound = FALSE;

        // проходим по списку сверху и исправляем указатели
        for (; !fFound && (p->m_pNext != NULL); p = p->m_pNext) {
            if (p->m_pNext == this) {
                // переадресуем узел с нашей функции на следующий узел
                p->m_pNext = p->m_pNext->m_pNext;
                break;
            }
        }
    }
}

```

см. след. стр.

Рис. 22-5. продолжение

```
    chASSERT(fFound);
}

//////////////////////////////////////////////////////////////////

// Примечание: эту функцию компилятор НЕ должен подставлять в конечный код
FARPROC CAPIHook::GetProcAddressRaw(HMODULE hmod, PCSTR pszProcName) {

    return(:GetProcAddress(hmod, pszProcName));
}

//////////////////////////////////////////////////////////////////

// возвращает HMODULE, содержащий указанный адрес памяти
static HMODULE ModuleFromAddress(PVOID pv) {

    MEMORY_BASIC_INFORMATION mbi;
    return((VirtualQuery(pv, &mbi, sizeof(mbi)) != 0)
        ? (HMODULE) mbi.AllocationBase : NULL);
}

//////////////////////////////////////////////////////////////////

void CAPIHook::ReplaceIATEEntryInAllMods(PCSTR pszCalleeModName,
    PROC pfnCurrent, PROC pfnNew, BOOL fExcludeAPIHookMod) {

    HMODULE hmodThisMod = fExcludeAPIHookMod
        ? ModuleFromAddress(ReplaceIATEEntryInAllMods) : NULL;

    // получаем список модулей в данном процессе
    CToolhelp th(TH32CS_SNAPMODULE, GetCurrentProcessId());

    MODULEENTRY32 me = { sizeof(me) };
    for (BOOL f0k = th.ModuleFirst(&me); f0k; f0k = th.ModuleNext(&me)) {

        // Примечание: мы не перехватываем функции в собственном модуле
        if (me.hModule != hmodThisMod) {

            // перехватываем данную функцию в этом модуле
            ReplaceIATEEntryInOneMod(
                pszCalleeModName, pfnCurrent, pfnNew, me.hModule);
        }
    }
}

//////////////////////////////////////////////////////////////////

void CAPIHook::ReplaceIATEEntryInOneMod(PCSTR pszCalleeModName,
    PROC pfnCurrent, PROC pfnNew, HMODULE hmodCaller) {

    // получаем адрес раздела импорта модуля
```

Рис. 22-5. продолжение

```

ULONG ulSize;
PIMAGE_IMPORT_DESCRIPTOR pImportDesc = (PIMAGE_IMPORT_DESCRIPTOR)
    ImageDirectoryEntryToData(hmodCaller, TRUE,
        IMAGE_DIRECTORY_ENTRY_IMPORT, &ulSize);

if (pImportDesc == NULL)
    return; // в этом модуле нет раздела импорта

// находим дескриптор раздела импорта со ссылками
// на функции DLL (вызываемого модуля)
for (; pImportDesc->Name; pImportDesc++) {
    PSTR pszModName = ((PBYTE) hmodCaller + pImportDesc->Name);
    if (lstrcmpiA(pszModName, pszCalleeModName) == 0)
        break; // найден
}

if (pImportDesc->Name == 0)
    return; // этот модуль не импортирует никаких функций из данной DLL

// получаем таблицу адресов импорта (IAT) для функций DLL
PIMAGE_THUNK_DATA pThunk = (PIMAGE_THUNK_DATA)
    ((PBYTE) hmodCaller + pImportDesc->FirstThunk);

// заменяем адреса исходных функций адресами своих функций
for (; pThunk->u1.Function; pThunk++) {

    // получаем адрес адреса функции
    PROC* ppfn = (PROC*) &pThunk->u1.Function;

    // та ли это функция, которая нас интересует?
    BOOL fFound = (*ppfn == pfnCurrent);

    if (!fFound && (*ppfn > sm_pvMaxAppAddr)) {

        // Если это "не та" функция и ее адрес находится в общей DLL,
        // то, вероятно, мы работаем под отладчиком в Windows 98.
        // Тогда адрес указывает на инструкцию, в которой, может быть,
        // есть правильный адрес.

        PBYTE pbInFunc = (PBYTE) *ppfn;
        if (pbInFunc[0] == cPushOpCode) {
            // это инструкция PUSH; за ней следует истинный адрес функции
            ppfn = (PROC*) &pbInFunc[1];

            // та ли это функция, которая нас интересует?
            fFound = (*ppfn == pfnCurrent);
        }
    }

    if (fFound) {
        // адреса сходятся; изменяем адрес в разделе импорта
    }
}

```

см. след. стр.

Рис. 22-5. продолжение

```

        WriteProcessMemory(GetCurrentProcess(), ppfn, &pfnNew,
                           sizeof(pfnNew), NULL);
        return; // получилось; выходим
    }
}
// если мы попали сюда, значит, в разделе импорта
// нет ссылки на нужную функцию
}

///////////////////////////////



// перехватываем LoadLibrary и GetProcAddress, чтобы наши ловушки
// работали корректно и при вызове этих функций

CAPIHook CAPIHook::sm_LoadLibraryA ("Kernel32.dll", "LoadLibraryA",
                                     (PROC) CAPIHook::LoadLibraryA, TRUE);

CAPIHook CAPIHook::sm_LoadLibraryW ("Kernel32.dll", "LoadLibraryW",
                                     (PROC) CAPIHook::LoadLibraryW, TRUE);

CAPIHook CAPIHook::sm_LoadLibraryExA("Kernel32.dll", "LoadLibraryExA",
                                     (PROC) CAPIHook::LoadLibraryExA, TRUE);

CAPIHook CAPIHook::sm_LoadLibraryExW("Kernel32.dll", "LoadLibraryExW",
                                     (PROC) CAPIHook::LoadLibraryExW, TRUE);

CAPIHook CAPIHook::sm_GetProcAddress("Kernel32.dll", "GetProcAddress",
                                     (PROC) CAPIHook::GetProcAddress, TRUE);

///////////////////////////////



void CAPIHook::FixupNewlyLoadedModule(HMODULE hmod, DWORD dwFlags) {

    // если загружается новый модуль, его вызовы функций тоже перехватываются
    if ((hmod != NULL) && ((dwFlags & LOAD_LIBRARY_AS_DATAFILE) == 0)) {

        for (CAPIHook* p = sm_pHead; p != NULL; p = p->m_pNext) {
            ReplaceIATEntryInOneMod(p->m_pszCalleeModName,
                                    p->m_pfnOrig, p->m_pfnHook, hmod);
        }
    }
}

///////////////////////////////



HMODULE WINAPI CAPIHook::LoadLibraryA(PCSTR pszModulePath) {

    HMODULE hmod = ::LoadLibraryA(pszModulePath);
    FixupNewlyLoadedModule(hmod, 0);
    return(hmod);
}

```

Рис. 22-5. продолжение

```
//////////  

HMODULE WINAPI CAPIHook::LoadLibraryW(PCWSTR pszModulePath) {  

    HMODULE hmod = ::LoadLibraryW(pszModulePath);  

    FixupNewlyLoadedModule(hmod, 0);  

    return(hmod);  

}  

//////////  

HMODULE WINAPI CAPIHook::LoadLibraryExA(PCSTR pszModulePath,  

    HANDLE hFile, DWORD dwFlags) {  

    HMODULE hmod = ::LoadLibraryExA(pszModulePath, hFile, dwFlags);  

    FixupNewlyLoadedModule(hmod, dwFlags);  

    return(hmod);  

}  

//////////  

HMODULE WINAPI CAPIHook::LoadLibraryExW(PCWSTR pszModulePath,  

    HANDLE hFile, DWORD dwFlags) {  

    HMODULE hmod = ::LoadLibraryExW(pszModulePath, hFile, dwFlags);  

    FixupNewlyLoadedModule(hmod, dwFlags);  

    return(hmod);  

}  

//////////  

FARPROC WINAPI CAPIHook::GetProcAddress(HMODULE hmod, PCSTR pszProcName) {  

    // получаем истинный адрес этой функции  

    FARPROC pfn = GetProcAddressRaw(hmod, pszProcName);  

    // относится ли эта функция к числу тех, которые мы хотим перехватывать?  

    CAPIHook* p = sm_pHead;  

    for (; (pfn != NULL) && (p != NULL); p = p->m_pNext) {  

        if (pfn == p->m_pfnOrig) {  

            // возвращаем адрес перехватываемой функции  

            pfn = p->m_pfnHook;  

            break;  

        }
    }
    return(pfn);
}  

////////// Конец файла //////////
```

см. след. стр.

Рис. 22-5. продолжение**APIHook.h**

```
/*****************************************************************************  
Модуль: APIHook.h  
Автор: Copyright (c) 2000, Джейфри Рихтер (Jeffrey Richter)  
*****/  
  
#pragma once  
  
//////////////////////////////  
  
class CAPIHook {  
public:  
    // перехватывает какую-либо функцию во всех модулях  
    CAPIHook(PSTR pszCalleeModName, PSTR pszFuncName, PROC pfnHook,  
             BOOL fExcludeAPIHookMod);  
  
    // отменяет перехват функции во всех модулях  
    ~CAPIHook();  
  
    // возвращает исходный адрес перехватываемой функции  
    operator PROC() { return(m_pfnOrig); }  
  
public:  
    // вызывает настоящую GetProcAddress  
    static FARPROC WINAPI GetProcAddressRaw(HMODULE hmod, PCSTR pszProcName);  
  
private:  
    static PVOID sm_pvMaxAppAddr; // старший адрес закрытой памяти  
    static CAPIHook* sm_pHead;    // адрес первого объекта  
    CAPIHook* m_pNext;          // адрес следующего объекта  
  
    PCSTR m_pszCalleeModName;    // модуль, содержащий функцию (ANSI)  
    PCSTR m_pszFuncName;        // имя функции в вызываемой DLL (ANSI)  
    PROC m_pfnOrig;            // исходный адрес функции в вызываемой DLL  
    PROC m_pfnHook;             // адрес заменяющей функции  
    BOOL m_fExcludeAPIHookMod; // не реализует ли данный модуль CAPIHook?  
  
private:  
    // заменяет адрес идентификатора в разделах импорта всех модулей  
    static void WINAPI ReplaceIATEEntryInAllMods(PCSTR pszCalleeModName,  
                                                 PROC pfnOrig, PROC pfnHook, BOOL fExcludeAPIHookMod);  
  
    // заменяет адрес идентификатора в разделе импорта одного модуля  
    static void WINAPI ReplaceIATEEntryInOneMod(PCSTR pszCalleeModName,  
                                                PROC pfnOrig, PROC pfnHook, HMODULE hmodCaller);  
  
private:  
    // применяется, если после установки ловушки загружается новая DLL  
    static void WINAPI FixupNewlyLoadedModule(HMODULE hmod, DWORD dwFlags);
```

Рис. 22-5. продолжение

```
// используется для отслеживания загружаемых DLL
static HMODULE WINAPI LoadLibraryA(PCSTR pszModulePath);
static HMODULE WINAPI LoadLibraryW(PCWSTR pszModulePath);
static HMODULE WINAPI LoadLibraryExA(PCSTR pszModulePath,
    HANDLE hFile, DWORD dwFlags);
static HMODULE WINAPI LoadLibraryExW(PCWSTR pszModulePath,
    HANDLE hFile, DWORD dwFlags);

// возвращает адрес заменяющей функции при вызове перехватываемой
static FARPROC WINAPI GetProcAddress(HMODULE hmod, PCSTR pszProcName);

private:
    static CAPIHook sm_LoadLibraryA;
    static CAPIHook sm_LoadLibraryW;
    static CAPIHook sm_LoadLibraryExA;
    static CAPIHook sm_LoadLibraryExW;
    static CAPIHook sm_GetProcAddress;
};

/////////////////////////////// Конец файла //////////////////////
```

ЧАСТЬ V

СТРУКТУРНАЯ ОБРАБОТКА ИСКЛЮЧЕНИЙ



Обработчики завершения

Закроем глаза и помечтаем, какие бы программы мы писали, если бы сбои в них были невозможны! Представляете: памяти навалом, неверных указателей никто не передает, нужные файлы всегда на месте. Не программирование, а праздник, да? А код программ? Насколько он стал бы проще и понятнее! Без всех этих *if* и *goto*.

И если Вы давно мечтали о такой среде программирования, Вы сразу же оцените *структурную обработку исключений* (structured exception handling, SEH). Преимущество SEH в том, что при написании кода можно сосредоточиться на решении своей задачи. Если при выполнении программы возникнут неприятности, система сама обнаружит их и сообщит Вам.

Хотя полностью игнорировать ошибки в программе при использовании SEH нельзя, она все же позволяет отделить основную работу от рутинной обработки ошибок, к которой можно вернуться позже.

Главное, почему Microsoft ввела в Windows поддержку SEH, было ее стремление упростить разработку операционной системы и повысить ее надежность. А нам SEH поможет сделать надежнее наши программы.

Основная нагрузка по поддержке SEH ложится на компилятор, а не на операционную систему. Он генерирует специальный код на входах и выходах блоков исключений (exception blocks), создает таблицы вспомогательных структур данных для поддержки SEH и предоставляет функции обратного вызова, к которым система могла бы обращаться для прохода по блокам исключений. Компилятор отвечает и за формирование стековых фреймов (stack frames) и другой внутренней информации, используемой операционной системой. Добавить поддержку SEH в компилятор — задача не из легких, поэтому не удивляйтесь, когда увидите, что разные поставщики по-разному реализуют SEH в своих компиляторах. К счастью, на детали реализации можно не обращать внимания, а просто задействовать возможности компилятора в поддержке SEH.

Различия в реализации SEH разными компиляторами могли бы затруднить описание конкретных примеров использования SEH. Но большинство поставщиков компиляторов придерживается синтаксиса, рекомендованного Microsoft. Синтаксис и ключевые слова в моих примерах могут отличаться от применяемых в других компиляторах, но основные концепции SEH везде одинаковы. В этой главе я использую синтаксис компилятора Microsoft Visual C++.



Не путайте SEH с обработкой исключений в C++, которая представляет собой еще одну форму обработки исключений, построенную на применении ключевых слов языка C++ *catch* и *throw*. При этом Microsoft Visual C++ использует преимущества поддержки SEH, уже обеспеченной компилятором и операционными системами Windows.

SEH предоставляет две основные возможности: обработку завершения (termination handling) и обработку исключений (exception handling). В этой главе мы рассмотрим обработку завершения.

Обработчик завершения гарантирует, что блок кода (собственно обработчик) будет выполнен независимо от того, как происходит выход из другого блока кода — защищенного участка программы. Синтаксис обработчика завершения при работе с компилятором Microsoft Visual C++ выглядит так:

```
_try {
    // защищенный блок
    :
}
finally {
    // обработчик завершения
    :
}
```

Ключевые слова `_try` и `_finally` обозначают два блока обработчика завершения. В предыдущем фрагменте кода совместные действия операционной системы и компилятора гарантируют, что код блока `finally` обработчика завершения будет выполнен независимо от того, как произойдет выход из защищенного блока. И неважно, разместите Вы в защищенном блоке операторы `return`, `goto` или даже `longjmp` — обработчик завершения все равно будет вызван. Далее я покажу Вам несколько примеров использования обработчиков завершения.

Примеры использования обработчиков завершения

Поскольку при использовании SEH компилятор и операционная система вместе контролируют выполнение Вашего кода, то лучший, на мой взгляд, способ продемонстрировать работу SEH — изучать исходные тексты программ и рассматривать порядок выполнения операторов в каждом из примеров.

Поэтому в следующих разделах приведен ряд фрагментов исходного кода, а связанный с каждым из фрагментов текст поясняет, как компилятор и операционная система изменяют порядок выполнения кода.

Funcenstein1

Чтобы оценить последствия применения обработчиков завершения, рассмотрим более конкретный пример:

```
DWORD Funcenstein1() {
    DWORD dwTemp;

    // 1. Что-то делаем здесь
    :
    _try {
        // 2. Запрашиваем разрешение на доступ
        // к защищенным данным, а затем используем их
        WaitForSingleObject(g_hSem, INFINITE);

        g_dwProtectedData = 5;
        dwTemp = g_dwProtectedData;
    }
```

см. след. стр.

```
--finally {
    // 3. Даем и другим попользоваться защищенными данными
    ReleaseSemaphore(g_hSem, 1, NULL);
}

// 4. Продолжаем что-то делать
return(dwTemp);
}
```

Пронумерованные комментарии подсказывают, в каком порядке будет выполнятся этот код. Использование в *Funcenstein1* блоков *try-finally* на самом деле мало что дает. Код ждет освобождения семафора, изменяет содержимое защищенных данных, сохраняет новое значение в локальной переменной *dwTemp*, освобождает семафор и возвращает новое значение тому, кто вызвал эту функцию.

Funcenstein2

Теперь чуть-чуть изменим код функции и посмотрим, что получится:

```
DWORD Funcenstein2() {
    DWORD dwTemp;

    // 1. Что-то делаем здесь
    :

    --try {
        // 2. Запрашиваем разрешение на доступ
        // к защищенным данным, а затем используем их
        WaitForSingleObject(g_hSem, INFINITE);

        g_dwProtectedData = 5;
        dwTemp = g_dwProtectedData;

        // возвращаем новое значение
        return(dwTemp);
    }
    --finally {
        // 3. Даем и другим попользоваться защищенными данными
        ReleaseSemaphore(g_hSem, 1, NULL);
    }

    // продолжаем что-то делать - в данной версии
    // этот участок кода никогда не выполняется
    dwTemp = 9;
    return(dwTemp);
}
```

В конец блока *try* в функции *Funcenstein2* добавлен оператор *return*. Он сообщает компилятору, что Вы хотите выйти из функции и вернуть значение переменной *dwTemp* (в данный момент равное 5). Но, если будет выполнен *return*, текущий поток никогда не освободит семафор, и другие потоки не получат шанса занять этот семафор. Такой порядок выполнения грозит вылиться в действительно серьезную проблему: ведь потоки, ожидающие семафора, могут оказаться не в состоянии возобновить свое выполнение.

Применив обработчик завершения, мы не допустили преждевременного выполнения оператора *return*. Когда *return* пытается реализовать выход из блока *try*, компи-

лятор проверяет, чтобы сначала был выполнен код в блоке *finally*, — причем до того, как оператору *return* в блоке *try* будет позволено реализовать выход из функции. Вызов *ReleaseSemaphore* в обработчике завершения (в функции *Funcenstein2*) гарантирует освобождение семафора — поток не сможет случайно сохранить права на семафор и тем самым лишить процессорного времени все ожидающие этот семафор потоки.

После выполнения блока *finally* функция фактически завершает работу. Любой код за блоком *finally* не выполняется, поскольку возврат из функции происходит внутри блока *try*. Так что функция возвращает 5 и никогда — 9.

Каким же образом компилятор гарантирует выполнение блока *finally* до выхода из блока *try*? Дело вот в чем. Просматривая исходный текст, компилятор видит, что Вы вставили *return* внутрь блока *try*. Тогда он генерирует код, который сохраняет возвращаемое значение (в нашем примере 5) в созданной им же временной переменной. Затем создает код для выполнения инструкций, содержащихся внутри блока *finally*, — это называется *локальной раскруткой* (local unwind). Точнее, локальная раскрутка происходит, когда система выполняет блок *finally* из-за преждевременного выхода из блока *try*. Значение временной переменной, сгенерированной компилятором, возвращается из функции после выполнения инструкций в блоке *finally*.

Как видите, чтобы все это вытянуть, компилятору приходится генерировать дополнительный код, а системе — выполнять дополнительную работу. На разных типах процессоров поддержка обработчиков завершения реализуется по-разному. Например, процессору Alpha понадобится несколько сотен или даже тысячи машинных команд, чтобы перехватить преждевременный возврат из *try* и вызвать код блока *finally*. Поэтому лучше не писать код, вызывающий преждевременный выход из блока *try* обработчика завершения, — это может отрицательно оказаться на быстродействии программы. Чуть позже мы обсудим ключевое слово *_leave*, которое помогает избежать написания кода, приводящего к локальной раскрутке.

Обработка исключений предназначена для перехвата тех исключений, которые происходят не слишком часто (в нашем случае — преждевременного возврата). Если же какое-то исключение — чуть ли не норма, гораздо эффективнее проверять его явно, не полагаясь на SEH.

Заметьте: когда поток управления выходит из блока *try* естественным образом (как в *Funcenstein1*), издержки от вызова блока *finally* минимальны. При использовании компилятора Microsoft на процессорах x86 для входа в *finally* при нормальном выходе из *try* исполняется всего одна машинная команда — вряд ли Вы заметите ее влияние на быстродействие своей программы. Но издержки резко возрастают, если компилятору придется генерировать дополнительный код, а операционной системе — выполнять дополнительную работу, как в *Funcenstein2*.

Funcenstein3

Снова изменим код функции:

```
DWORD Funcenstein3() {
    DWORD dwTemp;

    // 1. Что-то делаем здесь
    :

    __try {
        // 2. Запрашиваем разрешение на доступ
        // к защищенным данным, а затем используем их
```

см. след. стр.

```
WaitForSingleObject(g_hSem, INFINITE);

g_dwProtectedData = 5;
dwTemp = g_dwProtectedData;

// пытаемся перескочить через блок finally
goto ReturnValue;
}

__finally {
    // 3. Даем и другим попользоваться защищенными данными
    ReleaseSemaphore(g_hSem, 1, NULL);
}

dwTemp = 9;
// 4. Продолжаем что-то делать
ReturnValue:
    return(dwTemp);
}
```

Обнаружив в блоке *try* функции *Funcenstein3* оператор *goto*, компилятор генерирует код для локальной раскрутки, чтобы сначала выполнялся блок *finally*. Но на этот раз после *finally* исполняется код, расположенный за меткой *ReturnValue*, так как возврат из функции не происходит ни в блоке *try*, ни в блоке *finally*. В итоге функция возвращает 5. И опять, поскольку Вы прервали естественный ход потока управления из *try* в *finally*, быстродействие программы — в зависимости от типа процессора — может снизиться весьма значительно.

Funcfurter1

А сейчас разберем другой сценарий, в котором обработка завершения действительно полезна:

```
DWORD Funcfurter1() {
    DWORD dwTemp;

    // 1. Что-то делаем здесь
    :
    __try {
        // 2. Запрашиваем разрешение на доступ
        // к защищенным данным, а затем используем их
        WaitForSingleObject(g_hSem, INFINITE);

        dwTemp = Funcinator(g_dwProtectedData);
    }

    __finally {
        // 3. Даем и другим попользоваться защищенными данными
        ReleaseSemaphore(g_hSem, 1, NULL);
    }

    // 4. Продолжаем что-то делать
    return(dwTemp);
}
```

Допустим, в функции *Funcinator*, вызванной из блока *try*, — «жучок», из-за которого возникает нарушение доступа к памяти. Без SEH пользователь в очередной раз увидел бы самое известное диалоговое окно Application Error. Стоит его закрыть — завершится и приложение. Если бы процесс завершился (из-за неправильного доступа к памяти), семафор остался бы занят — соответственно и ожидающие его потоки не получили бы процессорное время. Но вызов *ReleaseSemaphore* в блоке *finally* гарантирует освобождение семафора, даже если нарушение доступа к памяти происходит в какой-то другой функции.

Раз обработчик завершения — такое мощное средство, способное перехватывать завершение программы из-за неправильного доступа к памяти, можно смело рассчитывать и на то, что оно также перехватит комбинации *setjmp/longjmp* и элементарные операторы типа *break* и *continue*.

Проверьте себя: *FuncaDoodleDoo*

Посмотрим, отгадаете ли Вы, что именно возвращает следующая функция:

```
DWORD FuncaDoodleDoo() {
    DWORD dwTemp = 0;

    while (dwTemp < 10) {

        __try {
            if (dwTemp == 2)
                continue;

            if (dwTemp == 3)
                break;
        }

        __finally {
            dwTemp++;
        }

        dwTemp++;
    }

    dwTemp += 10;
    return(dwTemp);
}
```

Проанализируем эту функцию шаг за шагом. Сначала *dwTemp* приравнивается 0. Код в блоке *try* выполняется, но ни одно из условий в операторах *if* не дает TRUE, и поток управления естественным образом переходит в блок *finally*, где *dwTemp* увеличивается до 1. Затем инструкция после блока *finally* снова увеличивает значение *dwTemp*, приравнивая его 2.

На следующей итерации цикла *dwTemp* равно 2, поэтому выполняется оператор *continue* в блоке *try*. Без обработчика завершения, вызывающего принудительное выполнение блока *finally* перед выходом из *try*, управление было бы передано непосредственно в начало цикла *while*, значение *dwTemp* больше бы не менялось — и мы в бесконечном цикле! В присутствии же обработчика завершения система обнаруживает, что оператор *continue* приводит к преждевременному выходу из *try*, и передает управление блоку *finally*. Значение *dwTemp* в нем увеличивается до 3, но код за этим

блоком не выполняется, так как управление снова передается оператору *continue*, и мы вновь в начале цикла.

Теперь обрабатываем третий проход цикла. На этот раз значение выражения в первом *if* равно FALSE, а во втором — TRUE. Система снова перехватывает нашу попытку прервать выполнение блока *try* и обращается к коду *finally*. Значение *dwTemp* увеличивается до 4. Так как выполнен оператор *break*, выполнение возобновляется после тела цикла. Поэтому код, расположенный за блоком *finally* (но в теле цикла), не выполняется. Код, расположенный за телом цикла, добавляет 10 к значению *dwTemp*, что дает в итоге 14, — это и есть результат вызова функции. Даже не стану убеждать Вас никогда не писать такой код, как в *FuncADoodleDoo*. Я-то включил *continue* и *break* в середину кода, только чтобы продемонстрировать поведение обработчика завершения.

Хотя обработчик завершения справляется с большинством ситуаций, в которых выход из блока *try* был бы преждевременным, он не может вызвать выполнение блока *finally* при завершении потока или процесса. Вызов *ExitThread* или *ExitProcess* сразу завершит поток или процесс — без выполнения какого-либо кода в блоке *finally*. То же самое будет, если Ваш поток или процесс погибнут из-за того, что некая программа вызвала *TerminateThread* или *TerminateProcess*. Некоторые функции библиотеки C (вроде *abort*), в свою очередь вызывающие *ExitProcess*, тоже исключают выполнение блока *finally*. Раз Вы не можете запретить другой программе завершение какого-либо из своих потоков или процессов, так хоть сами не делайте преждевременных вызовов *ExitThread* и *ExitProcess*.

Funcenstein4

Рассмотрим еще один сценарий обработки завершения.

```
DWORD Funcenstein4() {
    DWORD dwTemp;

    // 1. Что-то делаем здесь
    :
    __try {
        // 2. Запрашиваем разрешение на доступ
        // к защищенным данным, а затем используем их
        WaitForSingleObject(g_hSem, INFINITE);

        g_dwProtectedData = 5;
        dwTemp = g_dwProtectedData;

        // возвращаем новое значение
        return(dwTemp);
    }

    __finally {
        // 3. Даем и другим попользоваться защищенными данными
        ReleaseSemaphore(g_hSem, 1, NULL);
        return(103);
    }

    // продолжаем что-то делать - этот код
    // никогда не выполняется
    dwTemp = 9;
```

```

        return(dwTemp);
    }

```

Блок *try* в *Funcenstein4* пытается вернуть значение переменной *dwTemp* (5) функции, вызвавшей *Funcenstein4*. Как мы уже отметили при обсуждении *Funcenstein2*, попытка преждевременного возврата из блока *try* приводит к генерации кода, который записывает возвращаемое значение во временную переменную, созданную компилятором. Затем выполняется код в блоке *finally*. Кстати, в этом варианте *Funcenstein2* я добавил в блок *finally* оператор *return*. Вопрос: что вернет *Funcenstein4* — 5 или 103? Ответ: 103, так как оператор *return* в блоке *finally* приведет к записи значения 103 в ту же временную переменную, в которую занесено значение 5. По завершении блока *finally* текущее значение временной переменной (103) возвращается функции, вызвавшей *Funcenstein4*.

Итак, обработчики завершения, весьма эффективные при преждевременном выходе из блока *try*, могут дать нежелательные результаты именно потому, что предотвращают досрочный выход из блока *try*. Лучше всего избегать любых операторов, способных вызвать преждевременный выход из блока *try* обработчика завершения. А в идеале — удалить все операторы *return*, *continue*, *break*, *goto* (и им подобные) как из блоков *try*, так и из блоков *finally*. Тогда компилятор сгенерирует код и более компактный (перехватывать преждевременные выходы из блоков *try* не понадобится), и более быстрый (на локальную раскрутку потребуется меньше машинных команд). Да и читать Ваш код будет гораздо легче.

Funcarama1

Мы уже далеко продвинулись в рассмотрении базового синтаксиса и семантики обработчиков завершения. Теперь поговорим о том, как обработчики завершения упрощают более сложные задачи программирования. Взгляните на функцию, в которой не используются преимущества обработки завершения:

```

BOOL Funcarama1() {
    HANDLE hFile = INVALID_HANDLE_VALUE;
    PVOID pvBuf = NULL;
    DWORD dwNumBytesRead;
    BOOL fOk;

    hFile = CreateFile("SOMEDATA.DAT", GENERIC_READ,
                      FILE_SHARE_READ, NULL, OPEN_EXISTING, 0, NULL);
    if (hFile == INVALID_HANDLE_VALUE) {
        return(FALSE);
    }

    pvBuf = VirtualAlloc(NULL, 1024, MEM_COMMIT, PAGE_READWRITE);
    if (pvBuf == NULL) {
        CloseHandle(hFile);
        return(FALSE);
    }

    fOk = ReadFile(hFile, pvBuf, 1024, &dwNumBytesRead, NULL);
    if (!fOk || (dwNumBytesRead == 0)) {
        VirtualFree(pvBuf, MEM_RELEASE | MEM_DECOMMIT);
        CloseHandle(hFile);
    }
}

```

см. след. стр.

```
        return(FALSE);
    }

    // что-то делаем с данными
    :
    // очистка всех ресурсов
    VirtualFree(pvBuf, MEM_RELEASE | MEM_DECOMMIT);
    CloseHandle(hFile);
    return(TRUE);
}
```

Проверки ошибок в функции *Funcarama1* затрудняют чтение ее текста, что усложняет ее понимание, сопровождение и модификацию.

Funcarama2

Конечно, можно переписать *Funcarama1* так, чтобы она стала яснее:

```
BOOL Funcarama2() {
    HANDLE hFile = INVALID_HANDLE_VALUE;
    PVOID pvBuf = NULL;
    DWORD dwNumBytesRead;
    BOOL fOk, fSuccess = FALSE;

    hFile = CreateFile("SOMEDATA.DAT", GENERIC_READ,
                      FILE_SHARE_READ, NULL, OPEN_EXISTING, 0, NULL);

    if (hFile != INVALID_HANDLE_VALUE) {
        pvBuf = VirtualAlloc(NULL, 1024, MEM_COMMIT, PAGE_READWRITE);
        if (pvBuf != NULL) {
            fOk = ReadFile(hFile, pvBuf, 1024, &dwNumBytesRead, NULL);
            if (fOk && (dwNumBytesRead != 0)) {
                // что-то делаем с данными
                :
                fSuccess = TRUE;
            }
        }
        VirtualFree(pvBuf, MEM_RELEASE | MEM_DECOMMIT);
    }
    CloseHandle(hFile);
    return(fSuccess);
}
```

Funcarama2 легче для понимания, но по-прежнему трудна для модификации и сопровождения. Кроме того, приходится делать слишком много отступов по мере добавления новых условных операторов; после такой переделки Вы того и гляди начнете писать код на правом краю экрана и переносить операторы на другую строку через каждые пять символов!

Funcarama3

Перепишем-ка еще раз первый вариант (*Funcarama1*), задействовав преимущества обработки завершения:

```
BOOL Funcarama3() {
    // Внимание: инициализируйте все переменные, предполагая худшее
```

```

HANDLE hFile = INVALID_HANDLE_VALUE;
PVOID pvBuf = NULL;

__try {
    DWORD dwNumBytesRead;
    BOOL fOk;

    hFile = CreateFile("SOMEDATA.DAT", GENERIC_READ,
                       FILE_SHARE_READ, NULL, OPEN_EXISTING, 0, NULL);
    if (hFile == INVALID_HANDLE_VALUE) {
        return(FALSE);
    }

    pvBuf = VirtualAlloc(NULL, 1024, MEM_COMMIT, PAGE_READWRITE);
    if (pvBuf == NULL) {
        return(FALSE);
    }

    fOk = ReadFile(hFile, pvBuf, 1024, &dwNumBytesRead, NULL);
    if (!fOk || (dwNumBytesRead != 1024)) {
        return(FALSE);
    }

    // что-то делаем с данными
    :
}

__finally {
    // очистка всех ресурсов
    if (pvBuf != NULL)
        VirtualFree(pvBuf, MEM_RELEASE | MEM_DECOMMIT);
    if (hFile != INVALID_HANDLE_VALUE)
        CloseHandle(hFile);
}
// продолжаем что-то делать
return(TRUE);
}

```

Главное достоинство *Funcarama3* в том, что весь код, отвечающий за очистку, собран в одном месте — в блоке *finally*. Если понадобится включить что-то в эту функцию, то для очистки мы просто добавим одну-единственную строку в блок *finally* — возвращаться к каждому месту возможного возникновения ошибки и вставлять в него строку для очистки не нужно.

Funcarama4: последний рубеж

Настоящая проблема в *Funcarama3* — расплата за изящество. Я уже говорил: избегайте по возможности операторов *return* внутри блока *try*.

Чтобы облегчить последнюю задачу, Microsoft ввела еще одно ключевое слово в свой компилятор C++: *_leave*. Вот новая версия (*Funcarama4*), построенная на применении нового ключевого слова:

```

BOOL Funcarama4() {
    // Внимание: инициализируйте все переменные, предполагая худшее

```

см. след. стр.

```
HANDLE hFile = INVALID_HANDLE_VALUE;
PVOID pvBuf = NULL;

// предполагаем, что выполнение функции будет неудачным
BOOL fFunctionOk = FALSE;

__try {
    DWORD dwNumBytesRead;
    BOOL fOk;

    hFile = CreateFile("SOMEDATA.DAT", GENERIC_READ,
                      FILE_SHARE_READ, NULL, OPEN_EXISTING, 0, NULL);
    if (hFile == INVALID_HANDLE_VALUE) {
        __leave;
    }

    pvBuf = VirtualAlloc(NULL, 1024, MEM_COMMIT, PAGE_READWRITE);

    if (pvBuf == NULL) {
        __leave;
    }

    fOk = ReadFile(hFile, pvBuf, 1024, &dwNumBytesRead, NULL);
    if (!fOk || (dwNumBytesRead == 0)) {
        __leave;
    }

    // что-то делаем с данными
    :
    // функция выполнена успешно
    fFunctionOk = TRUE;
}

__finally {
    // очистка всех ресурсов
    if (pvBuf != NULL)
        VirtualFree(pvBuf, MEM_RELEASE | MEM_DECOMMIT);
    if (hFile != INVALID_HANDLE_VALUE)
        CloseHandle(hFile);
}
// продолжаем что-то делать
return(fFunctionOk);
}
```

Ключевое слово `__leave` в блоке `try` вызывает переход в конец этого блока. Можете рассматривать это как переход на закрывающую фигурную скобку блока `try`. И никаких неприятностей это не сулит, потому что выход из блока `try` и вход в блок `finally` происходит естественным образом. Правда, нужно ввести дополнительную булаву переменную `fFunctionOk`, сообщающую о завершении функции: удачно оно или нет. Но это дает минимальные издержки.

Разрабатывая функции, использующие обработчики завершения именно так, инициализируйте все описатели ресурсов недопустимыми значениями перед входом в блок `try`. Тогда в блоке `finally` Вы проверите, какие ресурсы выделены успешно, и узнаете тем самым, какие из них следует потом освободить. Другой распространенный

метод отслеживания ресурсов, подлежащих освобождению, — установка флага при успешном выделении ресурса. Код *finally* проверяет состояние флага и таким образом определяет, надо ли освобождать ресурс.

И еще о блоке *finally*

Пока нам с Вами удалось четко выделить только два сценария, которые приводят к выполнению блока *finally*:

- нормальная передача управления от блока *try* блоку *finally*;
- локальная раскрутка — преждевременный выход из блока *try* (из-за операторов *goto*, *longjmp*, *continue*, *break*, *return* и т. д.), вызывающий принудительную передачу управления блоку *finally*.

Третий сценарий — *глобальная раскрутка* (global unwind) — протекает не столь выраженно. Вспомним *Funcfurter1*. Ее блок *try* содержал вызов функции *Funcinator*. При неверном доступе к памяти в *Funcinator* глобальная раскрутка приводила к выполнению блока *finally* в *Funcfurter1*. Но подробнее о глобальной раскрутке мы поговорим в следующей главе.

Выполнение кода в блоке *finally* всегда начинается в результате возникновения одной из этих трех ситуаций. Чтобы определить, какая из них вызвала выполнение блока *finally*, вызовите встраиваемую функцию¹ *AbnormalTermination*:

```
BOOL AbnormalTermination();
```

Ее можно вызвать только из блока *finally*; она возвращает булево значение, которое сообщает, был ли преждевременный выход из блока *try*, связанного с данным блоком *finally*. Иначе говоря, если управление естественным образом передано из *try* в *finally*, *AbnormalTermination* возвращает FALSE. А если выход был преждевременным — обычно либо из-за локальной раскрутки, вызванной оператором *goto*, *return*, *break* или *continue*, либо из-за глобальной раскрутки, вызванной нарушением доступа к памяти, — то вызов *AbnormalTermination* дает TRUE. Но, когда она возвращает TRUE, различить, вызвано выполнение блока *finally* глобальной или локальной раскруткой, нельзя. Впрочем, это не проблема, так как Вы должны избегать кода, приводящего к локальной раскрутке.

Funcfurter2

Следующий фрагмент демонстрирует использование встраиваемой функции *AbnormalTermination*:

```
DWORD Funcfurter2() {
    DWORD dwTemp;

    // 1. Что-то делаем здесь
    :
```

см. след. стр.

¹ Встраиваемая функция (intrinsic function) — особая функция, распознаваемая компилятором. Вместо генерации вызова такой функции он подставляет в точке вызова ее код. Примером встраиваемой функции является *memset* (если указан ключ компилятора /Oi). Встречая вызов *memset*, компилятор подставляет ее код непосредственно в вызывающую функцию. Обычно это ускоряет работу программы ценой увеличения ее размера. Функция *AbnormalTermination* отличается от *memset* тем, что существует только во встраиваемом варианте. Ее нет ни в одной библиотеке С.

```
--try {
    // 2. Запрашиваем разрешение на доступ
    // к защищенным данным, а затем используем их
    WaitForSingleObject(g_hSem, INFINITE);

    dwTemp = Funcinator(g_dwProtectedData);
}

--finally {
    // 3. Даем и другим попользоваться защищенными данными
    ReleaseSemaphore(g_hSem, 1, NULL);

    if (!AbnormalTermination()) {
        // в блоке try не было ошибок - управление
        // передано в блок finally естественным образом
        :
    } else {
        // что-то вызвало исключение, и, так как в блоке try
        // нет кода, который мог бы вызвать преждевременный
        // выход, блок finally выполняется из-за глобальной
        // раскрутки

        // если бы в блоке try был оператор goto, мы бы
        // не узнали, как попали сюда
        :
    }
}
// 4. Продолжаем что-то делать
return(dwTemp);
}
```

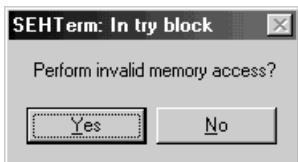
Теперь Вы знаете, как создавать обработчики завершения. Вскоре Вы увидите, что они могут быть еще полезнее и важнее, — когда мы дойдем до фильтров и обработчиков исключений (в следующей главе). А пока давайте суммируем причины, по которым следует применять обработчики завершения.

- Упрощается обработка ошибок — очистка гарантируется и проводится в одном месте.
- Улучшается восприятие текста программ.
- Облегчается сопровождение кода.
- Удается добиться минимальных издержек по скорости и размеру кода — при условии правильного применения обработчиков.

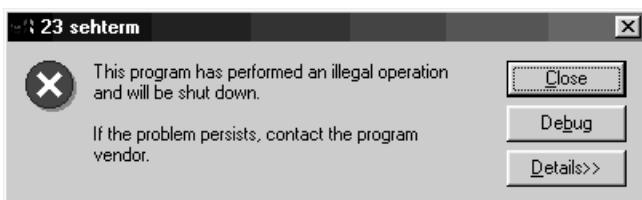
Программа-пример SEHTerm

Эта программа, «23 SEHTerm.exe» (см. листинг на рис. 23-1), демонстрирует обработчики завершения. Файлы исходного кода и ресурсов этой программы находятся в каталоге 23-SEHTerm на компакт-диске, прилагаемом к книге.

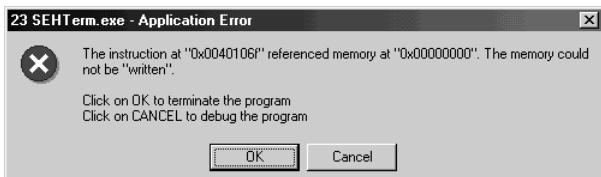
После запуска SEHTerm ее первичный поток входит в блок *try*. Из него открывается следующее окно.



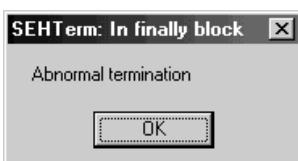
В этом окне предлагается обратиться к памяти по недопустимому адресу. (Большинство приложений не столь тактично — они обращаются по недопустимым адресам, никого не спрашивая.) Давайте обсудим, что случится, если Вы щелкнете кнопку Yes. В этом случае поток попытается записать значение 5 по нулевому адресу памяти. Запись по нулевому адресу всегда вызывает исключение, связанное с нарушением доступа. А когда поток возбуждает такое исключение, Windows 98 выводит окно, показанное ниже.



В Windows 2000 аналогичное окно выглядит иначе.

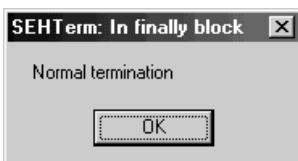


Если Вы теперь щелкнете кнопку Close (в Windows 98) или OK (в Windows 2000), процесс завершится. Однако в исходном коде этой программы присутствует блок *finally*, который будет выполнен до того, как процесс завершится. Из этого блока открывается следующее окно.



Блок *finally* выполняется потому, что происходит ненормальный выход из связанного с ним блока *try*. После закрытия этого окна процесс завершается.

О'кэй, а сейчас снова запустим эту программу. Но на этот раз попробуйте щелкнуть кнопку No, чтобы избежать обращения к памяти по недопустимому адресу. Тогда поток естественным образом перейдет из блока *try* в блок *finally*, откуда будет открыто следующее окно.



Обратите внимание, что на этот раз в окне сообщается о нормальном выходе из блока *try*. Когда Вы закроете это окно, поток выйдет из блока *finally* и покажет последнее окно.



После того как Вы закроете и это окно, процесс нормально завершится, поскольку функция *WinMain* вернет управление. Заметьте, что данное окно не появляется при аварийном завершении процесса.

```
SEHTerm.cpp

/*
Модуль: SEHTerm.cpp
Автор: Copyright (c) 2000, Джейфри Рихтер (Jeffrey Richter)
***** */

#include "..\CmnHdr.h"      /* см. приложение A */
#include <tchar.h>

///////////

int WINAPI _tWinMain(HINSTANCE hinstExe, HINSTANCE, PTSTR pszCmdLine, int) {

    __try {
        int n = MessageBox(NULL, TEXT("Perform invalid memory access?"),
                           TEXT("SEHTerm: In try block"), MB_YESNO);

        if (n == IDYES) {
            * (PBYTE) NULL = 5; // это приводит к нарушению доступа
        }
    } __finally {
        PCTSTR psz = AbnormalTermination()
            ? TEXT("Abnormal termination") : TEXT("Normal termination");
        MessageBox(NULL, psz, TEXT("SEHTerm: In finally block"), MB_OK);
    }

    MessageBox(NULL, TEXT("Normal process termination"),
              TEXT("SEHTerm: After finally block"), MB_OK);

    return(0);
}

/////////// Конец файла ///////////
```

Рис. 23-1. Программа-пример *SEHTerm*

Фильтры и обработчики исключений

Исключение — это событие, которого Вы не ожидали. В хорошо написанной программе не предполагается попыток обращения по неверному адресу или деления на нуль. И все же такие ошибки случаются. За перехват попыток обращения по неверному адресу и деления на нуль отвечает центральный процессор, возбуждающий исключения в ответ на эти ошибки. Исключение, возбужденное процессором, называется *аппаратным* (*hardware exception*). Далее мы увидим, что операционная система и прикладные программы способны возбуждать собственные исключения — *программные* (*software exceptions*).

При возникновении аппаратного или программного исключения операционная система дает Вашему приложению шанс определить его тип и самостоятельно обработать. Синтаксис обработчика исключений таков:

```
_try {  
    // защищенный блок  
    :  
}  
_except (фильтр исключений) {  
    // обработчик исключений  
    :  
}
```

Обратите внимание на ключевое слово *_except*. За блоком *try* всегда должен следовать либо блок *finally*, либо блок *except*. Для данного блока *try* нельзя указать одновременно и блок *finally*, и блок *except*; к тому же за *try* не может следовать несколько блоков *finally* или *except*. Однако *try-finally* можно вложить в *try-except*, и наоборот.

Примеры использования фильтров и обработчиков исключений

В отличие от обработчиков завершения (рассмотренных в предыдущей главе), фильтры и обработчики исключений выполняются непосредственно операционной системой — нагрузка на компилятор при этом минимальна. В следующих разделах я расскажу, как обычно выполняются блоки *try-except*, как и когда операционная система проверяет фильтры исключений и в каких случаях она выполняет код обработчиков исключений.

Funcmeister1

Вот более конкретный пример блока *try-except*:

```
DWORD Funcmeister1() {
    DWORD dwTemp;

    // 1. Что-то делаем здесь
    :

    __try {
        // 2. Выполняем какую-то операцию
        dwTemp = 0;
    }
    __except (EXCEPTION_EXECUTE_HANDLER) {
        // обрабатываем исключение; этот код никогда не выполняется
        :
    }

    // 3. Продолжаем что-то делать
    return(dwTemp);
}
```

В блоке *try* функции *Funcmeister1* мы просто присваиваем 0 переменной *dwTemp*. Такая операция не приведет к исключению, и поэтому код в блоке *except* никогда не выполняется. Обратите внимание на такую особенность: конструкция *try-finally* ведет себя иначе. После того как переменной *dwTemp* присваивается 0, следующим исполнением оператором оказывается *return*.

Хотя ставить операторы *return*, *goto*, *continue* и *break* в блоке *try* обработчика завершения настоятельно не рекомендуется, их применение в этом блоке не приводит к снижению быстродействия кода или к увеличению его размера. Использование этих операторов в блоке *try*, связанном с блоком *except*, не вызовет таких неприятностей, как локальная раскрутка.

Funcmeister2

Попробуем модифицировать нашу функцию и посмотрим, что будет:

```
DWORD Funcmeister2() {
    DWORD dwTemp = 0;

    // 1. Что-то делаем здесь
    :

    __try {
        // 2. Выполняем какую-то операцию
        dwTemp = 5 / dwTemp; // генерирует исключение
        dwTemp += 10;       // никогда не выполняется
    }
    __except ( /* 3. Проверяем фильтр */ EXCEPTION_EXECUTE_HANDLER ) {
        // 4. Обрабатываем исключение
        MessageBeep(0);
        :
    }

    // 5. Продолжаем что-то делать
    return(dwTemp);
}
```

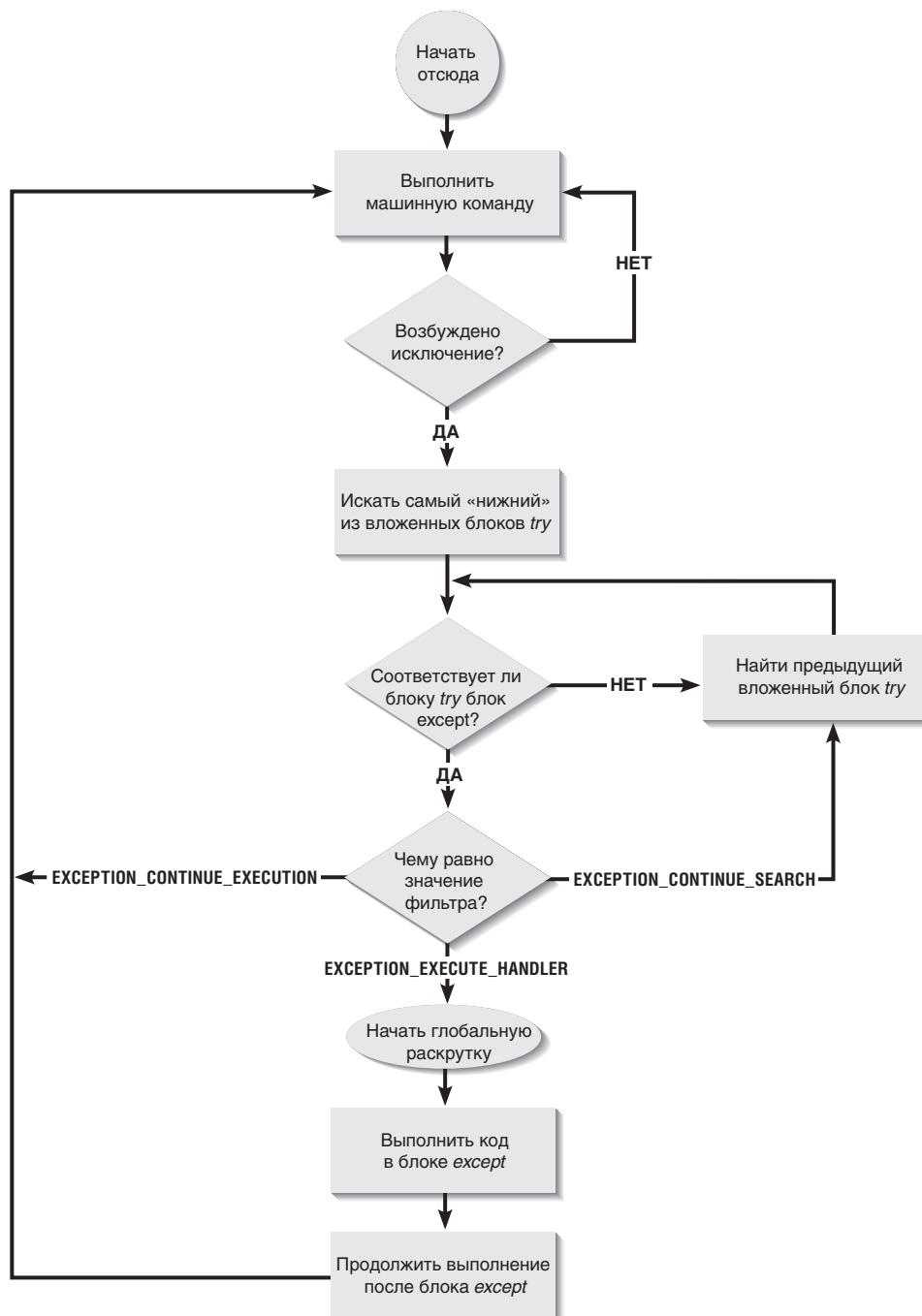


Рис. 24-1. Так система обрабатывает исключения

Инструкция внутри блока `try` функции `Funcmeister2` пытается поделить 5 на 0. Перехватив это событие, процессор возбуждает аппаратное исключение. Тогда операционная система ищет начало блока `except` и проверяет выражение, указанное в качестве фильтра исключений; оно должно дать один из трех идентификаторов, определенных в заголовочном Windows-файле `Excpt.h`.

Идентификатор	Значение
EXCEPTION_EXECUTE_HANDLER	1
EXCEPTION_CONTINUE_SEARCH	0
EXCEPTION_CONTINUE_EXECUTION	-1

Далее мы обсудим, как эти идентификаторы изменяют выполнение потока. Читая следующие разделы, посмотрите на блок-схему на рис. 24-1, которая иллюстрирует операции, выполняемые системой после генерации исключения.

EXCEPTION_EXECUTE_HANDLER

Фильтр исключений в *Funcmeister2* определен как EXCEPTION_EXECUTE_HANDLER. Это значение сообщает системе в основном вот что: «Я вижу это исключение; так и знал, что оно где-нибудь произойдет; у меня есть код для его обработки, и я хочу его сейчас выполнить.» В этот момент система проводит глобальную раскрутку (о ней — немного позже), а затем управление передается коду внутри блока *except* (коду обработчика исключений). После его выполнения система считает исключение обработанным и разрешает программе продолжить работу. Этот механизм позволяет Windows-приложениям перехватывать ошибки, обрабатывать их и продолжать выполнение — пользователь даже не узнает, что была какая-то ошибка.

Но вот откуда возобновится выполнение? Поразмыслив, можно представить несколько вариантов.

Первый вариант. Выполнение возобновляется сразу за строкой, возбудившей исключение. Тогда в *Funcmeister2* выполнение продолжилось бы с инструкции, которая прибавляет к *dwTemp* число 10. Вроде логично, но на деле в большинстве программ нельзя продолжить корректное выполнение, если одна из предыдущих инструкций вызвала ошибку.

В нашем случае нормальное выполнение можно продолжить, но *Funcmeister2* в этом смысле не типична. Ваш код скорее всего структурирован так, что инструкции, следующие за той, где произошло исключение, ожидают от нее корректное значение. Например, у Вас может быть функция, выделяющая блок памяти; тогда для операций с ним, несомненно, предусмотрена целая серия инструкций. Если блок памяти выделить не удастся, все они потерпят неудачу, и программа повторно вызовет исключение.

Вот еще пример того, почему выполнение нельзя продолжить сразу после команды, возбудившей исключение. Заменим оператор языка C, дающий исключение в *Funcmeister2*, строкой:

```
malloc(5 / dwTemp);
```

Компилятор генерирует для нее машинные команды, которые выполняют деление, результат помещают в стек и вызывают *malloc*. Если попытка деления привела к ошибке, дальнейшее (корректное) выполнение кода невозможно. Система должна поместить что-то в стек, иначе он будет разрушен.

К счастью, Microsoft не дает нам шанса возобновить выполнение со строки, расположенной вслед за возбудившей исключение. Это спасает нас от только что описанных потенциальных проблем.

Второй вариант. Выполнение возобновляется с той же команды, которая возбудила исключение. Этот вариант довольно интересен. Допустим, в блоке *except* присутствует оператор:

```
dwTemp = 2;
```

Тогда Вы вполне могли бы возобновить выполнение с возбудившей исключение команды. На этот раз Вы поделили бы 5 на 2, и программа спокойно продолжила бы свою работу. Иначе говоря, Вы что-то меняете и заставляете систему повторить выполнение команды, возбудившей исключение. Но, применяя такой прием, нужно иметь в виду некоторые тонкости (о них — чуть позже).

Третий, и последний, вариант — приложение возобновляет выполнение с инструкции, следующей за блоком *except*. Именно так и происходит, когда фильтр исключений определен как EXCEPTION_EXECUTE_HANDLER. По окончании выполнения кода в блоке *except* управление передается на первую строку за этим блоком.

Некоторые полезные примеры

Допустим, Вы хотите создать отказоустойчивое приложение, которое должно работать 24 часа в сутки и 7 дней в неделю. В наше время, когда программное обеспечение настолько усложнилось и подвержено влиянию множества непредсказуемых факторов, мне кажется, что без SEH просто нельзя создать действительно надежное приложение. Возьмем элементарный пример: функцию *strcpy* из библиотеки С:

```
char* strcpy(
    char* strDestination,
    const char* strSource);
```

Крошечная, давно известная и очень простая функция, да? Разве она может вызвать завершение процесса? Ну, если в каком-нибудь из параметров будет передан NULL (или любой другой недопустимый адрес), *strcpy* приведет к нарушению доступа, и весь процесс будет закрыт.

Создание абсолютно надежной функции *strcpy* возможно только при использовании SEH:

```
char* RobustStrCpy(char* strDestination, const char* strSource) {

    __try {
        strcpy(strDestination, strSource);
    }
    __except (EXCEPTION_EXECUTE_HANDLER) {
        // здесь ничего не делаем
    }

    return(strDestination);
}
```

Все, что делает эта функция, — помещает вызов *strcpy* в SEH-фрейм. Если вызов *strcpy* проходит успешно, *RobustStrCpy* просто возвращает управление. Если же *strcpy* генерирует нарушение доступа, фильтр исключений возвращает значение EXCEPTION_EXECUTE_HANDLER, которое заставляет поток выполнить код обработчика. В функции *RobustStrCpy* обработчик не делает ровным счетом ничего, и опять *RobustStrCpy* просто возвращает управление. Но она никогда не приведет к аварийному завершению процесса!

Рассмотрим другой пример. Вот функция, которая сообщает число отделенных пробелами лексем в строке.

```
int RobustHowManyToken(const char* str) {  
  
    int nHowManyTokens = -1; // значение, равное -1, сообщает о неудаче  
    char* strTemp = NULL; // предполагаем худшее  
  
    __try {  
  
        // создаем временный буфер  
        strTemp = (char*) malloc(strlen(str) + 1);  
  
        // копируем исходную строку во временный буфер  
        strcpy(strTemp, str);  
  
        // получаем первую лексему  
        char* pszToken = strtok(strTemp, " ");  
  
        // перечисляем все лексемы  
        for (; pszToken != NULL; pszToken = strtok(NULL, " "))  
            nHowManyTokens++;  
  
        nHowManyTokens++; // добавляем 1, так как мы начали с -1  
    }  
    __except (EXCEPTION_EXECUTE_HANDLER) {  
        // здесь ничего не делаем  
    }  
  
    // удаляем временный буфер (гарантированная операция)  
    free(strTemp);  
  
    return(nHowManyTokens);  
}
```

Эта функция создает временный буфер и копирует в него строку. Затем, вызывая библиотечную функцию *strtok*, она разбирает строку на отдельные лексемы. Временный буфер необходим из-за того, что *strtok* модифицирует анализируемую строку.

Благодаря SEH эта обманчиво простая функция справляется с любыми неожиданностями. Давайте посмотрим, как она работает в некоторых ситуациях.

Во-первых, если ей передается NULL (или любой другой недопустимый адрес), переменная *nHowManyTokens* сохраняет исходное значение –1. Вызов *strlen* внутри блока *try* приводит к нарушению доступа. Тогда управление передается фильтру исключений, а от него — блоку *except*, который ничего не делает. После блока *except* вызывается *free*, чтобы удалить временный буфер в памяти. Однако он не был создан, и в данной ситуации мы вызываем *free* с передачей ей NULL. Стандарт ANSI C допускает вызов *free* с передачей NULL, в каком случае эта функция просто возвращает управление, так что ошибки здесь нет. В итоге *RobustHowManyToken* возвращает значение –1, сообщая о неудаче, и аварийного завершения процесса не происходит.

Во-вторых, если функция получает корректный адрес, но вызов *malloc* (внутри блока *try*) заканчивается неудачно и дает NULL, то обращение к *strcpy* опять приводит к нарушению доступа. Вновь активизируется фильтр исключений, выполняется блок *except* (который ничего не делает), вызывается *free* с передачей NULL (из-за чего она тоже ничего не делает), и *RobustHowManyToken* возвращает –1, сообщая о неудаче. Аварийного завершения процесса не происходит.

Наконец, допустим, что функции передан корректный адрес и вызов *malloc* прошел успешно. Тогда преуспеет и остальной код, а в переменную *nHowManyTokens* будет записано число лексем в строке. В этом случае выражение в фильтре исключений (в конце блока *try*) не оценивается, код в блоке *except* не выполняется, временный буфер нормально удаляется, и *nHowManyTokens* сообщает количество лексем в строке.

Функция *RobustHowManyToken* демонстрирует, как обеспечить гарантированную очистку ресурса, не прибегая к *try-finally*. Также гарантируется выполнение любого кода, расположенного за обработчиком исключения (если, конечно, функция не возвращает управление из блока *try*, но таких вещей Вы должны избегать).

А теперь рассмотрим последний, особенно полезный пример использования SEH. Вот функция, которая дублирует блок памяти:

```
PBYTE RobustMemDup(PBYTE pbSrc, size_t cb) {
    PBYTE pbDup = NULL; // заранее предполагаем неудачу

    __try {
        // создаем буфер для дублированного блока памяти
        pbDup = (PBYTE) malloc(cb);

        memcpy(pbDup, pbSrc, cb);
    }
    __except (EXCEPTION_EXECUTE_HANDLER) {
        free(pbDup);
        pbDup = NULL;
    }
    return(pbDup);
}
```

Эта функция создает буфер в памяти и копирует в него байты из исходного блока. Затем она возвращает адрес этого дубликата (или NULL, если вызов закончился неудачно). Предполагается, что буфер освобождается вызывающей функцией — когда необходимость в нем отпадает. Это первый пример, где в блоке *except* понадобится какой-то код. Давайте проанализируем работу этой функции в различных ситуациях.

- Если в параметре *pbSrc* передается некорректный адрес или если вызов *malloc* завершается неудачно (и дает NULL), *memcpy* возбуждает нарушение доступа. А это приводит к выполнению фильтра, который передает управление блоку *except*. Код в блоке *except* освобождает буфер памяти и устанавливает *pbDup* в NULL, чтобы вызвавший эту функцию поток узнал о ее неудачном завершении. (Не забудьте, что стандарт ANSI C допускает передачу NULL функции *free*.)
- Если в параметре *pbSrc* передается корректный адрес и вызов *malloc* проходит успешно, функция возвращает адрес только что созданного блока памяти.

Глобальная раскрутка

Когда фильтр исключений возвращает EXCEPTION_EXECUTE_HANDLER, системе приходится проводить глобальную раскрутку. Она приводит к продолжению обработки всех незавершенных блоков *try-finally*, выполнение которых началось вслед за блоком *try-except*, обрабатывающим данное исключение. Блок-схема на рис. 24-2 поясняет, как система осуществляет глобальную раскрутку. Посматривайте на эту схему, когда будете читать мои пояснения к следующему примеру.

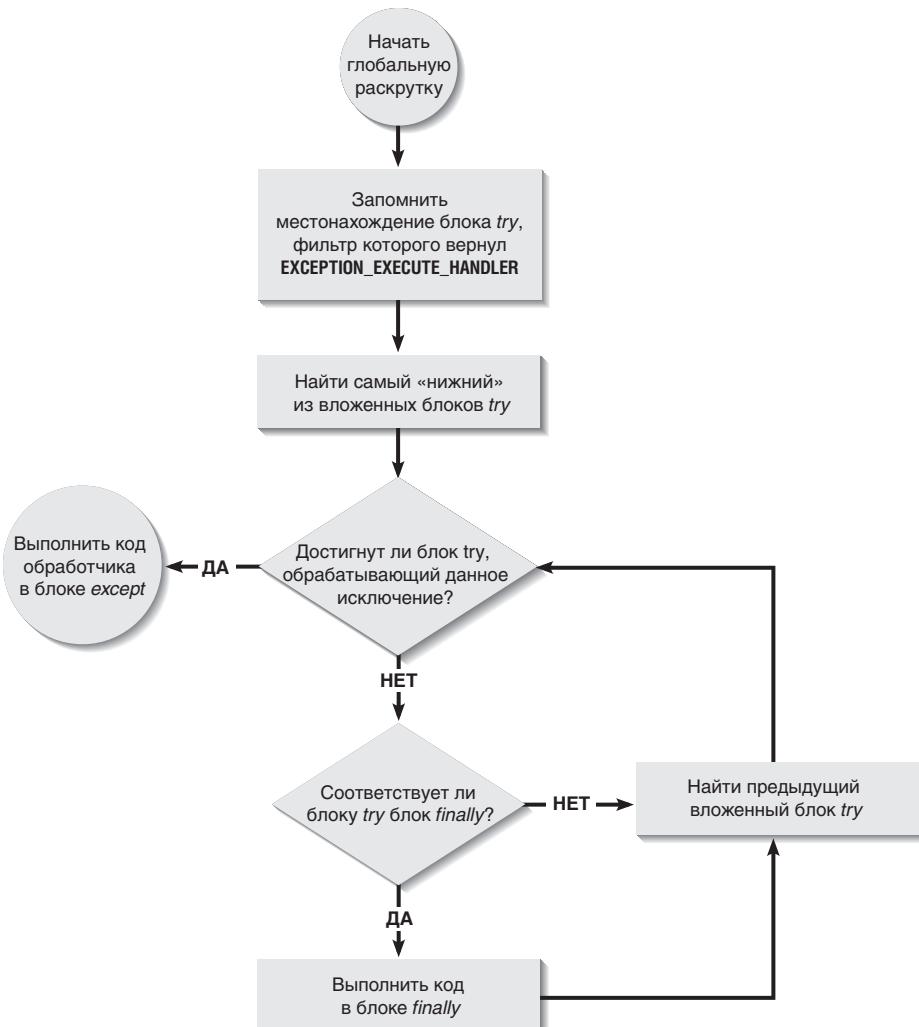


Рис. 24-2. Так система проводит глобальную раскрутку

```

void Func0Stimpy1() {

    // 1. Что-то делаем здесь
    :
    __try {
        // 2. Вызываем другую функцию
        Func0Ren1();

        // этот код никогда не выполняется
    }
    __except ( /* 6. Проверяем фильтр исключений */ EXCEPTION_EXECUTE_HANDLER ) {
        // 8. После раскрутки выполняется этот обработчик
        MessageBox(...);
    }
    // 9. Исключение обработано - продолжаем выполнение
    :
  
```

```

}

void FuncORen1() {
    DWORD dwTemp = 0;

    // 3. Что-то делаем здесь
    :

    __try {
        // 4. Запрашиваем разрешение на доступ к защищенным данным
        WaitForSingleObject(g_hSem, INFINITE);

        // 5. Изменяем данные, и здесь генерируется исключение
        g_dwProtectedData = 5 / dwTemp;
    }
    __finally {
        // 7. Происходит глобальная раскрутка, так как
        // фильтр возвращает EXCEPTION_EXECUTE_HANDLER

        // Даём и другим попользоваться защищенными данными
        ReleaseSemaphore(g_hSem, 1, NULL);
    }

    // сюда мы никогда не попадем
    :
}

```

FuncOStimpy1 и *FuncORen1* иллюстрируют самые запутанные аспекты структурной обработки исключений. Номера в начале комментариев показывают порядок выполнения, в котором сходу не разберешься, но возьмемся за руки и пойдем вместе.

FuncOStimpy1 начинает выполнение со входа в свой блок *try* и вызова *FuncORen1*. Последняя тоже начинает со входа в свой блок *try* и ждет освобождения семафора. Завладев им, она пытается изменить значение глобальной переменной *g_dwProtectedData*. Деление на нуль возбуждает исключение. Система, перехватив управление, ищет блок *try*, которому соответствует блок *except*. Поскольку блоку *try* функции *FuncORen1* соответствует блок *finally*, система продолжает поиск и находит блок *try* в *FuncOStimpy1*, которому как раз и соответствует блок *except*.

Тогда система проверяет значение фильтра исключений в блоке *except* функции *FuncOStimpy1*. Обнаружив, что оно — EXCEPTION_EXECUTE_HANDLER, система начинает глобальную раскрутку с блока *finally* в функции *FuncORen1*. Заметьте: раскрутка происходит до выполнения кода из блока *except* в *FuncOStimpy1*. Осуществляя глобальную раскрутку, система возвращается к последнему незавершенному блоку *try* и ищет теперь блоки *try*, которым соответствуют блоки *finally*. В нашем случае блок *finally* находится в функции *FuncORen1*.

Мощь SEH по-настоящему проявляется, когда система выполняет код *finally* в *FuncORen1*. Из-за его выполнения семафор освобождается, и поэтому другой поток получает возможность продолжить работу. Если бы вызов *ReleaseSemaphore* в блоке *finally* отсутствовал, семафор никогда бы не освободился.

Завершив выполнение блока *finally*, система ищет другие незавершенные блоки *finally*. В нашем примере таких нет. Дойдя до блока *except*, обрабатывающего исключение, система прекращает восходящий проход по цепочке блоков. В этой точке глобальная раскрутка завершается, и система может выполнить код в блоке *except*.

Вот так и работает структурная обработка исключений. Вообще-то, SEH — штука весьма трудная для понимания: в выполнение Вашего кода вмешивается операционная система. Код больше не выполняется последовательно, сверху вниз; система устанавливает свой порядок — сложный, но все же предсказуемый. Поэтому, следуя блок-схемам на рис. 24-1 и 24-2, Вы сможете уверенно применять SEH.

Чтобы лучше разобраться в порядке выполнения кода, посмотрим на происходящее под другим углом зрения. Возвращая EXCEPTION_EXECUTE_HANDLER, фильтр сообщает операционной системе, что регистр указателя команд данного потока должен быть установлен на код внутри блока *except*. Однако этот регистр указывал на код внутри блока *try* функции *FuncOREn1*. А из главы 23 Вы должны помнить, что всякий раз, когда поток выходит из блока *try*, соответствующему блоку *finally*, обязательно выполняется код в этом блоке *finally*. Глобальная раскрутка как раз и является тем механизмом, который гарантирует соблюдение этого правила при любом исключении.

Остановка глобальной раскрутки

Глобальную раскрутку, осуществляющую системой, можно остановить, если в блок *finally* включить оператор *return*. Взгляните:

```
void FuncMonkey() {
    __try {
        FuncFish();
    }
    __except (EXCEPTION_EXECUTE_HANDLER) {
        MessageBeep(0);
    }
    MessageBox(...);
}

void FuncFish() {
    FuncPheasant();
    MessageBox(...);
}

void FuncPheasant() {
    __try {
        strcpy(NULL, NULL);
    }
    __finally {
        return;
    }
}
```

При вызове *strcpy* в блоке *try* функции *FuncPheasant* из-за нарушения доступа к памяти генерируется исключение. Как только это происходит, система начинает просматривать код, пытаясь найти фильтр, способный обработать данное исключение. Обнаружив, что фильтр в *FuncMonkey* готов обработать его, система приступает к глобальной раскрутке. Она начинается с выполнения кода в блоке *finally* функции *FuncPheasant*. Но этот блок содержит оператор *return*. Он заставляет систему прекратить раскрутку, и *FuncPheasant* фактически завершается возвратом в *FuncFish*, которая выводит сообщение на экран. Затем *FuncFish* возвращает управление *FuncMonkey*, и та вызывает *MessageBox*.

Заметьте: код блока *except* в *FuncMonkey* никогда не вызовет *MessageBeep*. Оператор *return* в блоке *finally* функции *FuncPheasant* заставит систему вообще прекратить раскрутку, и поэтому выполнение продолжится так, будто ничего не произошло.

Microsoft намеренно вложила в SEH такую логику. Иногда ведь нужно прекратить раскрутку и продолжить выполнение программы. Хотя в большинстве случаев так все же не делают. А значит, будьте внимательны и избегайте операторов *return* в блоках *finally*.

EXCEPTION_CONTINUE_EXECUTION

Давайте приглядимся к тому, как фильтр исключений получает один из трех идентификаторов, определенных в файле *Excpt.h*. В *Funcmeister2* идентификатор *EXCEPTION_EXECUTE_HANDLER* «зашит» (простоты ради) в код самого фильтра, но Вы могли бы вызывать там функцию, которая определяла бы нужный идентификатор. Взгляните:

```
char g_szBuffer[100];

void FuncInRoosevelt1() {
    int x = 0;
    char *pchBuffer = NULL;

    __try {
        *pchBuffer = 'J';
        x = 5 / x;
    }
    __except (OilFilter1(&pchBuffer)) {
        MessageBox(NULL, "An exception occurred", NULL, MB_OK);
    }
    MessageBox(NULL, "Function completed", NULL, MB_OK);
}

LONG OilFilter1(char **ppchBuffer) {
    if (*ppchBuffer == NULL) {
        *ppchBuffer = g_szBuffer;
        return(EXCEPTION_CONTINUE_EXECUTION);
    }
    return(EXCEPTION_EXECUTE_HANDLER);
}
```

В первый раз проблема возникает, когда мы пытаемся поместить *J* в буфер, на который указывает *pchBuffer*. К сожалению, мы не определили *pchBuffer* как указатель на наш глобальный буфер *g_szBuffer* — вместо этого он указывает на NULL. Процессор генерирует исключение и вычисляет выражение в фильтре исключений в блоке *except*, связанном с блоком *try*, в котором и произошло исключение. В блоке *except* адрес переменной *pchBuffer* передается функции *OilFilter1*.

Получая управление, *OilFilter1* проверяет, не равен ли **ppchBuffer* значению NULL, и, если да, устанавливает его так, чтобы он указывал на глобальный буфер *g_szBuffer*. Тогда фильтр возвращает *EXCEPTION_CONTINUE_EXECUTION*. Обнаружив такое значение выражения в фильтре, система возвращается к инструкции, вызвавшей исключение, и пытается выполнить ее снова. На этот раз все проходит успешно, и *J* будет записана в первый байт буфера *g_szBuffer*.

Когда выполнение кода продолжится, мы опять столкнемся с проблемой в блоке *try* — теперь это деление на нуль. И вновь система вычислит выражение фильтра ис-

ключений. На этот раз **ppchBuffer* не равен NULL, и поэтому *OilFilter1* вернет EXCEPTION_EXECUTE_HANDLER, что подскажет системе выполнить код в блоке *except*, и на экране появится окно с сообщением об исключении.

Как видите, внутри фильтра исключений можно проделать массу всякой работы. Но, разумеется, в итоге фильтр должен вернуть один из трех идентификаторов.

Будьте осторожны с EXCEPTION_CONTINUE_EXECUTION

Будет ли удачной попытка исправить ситуацию в только что рассмотренной функции и заставить систему продолжить выполнение программы, зависит от типа процессора, от того, как компилятор генерирует машинные команды при трансляции операторов C/C++, и от параметров, заданных компилятору.

Компилятор мог сгенерировать две машинные команды для оператора:

```
*pchBuffer = 'J';
```

которые выглядят так:

```
MOV EAX, [pchBuffer] // адрес помещается в регистр EAX  
MOV [EAX], 'J' // символ J записывается по адресу из регистра EAX
```

Последняя команда и возбудила бы исключение. Фильтр исключений, перехватив его, исправил бы значение *pchBuffer* и указал бы системе повторить эту команду. Но проблема в том, что содержимое регистра не изменится так, чтобы отразить новое значение *pchBuffer*, и поэтому повторение команды снова приведет к исключению. Вот и бесконечный цикл!

Выполнение программы благополучно возобновится, если компилятор оптимизирует код, но может прерваться, если компилятор код не оптимизирует. Обнаружить такой «жучок» очень трудно, и — чтобы определить, откуда он взялся в программе, — придется анализировать ассемблерный текст, сгенерированный для исходного кода. Вывод: будьте крайне осторожны, возвращая EXCEPTION_CONTINUE_EXECUTION из фильтра исключений.

EXCEPTION_CONTINUE_EXECUTION всегда срабатывает лишь в одной ситуации: при передаче памяти зарезервированному региону. О том, как зарезервировать большую область адресного пространства, а потом передавать ей память лишь по мере необходимости, я рассказывал в главе 15. Соответствующий алгоритм демонстрировала программа-пример VMAlloc. На основе механизма SEH то же самое можно было бы реализовать гораздо эффективнее (и не пришлось бы все время вызывать функцию *VirtualAlloc*).

В главе 16 мы говорили о стеках потоков. В частности, я показал, как система резервирует для стека потока регион адресного пространства размером 1 Мб и как она автоматически передает ему новую память по мере разрастания стека. С этой целью система создает SEH-фрейм. Когда поток пытается задействовать несуществующую часть стека, генерируется исключение. Системный фильтр определяет, что исключение возникло из-за попытки обращения к адресному пространству, зарезервированному под стек, вызывает функцию *VirtualAlloc* для передачи дополнительной памяти стеку потока и возвращает EXCEPTION_CONTINUE_EXECUTION. После этого машинная команда, пытавшаяся обратиться к несуществующей части стека, благополучно выполняется, и поток продолжает свою работу.

Механизмы использования виртуальной памяти в сочетании со структурной обработкой исключений позволяют создавать невероятно «шустрые» приложения. Программа-пример Spreadsheet в следующей главе продемонстрирует, как на основе SEH

эффективно реализовать управление памятью в электронной таблице. Этот код выполняется чрезвычайно быстро.

EXCEPTION_CONTINUE_SEARCH

Приведенные до сих пор примеры были ну просто детскими. Чтобы немного встряхнуться, добавим вызов функции:

```
char g_szBuffer[100];

void FunclinRoosevelt2() {
    char *pchBuffer = NULL;

    __try {
        FuncAtude2(pchBuffer);
    }
    __except (OilFilter2(&pchBuffer)) {
        MessageBox(...);
    }
}

void FuncAtude2(char *sz) {
    *sz = 0;
}

LONG OilFilter2(char **ppchBuffer) {
    if (*ppchBuffer == NULL) {
        *ppchBuffer = g_szBuffer;
        return(EXCEPTION_CONTINUE_EXECUTION);
    }
    return(EXCEPTION_EXECUTE_HANDLER);
}
```

При выполнении *FunclinRoosevelt2* вызывается *FuncAtude2*, которой передается NULL. Последняя приводит к исключению. Как и раньше, система проверяет выражение в фильтре исключений, связанном с последним исполняемым блоком *try*. В нашем примере это блок *try* в *FunclinRoosevelt2*, поэтому для оценки выражения в фильтре исключений система вызывает *OilFilter2* (хотя исключение возникло в *FuncAtude2*).

Замесим ситуацию еще круче, добавив другой блок *try-except*:

```
char g_szBuffer[100];

void FunclinRoosevelt3() {
    char *pchBuffer = NULL;

    __try {
        FuncAtude3(pchBuffer);
    }
    __except (OilFilter3(&pchBuffer)) {
        MessageBox(...);
    }
}

void FuncAtude3(char *sz) {
```

см. след. стр.

```
--try {
    *sz = 0;
}
__except (EXCEPTION_CONTINUE_SEARCH) {
    // этот код никогда не выполняется
    :
}
}

LONG OilFilter3(char **ppchBuffer) {
    if (*ppchBuffer == NULL) {
        *ppchBuffer = g_szBuffer;
        return(EXCEPTION_CONTINUE_EXECUTION);
    }
    return(EXCEPTION_EXECUTE_HANDLER);
}
```

Теперь, когда *FuncAtude3* пытается занести 0 по адресу NULL, по-прежнему возбуждается исключение, но в работу вступает фильтр исключений из *FuncAtude3*. Значение этого очень простого фильтра — EXCEPTION_CONTINUE_SEARCH. Данный идентификатор указывает системе перейти к предыдущему блоку *try*, которому соответствует блок *except*, и обработать его фильтр.

Так как фильтр в *FuncAtude3* дает EXCEPTION_CONTINUE_SEARCH, система переходит к предыдущему блоку *try* (в функции *FunclinRoosevelt3*) и вычисляет его фильтр *OilFilter3*. Обнаружив, что значение *pcchBuffer* равно NULL, *OilFilter3* меняет его так, чтобы оно указывало на глобальный буфер, и сообщает системе возобновить выполнение с инструкции, вызвавшей исключение. Это позволяет выполнить код в блоке *try* функции *FuncAtude3*, но, увы, локальная переменная *sz* в этой функции не изменена, и возникает новое исключение. Опять бесконечный цикл!

Заметьте, я сказал, что система переходит к последнему исполнявшемуся блоку *try*, которому соответствует блок *except*, и проверяет его фильтр. Это значит, что система пропускает при просмотре цепочки блоков любые блоки *try*, которым соответствуют блоки *finally* (а не *except*). Причина этого очевидна: в блоках *finally* нет фильтров исключений, а потому и проверять в них нечего. Если бы в последнем примере *FuncAtude3* содержала вместо *except* блок *finally*, система начала бы проверять фильтры исключений с *OilFilter3* в *FunclinRoosevelt3*.

Дополнительную информацию об EXCEPTION_CONTINUE_SEARCH см. в главе 25.

Функция *GetExceptionCode*

Часто фильтр исключений должен проанализировать ситуацию, прежде чем определить, какое значение ему вернуть. Например, Ваш обработчик может знать, что делать при делении на нуль, но не знать, как обработать нарушение доступа к памяти. Именно поэтому фильтр отвечает за анализ ситуации и возврат соответствующего значения.

Этот фрагмент иллюстрирует метод, позволяющий определять тип исключения:

```
--try {
    x = 0;
    y = 4 / x;
}
```

```

__except ((GetExceptionCode() == EXCEPTION_INT_DIVIDE_BY_ZERO) ?
    EXCEPTION_EXECUTE_HANDLER : EXCEPTION_CONTINUE_SEARCH) {
    // обработка деления на нуль
}

```

Встраиваемая функция *GetExceptionCode* возвращает идентификатор типа исключения:

```
DWORD GetExceptionCode();
```

Ниже приведен список всех предопределенных идентификаторов исключений с пояснением их смысла (информация взята из документации Platform SDK). Эти идентификаторы содержатся в заголовочном файле WinBase.h. Я сгруппировал исключения по категориям.

Исключения, связанные с памятью

- **EXCEPTION_ACCESS_VIOLATION** Поток пытался считать или записать по виртуальному адресу, не имея на то необходимых прав. Это самое распространенное исключение.
- **EXCEPTION_DATATYPE_MISALIGNMENT** Поток пытался считать или записать невыровненные данные на оборудовании, которое не поддерживает автоматическое выравнивание. Например, 16-битные значения должны быть выровнены по двухбайтовым границам, 32-битные — по четырехбайтовым и т. д.
- **EXCEPTION_ARRAY_BOUNDS_EXCEEDED** Поток пытался обратиться к элементу массива, индекс которого выходит за границы массива; при этом оборудование должно поддерживать такой тип контроля.
- **EXCEPTION_IN_PAGE_ERROR** Ошибку страницы нельзя обработать, так как файловая система или драйвер устройства сообщили об ошибке чтения.
- **EXCEPTION_GUARD_PAGE** Поток пытался обратиться к странице памяти с атрибутом защиты PAGE_GUARD. Страница становится доступной, и генерируется данное исключение.
- **EXCEPTION_STACK_OVERFLOW** Стек, отведенный потоку, исчерпан.
- **EXCEPTION_ILLEGAL_INSTRUCTION** Поток выполнил недопустимую инструкцию. Это исключение определяется архитектурой процессора; можно ли перехватить выполнение неверной инструкции, зависит от типа процессора.
- **EXCEPTION_PRIV_INSTRUCTION** Поток пытался выполнить инструкцию, недопустимую в данном режиме работы процессора.

Исключения, связанные с обработкой самих исключений

- **EXCEPTION_INVALID_DISPOSITION** Фильтр исключений вернул значение, отличное от EXCEPTION_EXECUTE_HANDLER, EXCEPTION_CONTINUE_SEARCH или EXCEPTION_CONTINUE_EXECUTION.
- **EXCEPTION_NONCONTINUABLE_EXCEPTION** Фильтр исключений вернул EXCEPTION_CONTINUE_EXECUTION в ответ на невозобновляемое исключение (noncontinuable exception).

Исключения, связанные с отладкой

- **EXCEPTION_BREAKPOINT** Встретилась точка прерывания (останова).
- **EXCEPTION_SINGLE_STEP** Трассировочная ловушка или другой механизм пошагового исполнения команд подал сигнал о выполнении одной команды.
- **EXCEPTION_INVALID_HANDLE** В функцию передан недопустимый описатель.

Исключения, связанные с операциями над целыми числами

- **EXCEPTION_INT_DIVIDE_BY_ZERO** Поток пытался поделить число целого типа на делитель того же типа, равный 0.
- **EXCEPTION_INT_OVERFLOW** Операция над целыми числами вызвала перенос старшего разряда результата.

Исключения, связанные с операциями над вещественными числами

- **EXCEPTION_FLT_DENORMAL_OPERAND** Один из operandов в операции над числами с плавающей точкой (вещественного типа) не нормализован. Ненормализованными являются значения, слишком малые для стандартного представления числа с плавающей точкой.
- **EXCEPTION_FLT_DIVIDE_BY_ZERO** Поток пытался поделить число вещественного типа на делитель того же типа, равный 0.
- **EXCEPTION_FLT_INEXACT_RESULT** Результат операции над числами с плавающей точкой нельзя точно представить в виде десятичной дроби.
- **EXCEPTION_FLT_INVALID_OPERATION** Любое другое исключение, относящееся к операциям над числами с плавающей точкой и не включенное в этот список.
- **EXCEPTION_FLT_OVERFLOW** Порядок результата операции над числами с плавающей точкой превышает максимальную величину для указанного типа данных.
- **EXCEPTION_FLT_STACK_CHECK** Переполнение стека или выход за его нижнюю границу в результате выполнения операции над числами с плавающей точкой.
- **EXCEPTION_FLT_UNDERFLOW** Порядок результата операции над числами с плавающей точкой меньше минимальной величины для указанного типа данных.

Встраиваемую функцию *GetExceptionCode* можно вызвать только из фильтра исключений (между скобками, которые следуют за *_except*) или из обработчика исключений. Скажем, такой код вполне допустим:

```
_try {
    y = 0;
    x = 4 / y;
}

_except {
    ((GetExceptionCode() == EXCEPTION_ACCESS_VIOLATION) ||
```

```
(GetExceptionCode() == EXCEPTION_INT_DIVIDE_BY_ZERO)) ?
EXCEPTION_EXECUTE_HANDLER : EXCEPTION_CONTINUE_SEARCH) {

    switch (GetExceptionCode()) {
        case EXCEPTION_ACCESS_VIOLATION:
            // обработка нарушения доступа к памяти
            :
            break;

        case EXCEPTION_INT_DIVIDE_BY_ZERO:
            // обработка деления целого числа на нуль
            :
            break;
    }
}
```

Однако *GetExceptionCode* нельзя вызывать из функции фильтра исключений. Компилятор помогает вылавливать такие ошибки и обязательно сообщит о таковой, если Вы попытаетесь скомпилировать, например, следующий код:

```
_try {
    y = 0;
    x = 4 / y;
}

_except (CoffeeFilter()) {
    // обработка исключения
    :
}

LONG CoffeeFilter(void) {
    // ошибка при компиляции: недопустимый вызов GetExceptionCode
    return((GetExceptionCode() == EXCEPTION_ACCESS_VIOLATION) ?
        EXCEPTION_EXECUTE_HANDLER : EXCEPTION_CONTINUE_SEARCH);
}
```

Нужного эффекта можно добиться, переписав код так:

```
_try {
    y = 0;
    x = 4 / y;
}

_except (CoffeeFilter(GetExceptionCode())) {
    // обработка исключения
    :
}

LONG CoffeeFilter(DWORD dwExceptionCode) {
    return((dwExceptionCode == EXCEPTION_ACCESS_VIOLATION) ?
        EXCEPTION_EXECUTE_HANDLER : EXCEPTION_CONTINUE_SEARCH);
}
```

Коды исключений формируются по тем же правилам, что и коды ошибок, определенные в файле WinError.h. Каждое значение типа DWORD разбивается на поля, как показано в таблице 24-1.

Биты	31–30	29	28	27–16	15–0
Содержимое:	Код степени «тяжести» (severity)	Кем определен — Microsoft или пользователем	Зарезервирован	Код подсистемы (facility code)	Код исключения
Значение:	0 = успех 1 = информация 2 = предупреждение 3 = ошибка	0 = Microsoft 1 = пользователь	Должен быть 0 (см. таблицу ниже)	Определяется Microsoft	Определяется Microsoft или пользователем

Таблица 24-1. Поля кода ошибки

На сегодняшний день определены такие коды подсистемы.

Код подсистемы	Значение	Код подсистемы	Значение
FACILITY_NULL	0	FACILITY_CONTROL	10
FACILITY_RPC	1	FACILITY_CERT	11
FACILITY_DISPATCH	2	FACILITY_INTERNET	12
FACILITY_STORAGE	3	FACILITY_MEDIASERVER	13
FACILITY_ITF	4	FACILITY MSMQ	14
FACILITY_WIN32	7	FACILITY_SETUPAPI	15
FACILITY_WINDOWS	8	FACILITY_SCARD	16
FACILITY_SECURITY	9	FACILITY_COMPLUS	17

Разберем на части, например, код исключения EXCEPTION_ACCESS_VIOLATION. Если Вы посмотрите его значение в файле WinBase.h, то увидите, что оно равно 0xC0000005:

C 0 0 0 0 0 0 5 (в шестнадцатеричном виде)
1100 0000 0000 0000 0000 0000 0101 (в двоичном виде)

Биты 30 и 31 установлены в 1, указывая, что нарушение доступа является ошибкой (поток не может продолжить выполнение). Бит 29 равен 0, а это значит, что данный код определен Microsoft. Бит 28 равен 0, так как зарезервирован на будущее. Биты 16–27 равны 0, сообщая код подсистемы FACILITY_NULL (нарушение доступа может произойти в любой подсистеме операционной системы, а не в какой-то одной). Биты 0–15 дают значение 5, которое означает лишь то, что Microsoft присвоила исключению, связанному с нарушением доступа, код 5.

Функция *GetExceptionInformation*

Когда возникает исключение, операционная система заталкивает в стек соответствующего потока структуры EXCEPTION_RECORD, CONTEXT и EXCEPTION_POINTERS.

EXCEPTION_RECORD содержит информацию об исключении, независимую от типа процессора, а CONTEXT — машинно-зависимую информацию об этом исключении. В структуре EXCEPTION_POINTERS всего два элемента — указатели на помещенные в стек структуры EXCEPTION_RECORD и CONTEXT:

```
typedef struct _EXCEPTION_POINTERS {  
    PEXCEPTION_RECORD ExceptionRecord;
```

```
PCONTEXT ContextRecord;
} EXCEPTION_POINTERS, *PEXCEPTION_POINTERS;
```

Чтобы получить эту информацию и использовать ее в программе, вызовите *GetExceptionInformation*:

```
PEXCEPTION_POINTERS GetExceptionInformation();
```

Эта встраиваемая функция возвращает указатель на структуру EXCEPTION_POINTERS.

Самое важное в *GetExceptionInformation* то, что ее можно вызывать только в фильтре исключений и больше нигде, потому что структуры CONTEXT, EXCEPTION_RECORD и EXCEPTION_POINTERS существуют лишь во время обработки фильтра исключений. Когда управление переходит к обработчику исключений, эти данные в стеке разрушаются.

Если Вам нужно получить доступ к информации об исключении из обработчика, сохраните структуру EXCEPTION_RECORD и/или CONTEXT (на которые указывают элементы структуры EXCEPTION_POINTERS) в объявленных Вами переменных. Вот пример сохранения этих структур:

```
void FuncSkunk() {
    // объявляем переменные, которые мы сможем потом использовать
    // для сохранения информации об исключении (если оно произойдет)
    EXCEPTION_RECORD SavedExceptRec;
    CONTEXT SavedContext;
    :
    __try {
        :
    }
    __except {
        SavedExceptRec =
            *(GetExceptionInformation())->ExceptionRecord,
        SavedContext =
            *(GetExceptionInformation())->ContextRecord,
        EXCEPTION_EXECUTE_HANDLER) {

        // мы можем теперь использовать переменные SavedExceptRec
        // и SavedContext в блоке обработчика исключений
        switch (SavedExceptRec.ExceptionCode) {
            :
        }
    }
    :
}
```

В фильтре исключений применяется оператор-запятая (,) — мало кто из программистов знает о нем. Он указывает компилятору, что выражения, отделенные запятыми, следует выполнять слева направо. После вычисления всех выражений возвращается результат последнего из них — крайнего справа.

В *FuncSkunk* сначала вычисляется выражение слева, что приводит к сохранению находящейся в стеке структуры EXCEPTION_RECORD в локальной переменной *SavedExceptRec*. Результат этого выражения является значением *SavedExceptRec*. Но он отбрасывается, и вычисляется выражение, расположенное правее. Это приводит к сохранению размещеннной в стеке структуры CONTEXT в локальной переменной *Saved-*

Context. И снова результат — значение *SavedContext* — отбрасывается, и вычисляется третье выражение. Оно равно *EXCEPTION_EXECUTE_HANDLER* — это и будет результатом всего выражения в скобках.

Так как фильтр возвращает *EXCEPTION_EXECUTE_HANDLER*, выполняется код в блоке *except*. К этому моменту переменные *SavedExceptRec* и *SavedContext* уже инициализированы, и их можно использовать в данном блоке. Важно, чтобы переменные *SavedExceptRec* и *SavedContext* были объявлены вне блока *try*.

Вероятно, Вы уже догадались, что элемент *ExceptionRecord* структуры *EXCEPTION_POINTERS* указывает на структуру *EXCEPTION_RECORD*:

```
typedef struct _EXCEPTION_RECORD {
    DWORD ExceptionCode;
    DWORD ExceptionFlags;
    struct _EXCEPTION_RECORD *ExceptionRecord;
    PVOID ExceptionAddress;
    DWORD NumberParameters;
    ULONG_PTR ExceptionInformation[EXCEPTION_MAXIMUM_PARAMETERS];
} EXCEPTION_RECORD;
```

Структура *EXCEPTION_RECORD* содержит подробную машинно-независимую информацию о последнем исключении. Вот что представляют собой ее элементы.

- *ExceptionCode* — код исключения. Это информация, возвращаемая функцией *GetExceptionCode*.
- *ExceptionFlags* — флаги исключения. На данный момент определено только два значения: 0 (возобновляемое исключение) и *EXCEPTION_NONCONTINUABLE* (невозобновляемое исключение). Любая попытка возобновить работу программы после невозобновляемого исключения генерирует исключение *EXCEPTION_NONCONTINUABLE_EXCEPTION*.
- *ExceptionRecord* — указатель на структуру *EXCEPTION_RECORD*, содержащую информацию о другом необработанном исключении. При обработке одного исключения может возникнуть другое. Например, код внутри фильтра исключений может попытаться выполнить деление на нуль. Когда возникает серия вложенных исключений, записи с информацией о них могут образовывать связанный список. Исключение будет вложенным, если оно генерируется при обработке фильтра. В отсутствие необработанных исключений *ExceptionRecord* равен NULL.
- *ExceptionAddress* — адрес машинной команды, при выполнении которой произошло исключение.
- *NumberParameters* — количество параметров, связанных с исключением (0–15). Это число заполненных элементов в массиве *ExceptionInformation*. Почти для всех исключений значение этого элемента равно 0.
- *ExceptionInformation* — массив дополнительных аргументов, описывающих исключение. Почти для всех исключений элементы этого массива не определены.

Последние два элемента структуры *EXCEPTION_RECORD* сообщают фильтру дополнительную информацию об исключении. Сейчас такую информацию дает только один тип исключений: *EXCEPTION_ACCESS_VIOLATION*. Все остальные дают нулевое значение в элементе *NumberParameters*. Проверив его, Вы узнаете, надо ли просматривать массив *ExceptionInformation*.

При исключении EXCEPTION_ACCESS_VIOLATION элемент *ExceptionInformation[0]* содержит флаг, указывающий тип операции, которая вызвала нарушение доступа. Если его значение равно 0, поток пытался читать недоступные ему данные; 1 — записывать данные по недоступному ему адресу. Элемент *ExceptionInformation[1]* определяет адрес недоступных данных.

Эта структура позволяет писать фильтры исключений, сообщающие значительный объем информации о работе программы. Можно создать, например, такой фильтр:

```

__try {
    :
__except (ExpFltr(GetExceptionInformation()->ExceptionRecord)) {
    :
}

LONG ExpFltr(PEXCEPTION_RECORD pER) {
    char szBuf[300], *p;
    DWORD dwExceptionCode = pER->ExceptionCode;

    sprintf(szBuf, "Code = %x, Address = %p",
            dwExceptionCode, pER->ExceptionAddress);

    // находим конец строки
    p = strchr(szBuf, 0);

    // я использовал оператор switch на тот случай, если Microsoft
    // в будущем добавит информацию для других исключений
    switch (dwExceptionCode) {
        case EXCEPTION_ACCESS_VIOLATION:
            sprintf(p, "Attempt to %s data at address %p",
                    pER->ExceptionInformation[0] ? "write" : "read",
                    pER->ExceptionInformation[1]);
            break;

        default:
            break;
    }
    MessageBox(NULL, szBuf, "Exception", MB_OK | MB_ICONEXCLAMATION);
    return(EXCEPTION_CONTINUE_SEARCH);
}

```

Элемент *ContextRecord* структуры EXCEPTION_POINTERS указывает на структуру CONTEXT (см. главу 7), содержимое которой зависит от типа процессора.

С помощью этой структуры, в основном содержащей по одному элементу для каждого регистра процессора, можно получить дополнительную информацию о возникшем исключении. Увы, это потребует написания машинно-зависимого кода, способного распознавать тип процессора и использовать подходящую для него структуру CONTEXT. При этом Вам придется включить в код набор директив *#ifdef* для разных типов процессоров. Структуры CONTEXT для различных процессоров, поддерживаемых Windows, определены в заголовочном файле WinNT.h.

Программные исключения

До сих пор мы рассматривали обработку аппаратных исключений, когда процессор перехватывает некое событие и возбуждает исключение. Но Вы можете и сами генерировать исключения. Это еще один способ для функции сообщить о неудаче вызвавшему ее коду. Традиционно функции, которые могут закончиться неудачно, возвращают некое особое значение — признак ошибки. При этом предполагается, что код, вызвавший функцию, проверяет, не вернула ли она это особое значение, и, если да, выполняет какие-то альтернативные операции. Как правило, вызывающая функция проводит в таких случаях соответствующую очистку и в свою очередь тоже возвращает код ошибки. Подобная передача кодов ошибок по цепочке вызовов резко усложняет написание и сопровождение кода.

Альтернативный подход заключается в том, что при неудачном вызове функции возбуждают исключение. Тогда написание и сопровождение кода становится гораздо проще, а программы работают намного быстрее. Последнее связано с тем, что та часть кода, которая отвечает за контроль ошибок, вступает в действие лишь при сбоях, т. е. в исключительных ситуациях.

К сожалению, большинство разработчиков не привыкло пользоваться исключениями для обработки ошибок. На то есть две причины. Во-первых, многие просто не знакомы с SEH. Если один разработчик создаст функцию, которая генерирует исключение, а другой не сумеет написать SEH-фрейм для перехвата этого исключения, его приложение при неудачном вызове функции будет завершено операционной системой.

Вторая причина, по которой разработчики избегают пользоваться SEH, — невозможность его переноса на другие операционные системы. Ведь компании нередко выпускают программные продукты, рассчитанные на несколько операционных систем, и, естественно, предпочитают работать с одной базой исходного кода для каждого продукта. А структурная обработка исключений — это технология, специфичная для Windows.

Если Вы все же решились на уведомление об ошибках через исключения, я апплицирую этому решению и пишу этот раздел специально для Вас. Давайте для начала посмотрим на семейство *Heap*-функций (*HeapCreate*, *HeapAlloc* и т. д.). Наверное, Вы помните из главы 18, что они предлагают разработчику возможность выбора. Обычно, когда их вызовы заканчиваются неудачно, они возвращают NULL, сообщая об ошибке. Но Вы можете передать флаг *HEAP_GENERATE_EXCEPTIONS*, и тогда при неудачном вызове *Heap*-функция не станет возвращать NULL; вместо этого она возбудит программное исключение *STATUS_NO_MEMORY*, перехватываемое с помощью SEH-фрейма.

Чтобы использовать это исключение, напишите код блока *try* так, будто выделение памяти всегда будет успешным; затем — в случае ошибки при выполнении данной операции — Вы сможете либо обработать исключение в блоке *except*, либо заставить функцию провести очистку, дополнив блок *try* блоком *finally*. Очень удобно!

Программные исключения перехватываются точно так же, как и аппаратные. Иначе говоря, все, что я рассказывал об аппаратных исключениях, в полной мере относится и к программным исключениям.

В этом разделе основное внимание мы уделим тому, как возбуждать программные исключения в функциях при неудачных вызовах. В сущности, Вы можете реализовать свои функции по аналогии с *Heap*-функциями: пусть вызывающий их код передает специальный флаг, который сообщает функциям способ уведомления об ошибках.

Возбудить программное исключение несложно — достаточно вызвать функцию *RaiseException*:

```
VOID RaiseException(
    DWORD dwExceptionCode,
    DWORD dwExceptionFlags,
    DWORD nNumberOfArguments,
    CONST ULONG_PTR *pArguments);
```

Ее первый параметр, *dwExceptionCode*, — значение, которое идентифицирует генерируемое исключение. *HeapAlloc* передает в нем STATUS_NO_MEMORY. Если Вы определяете собственные идентификаторы исключений, придерживайтесь формата, применяемого для стандартных кодов ошибок в Windows (файл WinError.h). Не забудьте, что каждый такой код представляет собой значение типа DWORD; его поля описаны в таблице 24-1. Определяя собственные коды исключений, заполните все пять его полей:

- биты 31 и 30 должны содержать код степени «тяжести»;
- бит 29 устанавливается в 1 (0 зарезервирован для исключений, определяемых Microsoft, вроде STATUS_NO_MEMORY для *HeapAlloc*);
- бит 28 должен быть равен 0;
- биты 27–16 должны указывать один из кодов подсистемы, предопределенных Microsoft;
- биты 15–0 могут содержать произвольное значение, идентифицирующее ту часть Вашего приложения, которая возбуждает исключение.

Второй параметр функции *RaiseException* — *dwExceptionFlags* — должен быть либо 0, либо EXCEPTION_NONCONTINUABLE. В принципе этот флаг указывает, может ли фильтр исключений вернуть EXCEPTION_CONTINUE_EXECUTION в ответ на данное исключение. Если Вы передаете в этом параметре нулевое значение, фильтр может вернуть EXCEPTION_CONTINUE_EXECUTION. В нормальной ситуации это заставило бы поток снова выполнить машинную команду, вызвавшую программное исключение. Однако Microsoft пошла на некоторые ухищрения, и поток возобновляет выполнение с оператора, следующего за вызовом *RaiseException*.

Но, передав функции *RaiseException* флаг EXCEPTION_NONCONTINUABLE, Вы сообщаете системе, что возобновить выполнение после данного исключения нельзя. Операционная система использует этот флаг, сигнализируя о критических (фатальных) ошибках. Например, *HeapAlloc* устанавливает этот флаг при возбуждении программного исключения STATUS_NO_MEMORY, чтобы указать системе: выполнение продолжить нельзя. Ведь если вся память занята, выделить в ней новый блок и продолжить выполнение программы не удастся.

Если возбуждается исключение EXCEPTION_NONCONTINUABLE, а фильтр все же возвращает EXCEPTION_CONTINUE_EXECUTION, система генерирует новое исключение EXCEPTION_NONCONTINUABLE_EXCEPTION.

При обработке программой одного исключения вполне вероятно возбуждение нового исключения. И смысл в этом есть. Раз уж мы остановились на этом месте, замечу, что нарушение доступа к памяти возможно и в блоке *finally*, и в фильтре исключений, и в обработчике исключений. Когда происходит нечто подобное, система создает список исключений. Помните функцию *GetExceptionInformation*? Она возвращает адрес структуры EXCEPTION_POINTERS. Ее элемент *ExceptionRecord* указывает на структуру EXCEPTION_RECORD, которая в свою очередь тоже содержит элемент *Exception-*

Record. Он указывает на другую структуру EXCEPTION_RECORD, где содержится информация о предыдущем исключении.

Обычно система единовременно обрабатывает только одно исключение, и элемент *ExceptionRecord* равен NULL. Но если исключение возбуждается при обработке другого исключения, то в первую структуру EXCEPTION_RECORD помещается информация о последнем исключении, а ее элемент *ExceptionRecord* указывает на аналогичную структуру с аналогичными данными о предыдущем исключении. Если есть и другие необработанные исключения, можно продолжить просмотр этого связанного списка структур EXCEPTION_RECORD, чтобы определить, как обработать конкретное исключение.

Третий и четвертый параметры (*nNumberOfArguments* и *pArguments*) функции *RaiseException* позволяют передать дополнительные данные о генерируемом исключении. Обычно это не нужно, и в *pArguments* передается NULL; тогда *RaiseException* игнорирует параметр *nNumberOfArguments*. А если Вы передаете дополнительные аргументы, *nNumberOfArguments* должен содержать число элементов в массиве типа ULONG_PTR, на который указывает *pArguments*. Значение *nNumberOfArguments* не может быть больше EXCEPTION_MAXIMUM_PARAMETERS (в файле WinNT.h этот идентификатор определен равным 15).

При обработке исключения написанный Вами фильтр — чтобы узнать значения *nNumberOfArguments* и *pArguments* — может ссылаться на элементы *NumberParameters* и *ExceptionInformation* структуры EXCEPTION_RECORD.

Собственные программные исключения генерируют в приложениях по целому ряду причин. Например, чтобы посыпать информационные сообщения в системный журнал событий. Как только какая-нибудь функция в Вашей программе столкнется с той или иной проблемой, Вы можете вызвать *RaiseException*; при этом обработчик исключений следует разместить выше по дереву вызовов, тогда — в зависимости от типа исключения — он будет либо заносить его в журнал событий, либо сообщать о нем пользователю. Вполне допустимо возбуждать программные исключения и для уведомления о внутренних фатальных ошибках в приложении.

Необработанные исключения и исключения C++

В предыдущей главе мы обсудили, что происходит, когда фильтр возвращает значение `EXCEPTION_CONTINUE_SEARCH`. Оно заставляет систему искать дополнительные фильтры исключений, продвигаясь вверх по дереву вызовов. А что будет, если все фильтры вернут `EXCEPTION_CONTINUE_SEARCH`? Тогда мы получим *необработанное исключение* (*unhandled exception*).

Как Вы помните из главы 6, выполнение потока начинается с функции `BaseProcessStart` или `BaseThreadStart` в `Kernel32.dll`. Единственная разница между этими функциями в том, что первая используется для запуска первичного потока процесса, а вторая — для запуска остальных потоков процесса.

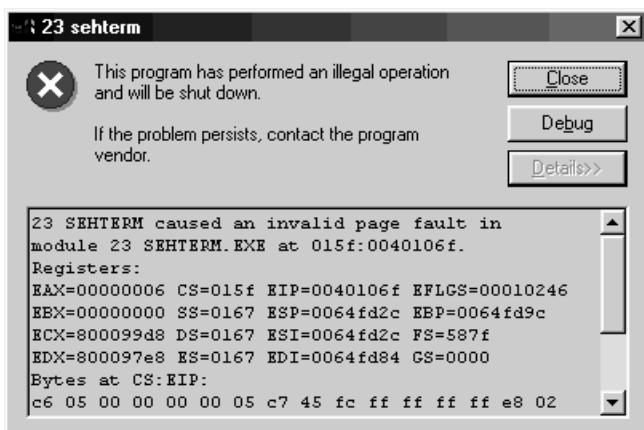
```
VOID BaseProcessStart(PPROCESS_START_ROUTINE pfnStartAddr) {
    __try {
        ExitThread((pfnStartAddr)());
    }
    __except (UnhandledExceptionFilter(GetExceptionInformation())) {
        ExitProcess(GetExceptionCode());
    }
    // Примечание: сюда мы никогда не попадем
}

VOID BaseThreadStart(PTHREAD_START_ROUTINE pfnStartAddr, PVOID pvParam) {
    __try {
        ExitThread((pfnStartAddr)(pvParam));
    }
    __except (UnhandledExceptionFilter(GetExceptionInformation())) {
        ExitProcess(GetExceptionCode());
    }
    // Примечание: сюда мы никогда не попадем
}
```

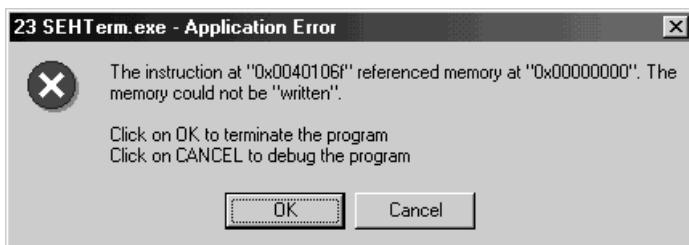
Обратите внимание, что обе функции содержат SEH-фрейм: поток запускается из блока `try`. Если поток возбудит исключение, в ответ на которое все Ваши фильтры вернут `EXCEPTION_CONTINUE_SEARCH`, будет вызвана особая функция фильтра, предоставляемая операционной системой:

```
LONG UnhandledExceptionFilter(PEXCEPTION_POINTERS pExceptionInfo);
```

Она выводит окно, указывающее на то, что поток в процессе вызвал необрабатываемое им исключение, и предлагает либо закрыть процесс, либо начать его отладку. В Windows 98 это окно выглядит следующим образом.



А в Windows 2000 оно имеет другой вид.



В Windows 2000 первая часть текста в этом окне подсказывает тип исключения и адрес вызвавшей его инструкции в адресном пространстве процесса. У меня окно появилось из-за нарушения доступа к памяти, поэтому система сообщила адрес, по которому произошла ошибка, и тип доступа к памяти — чтение. *UnhandledExceptionFilter* получает эту информацию из элемента *ExceptionInformation* структуры *EXCEPTION_RECORD*, инициализированной для этого исключения.

В данном окне можно сделать одно из двух. Во-первых, щелкнуть кнопку OK, и тогда *UnhandledExceptionFilter* вернет *EXCEPTION_EXECUTE_HANDLER*. Это приведет к глобальной раскрутке и соответственно к выполнению всех имеющихся блоков *finally*, а затем и к выполнению обработчика в *BaseProcessStart* или *BaseThreadStart*. Оба обработчика вызывают *ExitProcess*, поэтому-то Ваш процесс и закрывается. Причем кодом завершения процесса становится код исключения. Кроме того, процесс закрывается его же потоком, а не операционной системой! А это означает, что Вы можете вмешаться в ход завершения своего процесса.

Во-вторых, Вы можете щелкнуть кнопку Cancel (сбываются самые смелые мечты программистов). В этом случае *UnhandledExceptionFilter* попытается запустить отладчик и подключить его к процессу. Тогда Вы сможете просматривать состояние глобальных, локальных и статических переменных, расставлять точки прерывания, перезапускать процесс и вообще делать все, что делается при отладке процесса.

Но самое главное, что сбой в программе можно исследовать в момент его возникновения. В большинстве других операционных систем для отладки процесса сначала запускается отладчик. При генерации исключения в процессе, выполняемом в любой из таких систем, этот процесс надо завершить, запустить отладчик и прогнать программу уже под отладчиком. Проблема, правда, в том, что ошибку надо сначала воспроизвести; лишь потом можно попытаться ее исправить. А кто знает, какие значения были у переменных, когда Вы впервые заметили ошибку? Поэтому найти ее та-

ким способом гораздо труднее. Возможность динамически подключать отладчик к уже запущенному процессу — одно из лучших качеств Windows.

**WINDOWS
2000**

В этой книге рассматривается разработка приложений, работающих только в пользовательском режиме. Но, наверное, Вас интересует, что происходит, когда необработанное исключение возникает в потоке, выполняемом в режиме ядра. Так вот, исключения в режиме ядра обрабатываются так же, как и исключения пользователяского режима. Если низкоуровневая функция для работы с виртуальной памятью возбуждает исключение, система проверяет, есть ли фильтр режима ядра, готовый обработать это исключение. Если такого фильтра нет, оно остается необработанным. В этом случае необработанное исключение окажется в операционной системе или (что вероятнее) в драйвере устройства, а не в приложении. А это уже серьезно!

Так как дальнейшая работа системы после необработанного исключения в режиме ядра небезопасна, Windows не вызывает *UnhandledExceptionFilter*. Вместо этого появляется так называемый «синий экран смерти»: экран переключается в текстовый режим, окрашивается в синий фон, выводится информация о модуле, вызвавшем необработанное исключение, и система останавливается. Вам следует записать эту информацию и отправить ее в Microsoft или поставщику драйвера устройства. Прежде чем продолжить работу, придется перезагрузить машину; при этом все несохраненные данные теряются.

Отладка по запросу

Windows позволяет подключать отладчик к любому процессу в любой момент времени — эта функциональность называется отладкой по запросу (just-in-time debugging). В этом разделе я расскажу, как она работает. Щелкнув кнопку Cancel, Вы сообщаете функции *UnhandledExceptionFilter* о том, что хотите начать отладку процесса.

Для активизации отладчика *UnhandledExceptionFilter* просматривает раздел реестра: HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\AeDebug

Если Вы установили Visual Studio, то содержащийся в этом разделе параметр Debugger имеет следующее значение:

```
"C:\Program Files\Microsoft Visual Studio\Common\MSDev98\Bin\msdev.exe"
-p %ld -e %ld
```

**WINDOWS
98**

В Windows 98 соответствующие значения хранятся не в реестре, а в файле Win.ini.

Строка, приведенная выше, сообщает системе, какой отладчик надо запустить (в данном случае — MSDev.exe). Естественно, Вы можете изменить это значение, указав другой отладчик. *UnhandledExceptionFilter* передает отладчику два параметра в командной строке. Первый — это идентификатор процесса, который нужно отладить, а второй — наследуемое событие со сбросом вручную, которое создается функцией *UnhandledExceptionFilter* в занятом состоянии. Отладчик должен распознавать ключи *-p* и *-e* как идентификатор процесса и описатель события.

Сформировав командную строку из идентификатора процесса и описателя события, *UnhandledExceptionFilter* запускает отладчик вызовом *CreateProcess*. Отладчик про-

веряет аргументы в командной строке и, обнаружив ключ *-p*, подключается к соответствующему процессу вызовом *DebugActiveProcess*:

```
BOOL DebugActiveProcess(DWORD dwProcessID);
```

После этого система начинает уведомлять отладчик о состоянии отлаживаемого процесса, сообщая, например, сколько в нем потоков и какие DLL спроектированы на его адресное пространство. На сбор этих данных отладчику нужно какое-то время, в течение которого поток *UnhandledExceptionFilter* должен находиться в режиме ожидания. Для этого функция вызывает *WaitForSingleObject* и передает описатель созданного ею события со сбросом вручную. Как Вы помните, оно было создано в занятом состоянии, поэтому поток отлаживаемого процесса немедленно приостанавливается и ждет освобождения этого события.

Закончив инициализацию, отладчик вновь проверяет командную строку — на этот раз он ищет ключ *-e*. Найдя его, отладчик считывает описатель события и вызывает *SetEvent*. Он может напрямую использовать этот наследуемый описатель, поскольку процесс отладчика является дочерним по отношению к отлаживаемому процессу, который и породил его, вызвав *UnhandledExceptionFilter*.

Переход события в свободное состояние пробуждает поток отлаживаемого процесса, и он передает отладчику информацию о необработанном исключении. Получив эти данные, отладчик загружает соответствующий файл исходного кода и переходит к команде, которая вызвала исключение. Вот это действительно круто!

Кстати, совсем не обязательно дожидаться исключения, чтобы начать отладку. Отладчик можно подключить в любой момент командой «MSDEV -p PID», где PID — идентификатор отлаживаемого процесса. Task Manager в Windows 2000 еще больше упрощает эту задачу. Открыв вкладку Process, Вы можете щелкнуть строку с нужным процессом правой кнопкой мыши и выбрать из контекстного меню команду Debug. В ответ Task Manager обратится к только что рассмотренному разделу реестра и вызовет *CreateProcess*, передав ей идентификатор выбранного процесса. Но вместо описателя события Task Manager передаст 0.

Отключение вывода сообщений об исключении

Иногда нужно, чтобы окно с сообщением об исключении не появлялось на экране, — например, в готовом программном продукте. Ведь если такое окно появится, пользователь может случайно перейти в режим отладки Вашей программы. Стоит ему только щелкнуть кнопку Cancel, и он шагнет на незнакомую и страшную территорию — попадет в отладчик. Поэтому предусмотрено несколько способов, позволяющих избежать появления этого окна на экране.

Принудительное завершение процесса

Запретить функции *UnhandledExceptionFilter* вывод окна с сообщением об исключении можно вызовом *SetErrorMode* с передачей идентификатора SEM_NOGPFAULTERRORBOX:

```
UINT SetErrorMode(UINT fuErrorMode);
```

Тогда *UnhandledExceptionFilter*, вызванная для обработки исключения, немедленно вернет EXCEPTION_EXECUTE_HANDLER, что приведет к глобальной раскрутке и выполнению обработчика в *BaseProcessStart* или *BaseThreadStart*, который закроет процесс.

Лично мне этот способ не нравится, так как пользователь не получает никакого предупреждения — приложение просто исчезает.

Создание оболочки вокруг функции потока

Другой способ состоит в том, что Вы помещаете входную функцию первичного потока (*main*, *wmain*, *WinMain* или *wWinMain*) в блок *try-except*. Фильтр исключений должен всегда возвращать *EXCEPTION_EXECUTE_HANDLER*, чтобы исключение действительно обрабатывалось; это предотвратит вызов *UnhandledExceptionFilter*.

В обработчике исключений Вы выводите на экран диалоговое окно с какой-нибудь диагностической информацией. Пользователь может скопировать эту информацию и передать ее в службу технической поддержки Вашего приложения, что поможет выявить источник проблем. Это диалоговое окно надо разработать так, чтобы пользователь мог завершить приложение, но не отлаживать.

Этому способу присущ один недостаток: он позволяет перехватывать только те исключения, которые возникают в первичном потоке. Если исключение происходит в любом другом потоке процесса, система вызывает функцию *UnhandledExceptionFilter*. Чтобы вывернуться из этой ситуации, придется также включить блоки *try-except* во входные функции всех вторичных потоков Вашего процесса.

Создание оболочки вокруг всех функций потоков

Функция *SetUnhandledExceptionFilter* позволяет включать все функции потоков в SEH-фрейм:

```
PTOP_LEVEL_EXCEPTION_FILTER SetUnhandledExceptionFilter(
    PTOP_LEVEL_EXCEPTION_FILTER pTopLevelExceptionFilter);
```

После ее вызова необработанное исключение, возникшее в любом из потоков процесса, приведет к вызову Вашего фильтра исключений. Адрес фильтра следует передать в единственном параметре функции *SetUnhandledExceptionFilter*. Прототип этой функции-фильтра должен выглядеть так:

```
LONG UnhandledExceptionFilter(PEXCEPTION_POINTERS pExceptionInfo);
```

По форме она идентична функции *UnhandledExceptionFilter*. Внутри фильтра можно проводить любую обработку, а возвращаемым значением должен быть один из трех идентификаторов типа *EXCEPTION_**. В следующей таблице описано, что происходит в случае возврата каждого из идентификаторов.

Идентификатор	Действие
<i>EXCEPTION_EXECUTE_HANDLER</i>	Процесс просто завершается, так как система не выполняет никаких операций в своем обработчике исключений
<i>EXCEPTION_CONTINUE_EXECUTION</i>	Выполнение продолжается с инструкции, вызвавшей исключение; Вы можете модифицировать информацию об исключении, на которую указывает параметр типа <i>PEXCEPTION_POINTERS</i>
<i>EXCEPTION_CONTINUE_SEARCH</i>	Выполняется обычная Windows-функция <i>UnhandledExceptionFilter</i>

Чтобы функция *UnhandledExceptionFilter* вновь стала фильтром по умолчанию, вызовите *SetUnhandledExceptionFilter* со значением NULL. Заметьте также, что всякий раз, когда устанавливается новый фильтр для необработанных исключений, *SetUn-*

handledExceptionFilter возвращает адрес ранее установленного фильтра. Если таким фильтром была *UnhandledExceptionFilter*, возвращается NULL. Если Ваш фильтр возвращает EXCEPTION_CONTINUE_SEARCH, Вы должны вызывать ранее установленный фильтр, адрес которого вернула *SetUnhandledExceptionFilter*.

Автоматический вызов отладчика

Это последний способ отключения окна с сообщением об исключении. В уже упомянутом разделе реестра есть еще один параметр — Auto; его значение может быть либо 0, либо 1. В последнем случае *UnhandledExceptionFilter* не выводит окно, но сразу же вызывает отладчик. А при нулевом значении функция выводит сообщения и работает так, как я уже рассказывал.

Явный вызов функции *UnhandledExceptionFilter*

Функция *UnhandledExceptionFilter* полностью задокументирована, и Вы можете сами вызывать ее в своих программах. Вот пример ее использования:

```
void Funcadelic() {
    __try {
        :
    }
    __except (ExpFltr(GetExceptionInformation())) {
        :
    }
}

LONG ExpFltr(PEXCEPTION_POINTERS pEP) {
    DWORD dwExceptionCode = pEP->ExceptionRecord.ExceptionCode;

    if (dwExceptionCode == EXCEPTION_ACCESS_VIOLATION) {
        // что-то делаем здесь...
        return(EXCEPTION_CONTINUE_EXECUTION);
    }
    return(UnhandledExceptionFilter(pEP));
}
```

Исключение в блоке *try* функции *Funcadelic* приводит к вызову *ExpFltr*. Ей передается значение, возвращаемое *GetExceptionInformation*. Внутри фильтра определяется код исключения и сравнивается с EXCEPTION_ACCESS_VIOLATION. Если было нарушение доступа, фильтр исправляет ситуацию и возвращает EXCEPTION_CONTINUE_EXECUTION. Это значение заставляет систему возобновить выполнение программы с инструкции, вызвавшей исключение.

Если произошло какое-то другое исключение, *ExpFltr* вызывает *UnhandledExceptionFilter*, передавая ей адрес структуры EXCEPTION_POINTERS. Функция *UnhandledExceptionFilter* открывает окно, позволяющее завершить процесс или начать отладку. Ее возвращаемое значение становится и результатом функции *ExpFltr*.

Функция *UnhandledExceptionFilter* изнутри

Начав работать с исключениями, я решил, что можно извлечь массу информации, если детально вникнуть в механизм работы функции *UnhandledExceptionFilter*. Поэтому я тщательно его исследовал. Вот что делает функция *UnhandledExceptionFilter*.

1. Если возникло нарушение доступа и его причина связана с попыткой записи, система проверяет, не пытались ли Вы модифицировать ресурс в EXE- или DLL-модуле. По умолчанию такие ресурсы предназначены только для чтения. Однако 16-разрядная Windows разрешала модифицировать эти ресурсы, и из соображений обратной совместимости такие операции должны поддерживаться как в 32-, так и в 64-разрядной Windows. Поэтому, когда Вы пытаетесь модифицировать ресурс, *UnhandledExceptionFilter* вызывает *VirtualProtect* для изменения атрибута защиты страницы с этим ресурсом на PAGE_READWRITE и возвращает EXCEPTION_CONTINUE_EXECUTION.
2. Если Вы установили свой фильтр вызовом *SetUnhandledExceptionFilter*, функция *UnhandledExceptionFilter* обращается к Вашей функции фильтра. И если она возвращает EXCEPTION_EXECUTE_HANDLER или EXCEPTION_CONTINUE_EXECUTION, *UnhandledExceptionFilter* передает его системе. Но, если Вы не устанавливали свой фильтр необработанных исключений или если функция фильтра возвращает EXCEPTION_CONTINUE_SEARCH, *UnhandledExceptionFilter* переходит к операциям, описанным в п. 3.

**WINDOWS
98**

Из-за ошибки в Windows 98 Ваша функция фильтра необработанных исключений вызывается, только если к процессу не подключен отладчик. По той же причине в Windows 98 невозможна отладка программы *Spreadsheet*, представленной в следующем разделе.

3. Если Ваш процесс выполняется под управлением отладчика, то возвращается EXCEPTION_CONTINUE_SEARCH. Это может показаться странным, так как система уже выполняет самый «верхний» блок *try* или *except* и другого фильтра выше по дереву вызовов просто нет. Но, обнаружив этот факт, система сообщит отладчику о необработанном исключении в подопечном ему процессе. В ответ на это отладчик выведет окно, где предложит начать отладку. (Кстати, функция *IsDebuggerPresent* позволяет узнать, работает ли данный процесс под управлением отладчика.)
4. Если поток в Вашем процессе вызовет *SetErrorMode* с флагом SEM_NOGPFAULTERRORBOX, то *UnhandledExceptionFilter* вернет EXCEPTION_EXECUTE_HANDLER.
5. Если процесс включен в задание (см. главу 5), на которое наложено ограничение JOB_OBJECT_LIMIT_DIE_ON_UNHANDLED_EXCEPTION, то *UnhandledExceptionFilter* также вернет EXCEPTION_EXECUTE_HANDLER.

**WINDOWS
98**

Windows 98 не поддерживает задания, и в ней этот этап пропускается.

6. *UnhandledExceptionFilter* считывает в реестре значение параметра Auto. Если оно равно 1, происходит переход на этап 7, в ином случае выводится окно с информацией об исключении. Если в реестре присутствует и параметр Debugger, в этом окне появляются кнопки OK и Cancel. А если этого параметра нет — только кнопка OK. Как только пользователь щелкнет кнопку OK, функция *UnhandledExceptionFilter* вернет EXCEPTION_EXECUTE_HANDLER. Щелчок кнопки Cancel (если она есть) вызывает переход на следующий этап.

**WINDOWS
98**

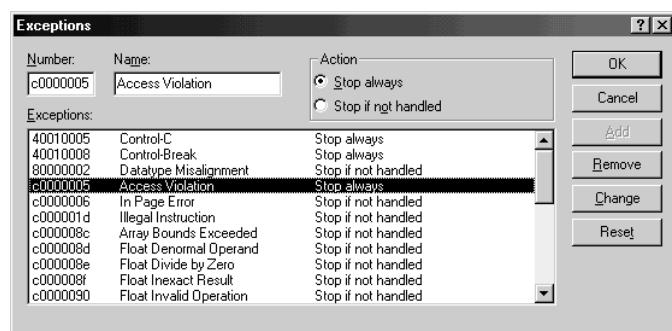
В Windows 98 упомянутые параметры хранятся не в реестре, а в файле Win.ini.

7. На этом этапе *UnhandledExceptionFilter* запускает отладчик как дочерний процесс. Но сначала создает событие со сбросом вручную в занятом состоянии и наследуемым описателем. Затем извлекает из реестра значение параметра Debugger и вызывает *sprintf* для вставки идентификатора процесса (полученного через функцию *GetCurrentProcessId*) и описателя события в командную строку. Элементу *lpDesktop* структуры STARTUPINFO присваивается значение «*Winsta0\\Default*», чтобы отладчик был доступен в интерактивном режиме на рабочем столе. Далее вызывается *CreateProcess* со значением TRUE в параметре *bInheritHandles*, благодаря чему отладчик получает возможность наследовать описатель объекта «событие». После этого *UnhandledExceptionFilter* ждет завершения инициализации отладчика, вызвав *WaitForSingleObjectEx* с передачей ей описателя события. Заметьте, что вместо *WaitForSingleObject* используется *WaitForSingleObjectEx*. Это заставляет поток ждать в «тревожном» состоянии, которое позволяет ему обрабатывать все поступающие APC-вызовы.
8. Закончив инициализацию, отладчик освобождает событие, и поток *UnhandledExceptionFilter* пробуждается. Теперь, когда процесс находится под управлением отладчика, *UnhandledExceptionFilter* возвращает EXCEPTION_CONTINUE_SEARCH. Обратите внимание: все, что здесь происходит, точно соответствует этапу 3.

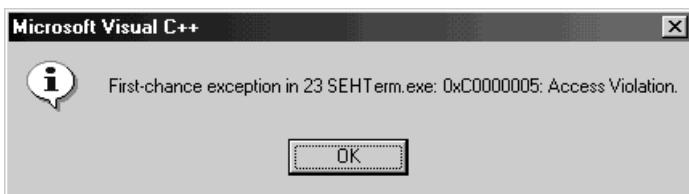
Исключения и отладчик

Отладчик Microsoft Visual C++ предоставляет фантастические возможности для отладки после исключений. Когда поток процесса вызывает исключение, операционная система немедленно уведомляет об этом отладчик (если он, конечно, подключен). Это уведомление называется «первым предупреждением» (first-chance notification). Реагируя на него, отладчик обычно заставляет поток искать фильтры исключений. Если все фильтры возвращают EXCEPTION_CONTINUE_SEARCH, операционная система вновь уведомляет отладчик, но на этот раз дает «последнее предупреждение» (last-chance notification). Существование этих двух типов предупреждений обеспечивает больший контроль за отладкой при исключениях.

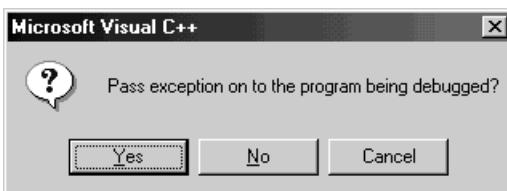
Чтобы сообщить отладчику, как реагировать на первое предупреждение, используйте диалоговое окно Exceptions отладчика.



Как видите, оно содержит список всех исключений, определенных в системе. Для каждого из них сообщаются 32-битный код, текстовое описание и ответные действия отладчика. Я выбрал исключение Access Violation (нарушение доступа) и указал для него Stop Always. Теперь, если поток в отлаживаемом процессе вызовет это исключение, отладчик выведет при первом предупреждении следующее окно.

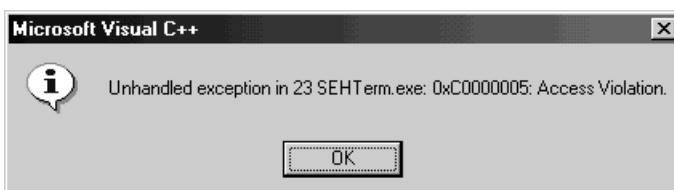


К этому моменту поток еще *не* получал шанса на поиск фильтров исключений. Сейчас я могу поместить в исходный код точки прерывания, просмотреть значения переменных или проверить стек вызовов потока. Пока ни один фильтр не выполнялся — исключение произошло только что. Когда я попытаюсь начать пошаговую отладку программы, на экране появится новое окно.



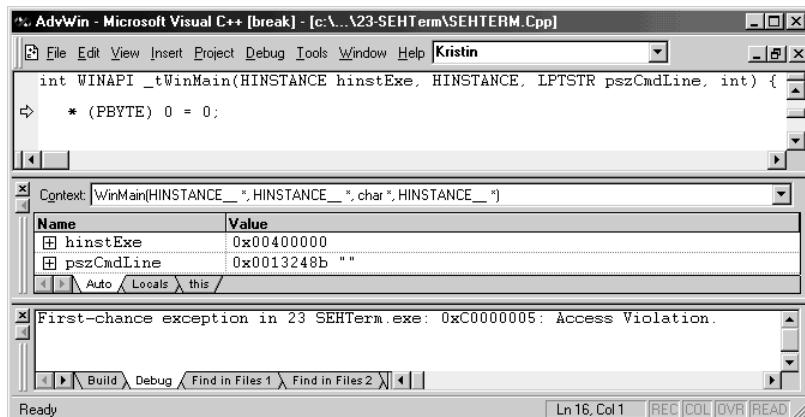
Кнопка *Cancel* вернет нас в отладчик. Кнопка *No* заставит поток отлаживаемого процесса повторить выполнение неудавшейся машинной команды. При большинстве исключений повторное выполнение команды ничего не даст, так как вновь вызовет исключение. Однако, если исключение было сгенерировано с помощью функции *RaiseException*, это позволит возобновить выполнение потока, и он продолжит работу, как ни в чем ни бывало. Данный метод может быть особенно полезен при отладке программ на C++: получится так, будто оператор *throw* никогда не выполнялся. (К обработке исключений в C++ мы вернемся в конце главы.)

И, наконец, кнопка *Yes* разрешит потоку отлаживаемого процесса начать поиск фильтров исключений. Если фильтр исключения, возвращающий *EXCEPTION_EXECUTE_HANDLER* или *EXCEPTION_CONTINUE_EXECUTION*, найден, то все хорошо и поток продолжает работу. Если же все фильтры вернут *EXCEPTION_CONTINUE_SEARCH*, отладчик получит последнее предупреждение и выведет окно с сообщением, аналогичным тому, которое показано ниже.

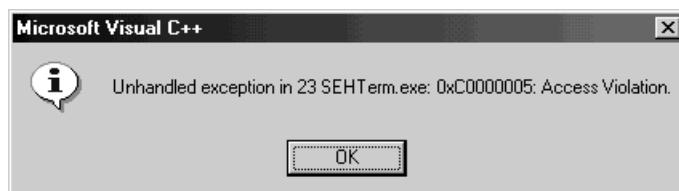


Здесь Вам придется либо начать отладку, либо закрыть приложение.

Я продемонстрировал Вам, что случится, если ответным действием отладчика выбран вариант *Stop Always*. Но для большинства исключений по умолчанию предлагаются варианты *Stop If Not Handled*. В этом случае отладчик, получив первое предупреждение, просто сообщает о нем в своем окне *Output*.

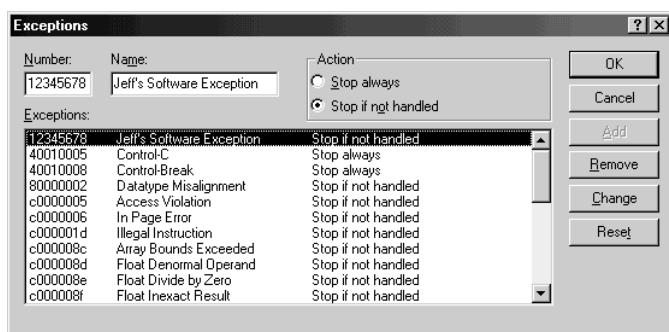


После этого отладчик разрешит потоку искать подходящие фильтры и, только если исключение не будет обработано, откроет следующее окно.



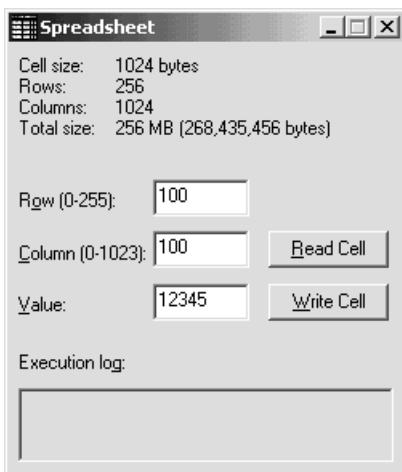
Очень важно помнить, что первое предупреждение вовсе не говорит о каких-либо проблемах или «жучках» в приложении. В сущности, оно появляется только при отладке. Отладчик просто сообщает о возникновении исключения, и, если после этого он не выводит уже известное Вам окно, это означает лишь одно: фильтр обработал исключение, приложение продолжает нормально работать. А вот последнее предупреждение говорит о том, что в Вашей программе есть некая проблема, которую надо устранить.

Прежде чем закончить обсуждение этой темы, хотелось бы упомянуть еще об одной особенности диалогового окна Exceptions отладчика. Оно полностью поддерживает любые определяемые Вами программные исключения. От Вас требуется лишь указать уникальный числовой код исключения, его название и ответное действие отладчика, а затем, щелкнув кнопку Add, добавить это новое исключение в список. Посмотрите, как это сделал я, определив собственное исключение.



Программа-пример Spreadsheet

Эта программа, «25 Spreadsheet.exe» (см. листинг на рис. 25-1), демонстрирует, как передавать физическую память зарезервированному региону адресного пространства — но не всему региону, а только его областям, нужным в данный момент. Алгоритм опирается на структурную обработку исключений. Файлы исходного кода и ресурсов этой программы находятся в каталоге 25-Spreadsheet на компакт-диске, прилагаемом к книге. После запуска Spreadsheet на экране появляется диалоговое окно, показанное ниже.



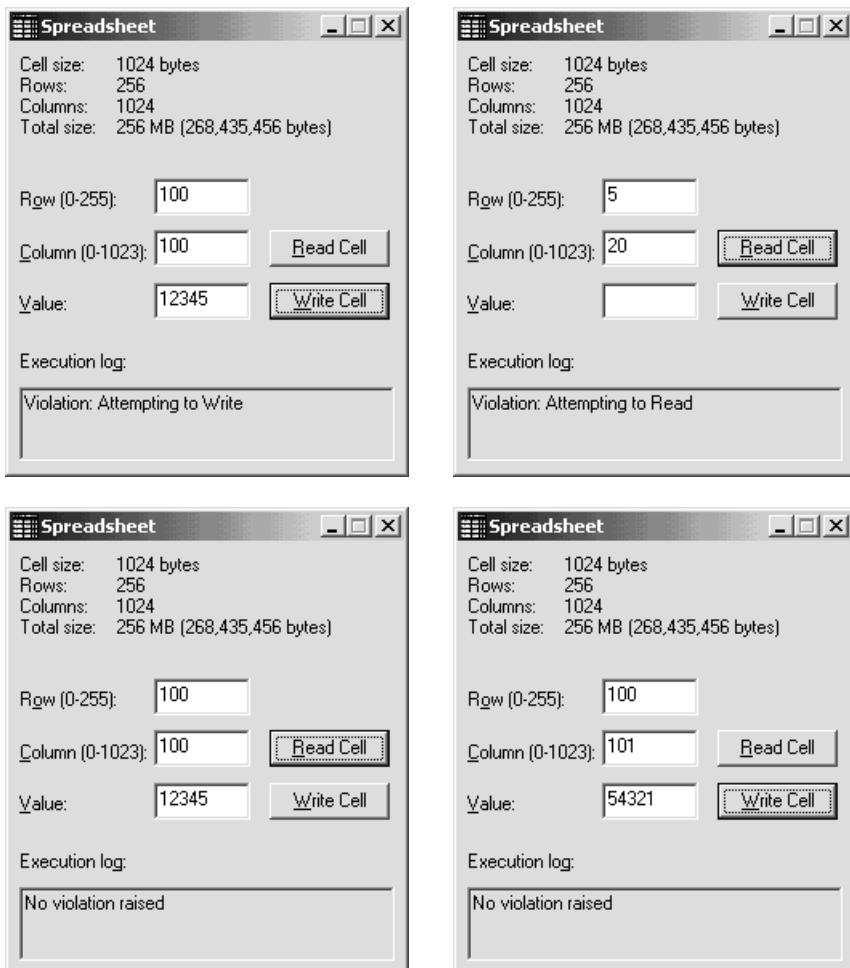
Программа Spreadsheet резервирует регион для двухмерной таблицы, содержащей 256 строк и 1024 колонки, с размером ячеек по 1024 байта. Если бы программа заранее передавала физическую память под всю таблицу, то ей понадобилось бы 268 435 456 байтов, или 256 Мб. Поэтому для экономии драгоценных ресурсов программа резервирует в своем адресном пространстве регион размером 256 Мб, не передавая ему физическую память.

Допустим, пользователь хочет поместить значение 12345 в ячейку на пересечении строки 100 и колонки 100 (как на предыдущей иллюстрации). Как только он щелкнет кнопку Write Cell, программа попытается записать это значение в указанную ячейку таблицы. Естественно, это вызовет нарушение доступа. Но, так как я использую в программе SEH, мой фильтр исключений, распознав попытку записи, выведет в нижней части диалогового окна сообщение «Violation: Attempting to Write», передаст память под нужную ячейку и заставит процессор повторить выполнение команды, возбудившей исключение. Теперь значение будет сохранено в ячейке таблицы, поскольку этой ячейке передана физическая память.

Проделаем еще один эксперимент. Попробуем считать значение из ячейки на пересечении строки 5 и колонки 20. Этим мы вновь вызовем нарушение доступа. На этот раз фильтр исключений не передаст память, а выведет в диалоговом окне сообщение «Violation: Attempting to Read». Программа корректно возобновит свою работу после неудавшейся попытки чтения, очистив поле Value диалогового окна.

Третий эксперимент: попробуем считать значение из ячейки на пересечении строки 100 и колонки 100. Так как этой ячейке передана физическая память, никаких исключений не возбуждается, и фильтр не выполняется (что положительно сказывается на быстродействии программы). Диалоговое окно будет выглядеть следующим образом.

Ну и последний эксперимент: запишем значение 54321 в ячейку на пересечении строки 100 и колонки 101. Эта операция пройдет успешно, без исключений, потому что данная ячейка находится на той же странице памяти, что и ячейка (100, 100). В подтверждение этого Вы увидите сообщение «No Violation raised» в нижней части диалогового окна.



В своих проектах я довольно часто пользуюсь виртуальной памятью и СЕН. Как-то раз я решил создать шаблонный C++-класс CVMArray, который инкапсулирует все, что нужно для использования этих механизмов. Его исходный код содержится в файле VMArray.h (он является частью программы-примера Spreadsheet). Вы можете работать с классом CVMArray двумя способами. Во-первых, просто создать экземпляр этого класса, передав конструктору максимальное число элементов массива. Класс автоматически устанавливает действующий на уровне всего процесса фильтр необработанных исключений, чтобы любое обращение из любого потока к адресу в виртуальном массиве памяти заставляло фильтр вызывать *VirtualAlloc* (для передачи физической памяти новому элементу) и возвращать *EXCEPTION_CONTINUE_EXECUTION*. Такое применение класса CVMArray позволяет работать с разреженной памятью (sparse storage), не забивая СЕН-фреймами исходный код программы. Единственный недостаток

ток в том, что Ваше приложение не сможет возобновить корректную работу, если по каким-то причинам передать память не удастся.

Второй способ использования CVMArray — создание производного C++-класса. Производный класс даст Вам все преимущества базового класса, и, кроме того, Вы сможете расширить его функциональность — например, заменив исходную виртуальную функцию *OnAccessViolation* собственной реализацией, более аккуратно обрабатывающей нехватку памяти. Программа Spreadsheet как раз и демонстрирует этот способ использования класса CVMArrray.



Spreadsheet.cpp

```

 ****
Модуль: Spreadsheet.cpp
Автор: Copyright (c) 2000, Джейфри Рихтер (Jeffrey Richter)
****

#include "..\CmnHdr.h"      /* см. приложение A */
#include <windowsx.h>
#include <tchar.h>
#include "Resource.h"
#include "VMArray.h"

///////////////////////////////
HWND g_hwnd; // глобальный описатель окна, применяемого для SEH-отчетов

const int g_nNumRows = 256;
const int g_nNumCols = 1024;

// определяем структуру ячеек таблицы
typedef struct {
    DWORD dwValue;
    BYTE bDummy[1020];
} CELL, *PCELL;

// объявляем тип данных для всей таблицы
typedef CELL SPREADSHEET[g_nNumRows][g_nNumCols];
typedef SPREADSHEET *PSPREADSHEET;

///////////////////////////////

// таблица является двухмерным массивом переменных типа CELL
class CVMSpreadsheet : public CVMArray<CELL> {
public:
    CVMSpreadsheet() : CVMArray<CELL>(g_nNumRows * g_nNumCols) {}

private:
    LONG OnAccessViolation(PVOID pvAddrTouched, BOOL fAttemptedRead,
        PEXCEPTION_POINTERS pep, BOOL fRetryUntilSuccessful);
};


```

Рис. 25-1. Программа-пример *Spreadsheet*

см. след. стр.

Рис. 25-1. продолжение

```
//////////  
  
LONG CVMSpreadsheet::OnAccessViolation(PVOID pvAddrTouched, BOOL fAttemptedRead,  
PEXCEPTION_POINTERS pep, BOOL fRetryUntilSuccessful) {  
  
    TCHAR sz[200];  
    wsprintf(sz, TEXT("Violation: Attempting to %s"),  
        fAttemptedRead ? TEXT("Read") : TEXT("Write"));  
    SetDlgItemText(g_hwnd, IDC_LOG, sz);  
  
    LONG lDisposition = EXCEPTION_EXECUTE_HANDLER;  
    if (!fAttemptedRead) {  
  
        // возвращаем значение, определяемое базовым классом  
        lDisposition = CVMArraу<CELL>::OnAccessViolation(pvAddrTouched,  
            fAttemptedRead, pep, fRetryUntilSuccessful);  
    }  
    return(lDisposition);  
}  
  
//////////  
  
// это глобальный объект CVMSpreadsheet  
static CVMSpreadsheet g_ssObject;  
  
// создаем глобальный указатель на весь регион, зарезервированный под таблицу  
SPREADSHEET& g_ss = * (PSPREADSHEET) (PCELL) g_ssObject;  
  
//////////  
  
BOOL Dlg_OnInitDialog(HWND hwnd, HWND hwndFocus, LPARAM lParam) {  
  
    chSETDLGICONS(hwnd, IDI_SPREADSHEET);  
  
    g_hwnd = hwnd; // сохраняем для SEH-отчетов  
  
    // инициализируем элементы управления в диалоговом окне значениями по умолчанию  
    Edit_LimitText(GetDlgItem(hwnd, IDC_ROW), 3);  
    Edit_LimitText(GetDlgItem(hwnd, IDC_COLUMN), 4);  
    Edit_LimitText(GetDlgItem(hwnd, IDC_VALUE), 7);  
    SetDlgItemInt(hwnd, IDC_ROW, 100, FALSE);  
    SetDlgItemInt(hwnd, IDC_COLUMN, 100, FALSE);  
    SetDlgItemInt(hwnd, IDC_VALUE, 12345, FALSE);  
    return(TRUE);  
}  
  
//////////  
  
void Dlg_OnCommand(HWND hwnd, int id, HWND hwndCtl, UINT codeNotify) {  
  
    int nRow, nCol;
```

Рис. 25-1. продолжение

```

switch (id) {
    case IDCANCEL:
        EndDialog(hwnd, id);
        break;

    case IDC_ROW:
        // пользователь изменил строку, обновляем данные в окне
        nRow = GetDlgItemInt(hwnd, IDC_ROW, NULL, FALSE);
        EnableWindow(GetDlgItem(hwnd, IDC_READCELL),
                     chINRANGE(0, nRow, g_nNumRows - 1));
        EnableWindow(GetDlgItem(hwnd, IDC_WRITECELL),
                     chINRANGE(0, nRow, g_nNumRows - 1));
        break;

    case IDC_COLUMN:
        // пользователь изменил колонку, обновляем данные в окне
        nCol = GetDlgItemInt(hwnd, IDC_COLUMN, NULL, FALSE);
        EnableWindow(GetDlgItem(hwnd, IDC_READCELL),
                     chINRANGE(0, nCol, g_nNumCols - 1));
        EnableWindow(GetDlgItem(hwnd, IDC_WRITECELL),
                     chINRANGE(0, nCol, g_nNumCols - 1));
        break;

    case IDC_READCELL:
        // пытаемся считать значение ячейки, выбранной пользователем
        SetDlgItemText(g_hwnd, IDC_LOG, TEXT("No violation raised"));
        nRow = GetDlgItemInt(hwnd, IDC_ROW, NULL, FALSE);
        nCol = GetDlgItemInt(hwnd, IDC_COLUMN, NULL, FALSE);
        __try {
            SetDlgItemInt(hwnd, IDC_VALUE, g_ss[nRow][nCol].dwValue, FALSE);
        }
        __except (g_ssObject.ExceptionFilter(GetExceptionInformation(), FALSE)) {

            // ячейке не передана физическая память, и ее значение не определено
            SetDlgItemText(hwnd, IDC_VALUE, TEXT(""));
        }
        break;

    case IDC_WRITECELL:
        // пытаемся считать значение ячейки, выбранной пользователем
        SetDlgItemText(g_hwnd, IDC_LOG, TEXT("No violation raised"));
        nRow = GetDlgItemInt(hwnd, IDC_ROW, NULL, FALSE);
        nCol = GetDlgItemInt(hwnd, IDC_COLUMN, NULL, FALSE);

        // если ячейке не передана физическая память,
        // возбуждается исключение, и память передается ей автоматически
        g_ss[nRow][nCol].dwValue =
            GetDlgItemInt(hwnd, IDC_VALUE, NULL, FALSE);
        break;
}

```

см. след. стр.

Рис. 25-1. продолжение

```
//////////  

INT_PTR WINAPI Dlg_Proc(HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam) {  

    switch (uMsg) {  

        chHANDLE_DLGMMSG(hwnd, WM_INITDIALOG, Dlg_OnInitDialog);  

        chHANDLE_DLGMMSG(hwnd, WM_COMMAND, Dlg_OnCommand);  

    }  

    return(FALSE);  

}  

//////////  

int WINAPI _tWinMain(HINSTANCE hinstExe, HINSTANCE, PTSTR pszCmdLine, int) {  

    DialogBox(hinstExe, MAKEINTRESOURCE(IDD_SPREADSHEET), NULL, Dlg_Proc);  

    return(0);  

}  

////////// Конец файла /////////////////
```

VMArray.h

```
*****  

Модуль: VMArray.h  

Автор: Copyright (c) 2000, Джейфри Рихтер (Jeffrey Richter)  

*****  

#pragma once  

//////////  

// Примечание: этот C++-класс небезопасен в многопоточной среде. Несколько потоков  

// не может одновременно создавать/уничтожать объекты этого класса. Однако несколько  

// потоков может одновременно обращаться к уже созданным объектам класса CVMArry  

// (для доступа к одному объекту Вам придется самостоятельно синхронизировать потоки).  

//////////  

template <class TYPE>  

class CVMArry {  

public:  

    // резервирует разреженную матрицу  

    CVMArry(DWORD dwReserveElements);  

    // освобождает разреженную матрицу  

    virtual ~CVMArry();  

    // обеспечивает доступ к элементу массива  

    operator TYPE*() { return(m_pArray); }  

    operator const TYPE*() const { return(m_pArray); }
```

Рис. 25-1. продолжение

```

// может вызываться для более "тонкой" обработки,
// если передать память не удалось
LONG ExceptionFilter(PEXCEPTION_POINTERS pep,
    BOOL fRetryUntilSuccessful = FALSE);

protected:
    // замещается для более "тонкой" обработки исключений,
    // связанного с нарушением доступа
    virtual LONG OnAccessViolation(PVOID pvAddrTouched, BOOL fAttemptedRead,
        PEXCEPTION_POINTERS pep, BOOL fRetryUntilSuccessful);

private:
    static CVMArray* sm_pHead;      // адрес первого объекта
    CVMArray* m_pNext;            // адрес следующего объекта
    TYPE* m_pArray;               // указатель на регион, зарезервированный
                                  // под массив (разреженную матрицу)
    DWORD m_cbReserve;           // размер зарезервированного региона (в байтах)

private:
    // адрес предыдущего фильтра необработанных исключений
    static PTOP_LEVEL_EXCEPTION_FILTER sm_pfnUnhandledExceptionFilterPrev;

    // наш глобальный фильтр необработанных исключений
    // для экземпляров этого класса
    static LONG WINAPI UnhandledExceptionFilter(PEXCEPTION_POINTERS pep);
};

///////////////////////////////



// заголовок связанного списка объектов
template <class TYPE>
CVMArray<TYPE>* CVMArray<TYPE>::sm_pHead = NULL;

// адрес предыдущего фильтра необработанных исключений
template <class TYPE>
PTOP_LEVEL_EXCEPTION_FILTER CVMArray<TYPE>::sm_pfnUnhandledExceptionFilterPrev;

///////////////////////////////



template <class TYPE>
CVMArray<TYPE>::CVMArray(DWORD dwReserveElements) {

    if (sm_pHead == NULL) {
        // устанавливаем наш глобальный фильтр необработанных исключений
        // при создании первого экземпляра класса
        sm_pfnUnhandledExceptionFilterPrev =
            SetUnhandledExceptionFilter(UnhandledExceptionFilter);
    }

    m_pNext = sm_pHead; // следующий узел был вверху списка
    sm_pHead = this;   // сейчас вверху списка находится этот узел
}

```

см. след. стр.

Рис. 25-1. продолжение

```
m_cbReserve = sizeof(TYPE) * dwReserveElements;

// резервируем регион для всего массива
m_pArray = (TYPE*) VirtualAlloc(NULL, m_cbReserve,
    MEM_RESERVE | MEM_TOP_DOWN, PAGE_READWRITE);
chASSERT(m_pArray != NULL);
}

///////////////////////////////



template <class TYPE>
CVMArray<TYPE>::~CVMArray() {

    // освобождаем регион массива (возвращаем всю переданную ему память)
    VirtualFree(m_pArray, 0, MEM_RELEASE);

    // удаляем этот объект из связанного списка
    CVMArray* p = sm_pHead;
    if (p == this) { // удаляем верхний узел
        sm_pHead = p->m_pNext;
    } else {

        BOOL fFound = FALSE;

        // проходим по списку сверху и модифицируем указатели
        for (; !fFound && (p->m_pNext != NULL); p = p->m_pNext) {
            if (p->m_pNext == this) {
                // узел, указывающий на нас, должен указывать на следующий узел
                p->m_pNext = p->m_pNext->m_pNext;
                break;
            }
        }
        chASSERT(fFound);
    }
}

///////////////////////////////



// предлагаемый по умолчанию механизм обработки нарушенений доступа,
// возникающих при попытках передачи физической памяти
template <class TYPE>
LONG CVMArray<TYPE>::OnAccessViolation(PVOID pvAddrTouched,
    BOOL fAttemptedRead, PEXCEPTION_POINTERS pep, BOOL fRetryUntilSuccessful) {

    BOOL fCommittedStorage = FALSE; // считаем, что передать память не удалось

    do {
        // пытаемся передать физическую память
        fCommittedStorage = (NULL != VirtualAlloc(pvAddrTouched,
            sizeof(TYPE), MEM_COMMIT, PAGE_READWRITE));
    }
```

Рис. 25-1. продолжение

```

// если передать память не удается, предлагаем пользователю освободить память
if (!fCommittedStorage && fRetryUntilSuccessful) {
    MessageBox(NULL,
        TEXT("Please close some other applications and Press OK."),
        TEXT("Insufficient Memory Available"), MB_ICONWARNING | MB_OK);
}
} while (!fCommittedStorage && fRetryUntilSuccessful);

// если память передана, пытаемся возобновить выполнение программы;
// в ином случае активизируем обработчик
return(fCommittedStorage
    ? EXCEPTION_CONTINUE_EXECUTION : EXCEPTION_EXECUTE_HANDLER);
}

///////////////////////////////



// фильтр, связываемый с отдельным объектом класса CVMArray
template <class TYPE>
LONG CVMArray<TYPE>::ExceptionFilter(PEXCEPTION_POINTERS pep,
    BOOL fRetryUntilSuccessful) {

    // по умолчанию пытаемся использовать другой фильтр (самый безопасный путь)
    LONG lDisposition = EXCEPTION_CONTINUE_SEARCH;

    // мы обрабатываем только нарушение доступа
    if (pep->ExceptionRecord->ExceptionCode != EXCEPTION_ACCESS_VIOLATION)
        return(lDisposition);

    // получаем адрес и информацию о попытке чтения/записи
    PVOID pvAddrTouched = (PVOID) pep->ExceptionRecord->ExceptionInformation[1];
    BOOL fAttemptedRead = (pep->ExceptionRecord->ExceptionInformation[0] == 0);

    // попадает ли полученный адрес в регион,
    // зарезервированный для этого VMArray?
    if ((m_pArray <= pvAddrTouched) &&
        (pvAddrTouched < ((PBYTE) m_pArray + m_cbReserve))) {

        // была попытка записи в наш массив, пытаемся исправить ситуацию
        lDisposition = OnAccessViolation(pvAddrTouched, fAttemptedRead,
            pep, fRetryUntilSuccessful);
    }
    return(lDisposition);
}

///////////////////////////////



// фильтр, связываемый со всеми объектами класса CVMArray
template <class TYPE>
LONG WINAPI CVMArray<TYPE>::UnhandledExceptionFilter(PEXCEPTION_POINTERS pep) {

```

см. след. стр.

Рис. 25-1. продолжение

```
// по умолчанию пытаемся использовать другой фильтр (самый безопасный путь)
LONG lDisposition = EXCEPTION_CONTINUE_SEARCH;

// мы обрабатываем только нарушение доступа
if (pep->ExceptionRecord->ExceptionCode == EXCEPTION_ACCESS_VIOLATION) {

    // проходим все узлы связанного списка
    for (CVMArray* p = sm_pHead; p != NULL; p = p->m_pNext) {

        // Интересуемся, может ли данный узел обработать исключение.
        // Примечание: исключение НАДО обработать, иначе процесс будет закрыт!
        lDisposition = p->ExceptionFilter(pep, TRUE);

        // если подходящий узел найден и он обработал исключение,
        // выходим из цикла
        if (lDisposition != EXCEPTION_CONTINUE_SEARCH)
            break;
    }
}

// если ни один узел не смог устраниить проблему,
// пытаемся использовать предыдущий фильтр исключений
if (lDisposition == EXCEPTION_CONTINUE_SEARCH)
    lDisposition = sm_pfnUnhandledExceptionFilterPrev(pep);

return(lDisposition);
}

/////////////////////////////// Конец файла ///////////////////////////////
```

Исключения C++ и структурные исключения

Разработчики часто спрашивают меня, что лучше использовать: SEH или исключения C++. Ответ на этот вопрос Вы найдете здесь.

Для начала позвольте напомнить, что SEH — механизм операционной системы, доступный в любом языке программирования, а исключения C++ поддерживаются только в C++. Создавая приложение на C++, Вы должны использовать средства именно этого языка, а не SEH. Причина в том, что исключения C++ — часть самого языка и его компилятор автоматически создает код, который вызывает деструкторы объектов и тем самым обеспечивает корректную очистку ресурсов.

Однако Вы должны иметь в виду, что компилятор Microsoft Visual C++ реализует обработку исключений C++ на основе SEH операционной системы. Например, когда Вы создаете C++-блок *try*, компилятор генерирует SEH-блок *_try*. C++-блок *catch* становится SEH-фильтром исключений, а код блока *catch* — кодом SEH-блока *_except*. По сути, обрабатывая C++-оператор *throw*, компилятор генерирует вызов Windows-функции *RaiseException*, и значение переменной, указанной в *throw*, передается этой функции как дополнительный аргумент.

Сказанное мной поясняет фрагмент кода, показанный ниже. Функция слева использует средства обработки исключений C++, а функция справа демонстрирует, как компилятор C++ создает соответствующие им SEH-эквиваленты.

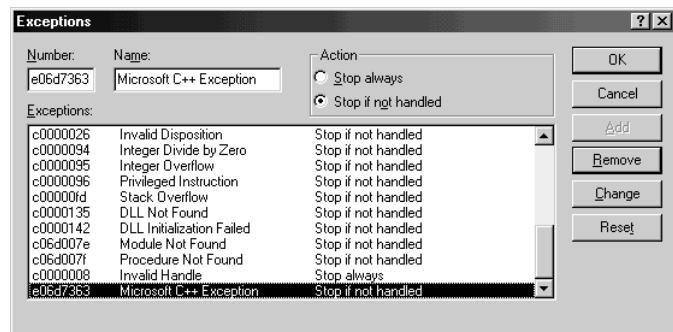
```

void ChunkyFunky() {
    try {
        // тело блока try
        :
        throw 5;
    }
    catch (int x) {
        // тело блока catch
        :
    }
}
}

void ChunkyFunky() {
    __try {
        // тело блока try
        :
        RaiseException(Code=0xE06D7363,
                       Flag=EXCEPTION_NONCONTINUABLE, Args=5);
    }
    __except ((ArgType == Integer) ?
              EXCEPTION_EXECUTE_HANDLER :
              EXCEPTION_CONTINUE_SEARCH) {
        // тело блока catch
        :
    }
}
}

```

Обратите внимание на несколько интересных особенностей этого кода. Во-первых, *RaiseException* вызывается с кодом исключения 0xE06D7363. Это код программного исключения, выбранный разработчиками Visual C++ на случай выталкивания (throwing) исключений C++. Вы можете сами в этом убедиться, открыв диалоговое окно Exceptions отладчика и прокрутив его список до конца, как на следующей иллюстрации.



Заметьте также, что при выталкивании исключения C++ всегда используется флаг EXCEPTION_NONCONTINUABLE. Исключения C++ не разрешают возобновлять выполнение программы, и возврат EXCEPTION_CONTINUE_EXECUTION фильтром, диагностирующим исключения C++, был бы ошибкой. Если Вы посмотрите на фильтр *_except* в функции справа, то увидите, что он возвращает только EXCEPTION_EXECUTE_HANDLER или EXCEPTION_CONTINUE_SEARCH.

Остальные аргументы *RaiseException* используются механизмом, который фактически выталкивает (throw) указанную переменную. Точный механизм того, как данные из переменной передаются *RaiseException*, не задокументирован, но догадаться о его реализации в компиляторе не так уж трудно.

И последнее, о чём хотелось бы сказать. Назначение фильтра *_except* — сравнивать тип *throw*-переменных с типом переменной, используемой в C++-операторе *catch*. Если их типы совпадают, фильтр возвращает EXCEPTION_EXECUTE_HANDLER, вызывая выполнение операторов в блоке *catch* (*_except*). А если они не совпадают, фильтр возвращает EXCEPTION_CONTINUE_SEARCH для проверки «вышестоящих» по дереву вызовов фильтров *catch*.



Так как исключения C++ реализуются через SEH, оба эти механизма можно использовать в одной программе. Например, я предпочитаю передавать физическую память при исключениях, вызываемых нарушениями доступа. Хотя C++ вообще не поддерживает этот тип обработки исключений (с возобновлением выполнения), он позволяет применять SEH в тех местах программы, где это нужно, и Ваш фильтр `_except` может возвращать `EXCEPTION_CONTINUE_EXECUTION`. Ну а в остальных частях исходного кода, где возобновление выполнения после обработки исключения не требуется, я пользуюсь механизмом обработки исключений, предлагаемым C++.

Перехват структурных исключений в C++

Обычно механизм обработки исключений в C++ не позволяет приложению восстановиться после таких серьезных исключений, как нарушение доступа или деление на нуль. Однако Microsoft добавила поддержку соответствующей функциональности в свой компилятор. Так, следующий код предотвратит аварийное завершение процесса.

```
void main() {
    try {
        * (PBYTE) 0 = 0; // нарушение доступа
    }
    catch (...) {
        // этот код обрабатывает исключения, связанные с нарушением доступа
    }
    // процесс завершается корректно
}
```

И это прекрасно, так как приложение может корректно справляться с серьезными исключениями. Но было бы еще лучше, если бы блок `catch` как-то различал коды исключений — чтобы мы могли писать, например, такой исходный код:

```
void Functastic() {

    try {
        * (PBYTE) 0 = 0; // нарушение доступа

        int x = 0;
        x = 5 / x; // деление на нуль
    }
    catch (StructuredException) {
        switch (StructuredExceptionCode) {
            case EXCEPTION_ACCESS_VIOLATION:
                // здесь обрабатывается нарушение доступа
                break;

            case EXCEPTION_INT_DIVIDE_BY_ZERO:
                // здесь обрабатывается деление на нуль
                break;

            default:
                // другие исключения мы не обрабатываем
                throw; // может, какой-нибудь другой блок catch
                      // обработает это исключение
        }
    }
}
```

```
        break; // никогда не выполняется  
    }  
}
```

Так вот, хочу Вас порадовать. В Visual C++ теперь возможно и такое. От Вас потребуется создать C++-класс, используемый специально для идентификации структурных исключений. Например:

```
#include <eh.h>           // для доступа к _set_se_translator
:
class CSE {
public:
    // вызовите эту функцию для каждого потока
    static void MapSEtoCE() { _set_se_translator(TranslateSEtoCE);
operator DWORD() { return(m_er.ExceptionCode); }

private:
    CSE(PEXCEPTION_POINTERS pep) {
        m_er      = *pep->ExceptionRecord;
        m_context = *pep->ContextRecord;
    }
    static void __cdecl TranslateSEtoCE(UINT dwEC,
        PEXCEPTION_POINTERS pep) {
        throw CSE(pep);
    }
};

private:
    EXCEPTION_RECORD m_er;          // машинно-независимая информация
    CONTEXT          m_context;     // машинно-зависимая информация
};
```

Внутри входных функций потоков вызывайте статическую функцию-член *MapSEtoCE*. В свою очередь она обращается к библиотечной С-функции *_set_se_translator*, передавая ей адрес функции *TranslateSEtoCE* класса CSE. Вызов *_set_se_translator* сообщает C++, что при возбуждении структурных исключений Вы хотите вызывать *TranslateSEtoCE*. Эта функция вызывает конструктор CSE-объекта и инициализирует два элемента данных машинно-зависимой и машинно-независимой информацией об исключении. Созданный таким образом CSE-объект может быть вытолкнут так же, как и любая другая переменная. И теперь Ваш C++-код способен обрабатывать структурные исключения, захватывая (catching) переменную этого типа.

Вот пример захвата такого C++-объекта:

```
void Functastic() {  
    CSE::MapSEtoCE(); // должна быть вызвана до возникновения исключений  
  
    try {  
        * (PBYTE) 0 = 0;      // нарушение доступа  
        int x = 0;  
        x = 5 / x;          // деление на нуль  
    }  
    catch (CSE se) {  
        switch (se) { // вызывает функцию-член оператора DWORD()  
    }
```

см. след. стр.

```
case EXCEPTION_ACCESS_VIOLATION:  
    // здесь обрабатывается исключение, вызванное нарушением доступа  
    break;  
  
case EXCEPTION_INT_DIVIDE_BY_ZERO:  
    // здесь обрабатывается исключение, вызванное делением на нуль  
    break;  
  
default:  
    // другие исключения мы не обрабатываем  
    throw;    // может, какой-нибудь другой блок catch  
              // обработает это исключение  
    break;    // никогда не выполняется  
}  
}  
}
```

ЧАСТЬ VI

ОПЕРАЦИИ С ОКНАМИ



Оконные сообщения

В этой главе я расскажу, как работает подсистема передачи сообщений в Windows применительно к приложениям с графическим пользовательским интерфейсом. Разрабатывая подсистему управления окнами в Windows 2000 и Windows 98, Microsoft преследовала две основные цели:

- обратная совместимость с 16-разрядной Windows, облегчающая перенос существующих 16-разрядных приложений;
- отказоустойчивость подсистемы управления окнами, чтобы ни один поток не мог нарушить работу других потоков в системе.

К сожалению, эти цели прямо противоречат друг другу. В 16-разрядной Windows передача сообщения в окно всегда осуществляется синхронно: отправитель не может продолжить работу, пока окно не обработает полученное сообщение. Обычно так и нужно. Но, если на обработку сообщения потребуется длительное время или если окно «зависнет», выполнение отправителя просто прекратится. А значит, такая операционная система не вправе претендовать на устойчивость к сбоям.

Это противоречие было серьезным вызовом для команды разработчиков из Microsoft. В итоге было выбрано компромиссное решение, отвечающее двум вышеупомянутым целям. Помните о них, читая эту главу, и Вы поймете, почему Microsoft сделала именно такой выбор.

Для начала рассмотрим некоторые базовые принципы. Один процесс в Windows может создать до 10 000 User-объектов различных типов — значков, курсоров, оконных классов, меню, таблиц клавиш-акселераторов и т. д. Когда поток из какого-либо процесса вызывает функцию, создающую один из этих объектов, последний переходит во владение процесса. Поэтому, если процесс завершается, не уничтожив данный объект явным образом, операционная система делает это за него. Однако два User-объекта (окна и ловушки) принадлежат только создавшему их потоку. И вновь, если поток создает окно или устанавливает ловушку, а потом завершается, операционная система автоматически уничтожает окно или удаляет ловушку.

Этот принцип принадлежности окон и ловушек создавшему их потоку оказывает существенное влияние на механизм функционирования окон: поток, создавший окно, должен обрабатывать все его сообщения. Поясню данный принцип на примере. Допустим, поток создал окно, а затем прекратил работу. Тогда его окно уже не получит сообщение WM_DESTROY или WM_NCDESTROY, потому что поток уже завершился и обрабатывать сообщения, посыпаемые этому окну, больше некому.

Это также означает, что каждому потоку, создавшему хотя бы одно окно, система выделяет очередь сообщений, используемую для их диспетчеризации. Чтобы окно в конечном счете получило эти сообщения, поток *должен* иметь собственный цикл выборки сообщений. В этой главе мы детально рассмотрим, что представляют собой

очереди сообщений потоков. В частности, я расскажу, как сообщения помещаются в эту очередь и как они извлекаются из нее, а потом обрабатываются.

Очередь сообщений потока

Как я уже говорил, одна из главных целей Windows — предоставить всем приложениям отказоустойчивую среду. Для этого любой поток должен выполняться в такой среде, где он может считать себя единственным. Точнее, у каждого потока должны быть очереди сообщений, полностью независимые от других потоков. Кроме того, для каждого потока нужно смоделировать среду, позволяющую ему самостоятельно управлять фокусом ввода с клавиатуры, активизировать окна, захватывать мышь и т. д.

Создавая какой-либо поток, система предполагает, что он не будет иметь отношения к поддержке пользовательского интерфейса. Это позволяет уменьшить объем выделяемых ему системных ресурсов. Но, как только поток обратится к той или иной GUI-функции (например, для проверки очереди сообщений или создания окна), система автоматически выделит ему дополнительные ресурсы, необходимые для выполнения задач, связанных с пользовательским интерфейсом. А если конкретнее, то система создает структуру `THREADINFO` и сопоставляет ее с этим потоком.

Элементы этой структуры используются, чтобы обмануть поток — заставить его считать, будто он выполняется в среде, принадлежащей только ему. `THREADINFO` — это внутренняя (недокументированная) структура, идентифицирующая очередь асинхронных сообщений потока (`posted-message queue`), очередь синхронных сообщений потока (`sent-message queue`), очередь ответных сообщений (`reply-message queue`), очередь виртуального ввода (`virtualized input queue`) и флаги пробуждения (`wake flags`); она также включает ряд других переменных-членов, характеризующих локальное состояние ввода для данного потока. На рис. 26-1 показаны структуры `THREADINFO`, сопоставленные с тремя потоками.

Структура `THREADINFO` — фундамент всей подсистемы передачи сообщений; читая следующие разделы, время от времени посмотрите на эту иллюстрацию.

Посылка асинхронных сообщений в очередь потока

Когда с потоком связывается структура `THREADINFO`, он получает свой набор очередей сообщений. Если процесс создает три потока и все они вызывают функцию `CreateWindow`, то и наборов очередей сообщений будет тоже три. Сообщения ставятся в очередь асинхронных сообщений вызовом функции `PostMessage`:

```
BOOL PostMessage(
    HWND hwnd,
    UINT uMsg,
    WPARAM wParam,
    LPARAM lParam);
```

При вызове этой функции система определяет, каким потоком создано окно, идентифицируемое параметром `hwnd`. Далее система выделяет блок памяти, сохраняет в нем параметры сообщения и записывает этот блок в очередь асинхронных сообщений данного потока. Кроме того, функция устанавливает флаг пробуждения `QS_POSTMESSAGE` (о нем — чуть позже). Возврат из `PostMessage` происходит сразу после того, как сообщение поставлено в очередь, поэтому вызывающий поток остается в неведении, обработано ли оно процедурой соответствующего окна. На самом деле вполне вероятно, что окно даже не получит это сообщение. Такое возможно, если поток, создавший это окно, завершится до того, как обработает все сообщения из своей очереди.

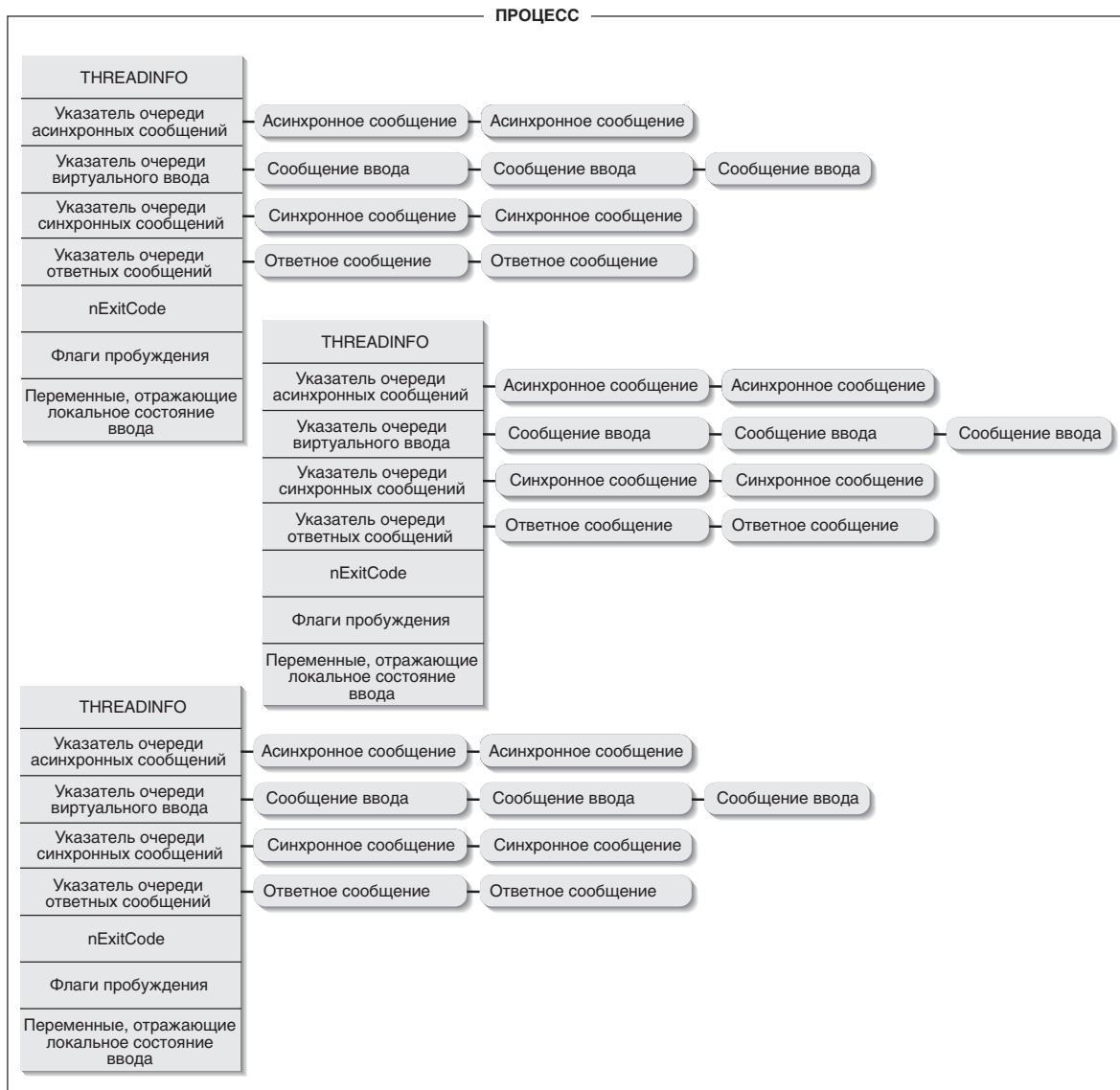


Рис. 26-1. Три потока и соответствующие им структуры THREADINFO

Сообщение можно поставить в очередь асинхронных сообщений потока и вызовом *PostThreadMessage*:

```
BOOL PostThreadMessage(
    DWORD dwThreadId,
    UINT uMsg,
    WPARAM wParam,
    LPARAM lParam);
```



Какой поток создал окно, можно определить с помощью *GetWindowThreadProcessId*:

```
DWORD GetWindowThreadProcessId(
    HWND hwnd,
    PDWORD pdwProcessId);
```

Она возвращает уникальный общесистемный идентификатор потока, который создал окно, определяемое параметром *hwnd*. Передав адрес переменной типа DWORD в параметре *pdwProcessId*, можно получить и уникальный общесистемный идентификатор процесса, которому принадлежит этот поток. Но обычно такой идентификатор не нужен, и мы просто передаем NULL.

Нужный поток идентифицируется первым параметром, *dwThreadId*. Когда сообщение помещено в очередь, элемент *hwnd* структуры MSG устанавливается как NULL. Применяется эта функция, когда приложение выполняет какую-то особую обработку в основном цикле выборки сообщений потока, — в этом случае он пишется так, чтобы после выборки сообщения функцией *GetMessage* (или *PeekMessage*) код в цикле сравнивал *hwnd* с NULL и, выполняя эту самую особую обработку, мог проверить значение элемента *msg* структуры MSG. Если поток определил, что сообщение не адресовано какому-либо окну, *DispatchMessage* не вызывается, и цикл переходит к выборке следующего сообщения.

Как и *PostMessage*, функция *PostThreadMessage* возвращает управление сразу после того, как сообщение поставлено в очередь потока. И вновь вызывающий поток остается в неведении о дальнейшей судьбе сообщения.

И, наконец, еще одна функция, позволяющая поместить сообщение в очередь асинхронных сообщений потока:

```
VOID PostQuitMessage(int nExitCode);
```

Она вызывается для того, чтобы завершить цикл выборки сообщений потока. Ее вызов аналогичен вызову:

```
PostThreadMessage(GetCurrentThreadId(), WM_QUIT, nExitCode, 0);
```

Но в действительности *PostQuitMessage* не помещает сообщение ни в одну из очередей структуры THREADINFO. Эта функция просто устанавливает флаг пробуждения QS_QUIT (о нем я тоже расскажу чуть позже) и элемент *nExitCode* структуры THREADINFO. Так как эти операции не могут вызвать ошибку, функция *PostQuitMessage* не возвращает никаких значений (VOID).

Посылка синхронных сообщений окну

Оконное сообщение можно отправить непосредственно оконной процедуре вызовом *SendMessage*:

```
LRESULT SendMessage(
    HWND hwnd,
    UINT uMsg,
    WPARAM wParam,
    LPARAM lParam);
```

Оконная процедура обработает сообщение, и только по окончании обработки функция *SendMessage* вернет управление. Благодаря этому ее используют гораздо чаще, чем *PostMessage* или *PostThreadMessage*. При переходе к выполнению следующей строки кода поток, вызвавший *SendMessage*, может быть уверен, что сообщение уже обработано.

Вот как работает *SendMessage*. Если поток вызывает *SendMessage* для посылки сообщения окну, созданному им же, то функция просто обращается к оконной процедуре соответствующего окна как к подпрограмме. Закончив обработку, оконная процедура передает функции *SendMessage* некое значение, а та возвращает его вызвавшему потоку.

Однако, если поток посыпает сообщение окну, созданному другим потоком, операции, выполняемые функцией *SendMessage*, значительно усложняются.¹ Windows требует, чтобы оконное сообщение обрабатывалось потоком, создавшим окно. Поэтому, если вызвать *SendMessage* для отправки сообщения окну, созданному в другом процессе и, естественно, другим потоком, Ваш поток не сможет обработать это сообщение — ведь он не работает в адресном пространстве чужого процесса, а потому не имеет доступа к коду и данным соответствующей оконной процедуры. И действительно, Ваш поток приостанавливается, пока другой поток обрабатывает сообщение. Поэтому, чтобы один поток мог отправить сообщение окну, созданному другим потоком, система должна выполнить следующие действия.

Во-первых, переданное сообщение присоединяется к очереди сообщений потока-приемника, в результате чего для этого потока устанавливается флаг `QS_SENDSMESSAGE`. Во-вторых, если поток-приемник в данный момент выполняет какой-то код и не ожидает сообщений (через вызов *GetMessage*, *PeekMessage* или *WaitMessage*), переданное сообщение обработать не удастся — система не прервет работу потока для немедленной обработки сообщения. Но когда поток-приемник ждет сообщений, система сначала проверяет, установлен ли флаг пробуждения `QS_SENDSMESSAGE`, и, если да, просматривает очередь синхронных сообщений, отыскивая первое из них. В очереди может находиться более одного сообщения. Скажем, несколько потоков одновременно послали сообщение одному и тому же окну. Тогда система просто ставит эти сообщения в очередь синхронных сообщений потока.

Итак, когда поток ждет сообщений, система извлекает из очереди синхронных сообщений первое и вызывает для его обработки нужную оконную процедуру. Если таких сообщений больше нет, флаг `QS_SENDSMESSAGE` сбрасывается. Пока поток-приемник обрабатывает сообщение, поток, отправивший сообщение через *SendMessage*, пристаивает, ожидая появления сообщения в очереди ответных сообщений. По окончании обработки значение, возвращенное оконной процедурой, передается асинхронно в очередь ответных сообщений потока-отправителя. Теперь он пробудится и извлечет упомянутое значение из ответного сообщения. Именно это значение и будет результатом вызова *SendMessage*. С этого момента поток-отправитель возобновляет работу в обычном режиме.

Ожидая возврата управления функцией *SendMessage*, поток в основном пристаивает. Но кое-чем он может заняться: если другой поток посыпает сообщение окну, созданному первым (ожидающим) потоком, система тут же обрабатывает это сообщение, не дожидаясь, когда поток вызовет *GetMessage*, *PeekMessage* или *WaitMessage*.

Поскольку Windows обрабатывает межпоточные сообщения описанным выше образом, Ваш поток может зависнуть. Допустим, в потоке, обрабатывающем синхронное сообщение, имеется «жучок», из-за которого поток входит в бесконечный цикл. Что же произойдет с потоком, вызвавшим *SendMessage*? Возобновится ли когда-нибудь его выполнение? Значит ли это, что ошибка в одном приложении «подвесит» другое? Ответ — да!

¹ Это верно даже в том случае, если оба потока принадлежат одному процессу.

Избегать подобных ситуаций позволяют четыре функции, и первая из них — *SendMessageTimeout*:

```
LRESULT SendMessageTimeout(
    HWND hwnd,
    UINT uMsg,
    WPARAM wParam,
    LPARAM lParam,
    UINT fuFlags,
    UINT uTimeout,
    PDWORD_PTR pdwResult);
```

Она позволяет задавать отрезок времени, в течение которого Вы готовы ждать ответа от другого потока на Ваше сообщение. Ее первые четыре параметра идентичны параметрам функции *SendMessage*. В параметре *fuFlags* можно передавать флаги *SMTO_NORMAL* (0), *SMTO_ABORTIFHUNG*, *SMTO_BLOCK*, *SMTO_NOTIMEOUTIFNOTHUNG* или комбинацию этих флагов.

Флаг *SMTO_ABORTIFHUNG* заставляет *SendMessageTimeout* проверить, не завис ли поток-приемник², и, если да, немедленно вернуть управление. Флаг *SMTO_NOTIMEOUTIFNOTHUNG* сообщает функции, что она должна игнорировать ограничение по времени, если поток-приемник не завис. Флаг *SMTO_BLOCK* предотвращает обработку вызывающим потоком любых других синхронных сообщений до возврата из *SendMessageTimeout*. Флаг *SMTO_NORMAL* определен в файле *WinUser.h* как 0; он используется в том случае, если Вы не указали другие флаги.

Я уже говорил, что ожидание потоком окончания обработки синхронного сообщения может быть прервано для обработки другого синхронного сообщения. Флаг *SMTO_BLOCK* предотвращает такое прерывание. Он применяется, только если поток, ожидая окончания обработки своего сообщения, не в состоянии обрабатывать прочие синхронные сообщения. Этот флаг иногда приводит к взаимной блокировке потоков до конца таймаута. Так, если Ваш поток отправит сообщение другому, а тому нужно послать сообщение Вашему, ни один из них не сможет продолжить обработку, и оба зависнут.

Параметр *uTimeout* определяет таймаут — время (в миллисекундах), в течение которого Вы готовы ждать ответного сообщения. При успешном выполнении функция возвращает TRUE, а результат обработки сообщения копируется по адресу, указанному в параметре *pdwResult*.

Кстати, прототип этой функции в заголовочном файле *WinUser.h* неверен. Функцию следовало бы определить как возвращающую значение типа BOOL, поскольку значение типа LRESULT на самом деле возвращается через ее параметр. Это создает определенные проблемы, так как *SendMessageTimeout* вернет FALSE, если Вы передадите неверный описатель окна или если закончится заданный период ожидания. Единственный способ узнать причину неудачного завершения функции — вызвать *GetLastError*. Последняя вернет 0 (ERROR_SUCCESS), если ошибка связана с окончанием периода ожидания. А если причина в неверном описателе, *GetLastError* даст код 1400 (ERROR_INVALID_WINDOW_HANDLE).

Если Вы обращаетесь к *SendMessageTimeout* для посылки сообщения окну, созданному вызывающим потоком, система просто вызывает оконную процедуру, помещая возвращаемое значение в *pdwResult*. Из-за этого код, расположенный за вызовом

² Операционная система считает поток зависшим, если он прекращает обработку сообщений более чем на 5 секунд.

SendMessageTimeout, не выполняется до тех пор, пока не заканчивается обработка сообщения, — ведь все эти операции осуществляются одним потоком.

Теперь рассмотрим вторую функцию, предназначенную для отправки межпоточных сообщений:

```
BOOL SendMessageCallback(
    HWND hwnd,
    UINT uMsg,
    WPARAM wParam,
    LPARAM lParam,
    SENDASYNCPROC pfnResultCallBack,
    ULONG_PTR dwData);
```

И вновь первые четыре параметра идентичны параметрам функции *SendMessage*. При вызове Вашим потоком *SendMessageCallback* отправляет сообщение в очередь синхронных сообщений потока-приемника и тут же возвращает управление вызывающему (т. е. Вашему) потоку. Закончив обработку сообщения, поток-приемник асинхронно отправляет свое сообщение в очередь ответных сообщений Вашего потока. Позже система уведомит Ваш поток об этом, вызвав написанную Вами функцию; у нее должен быть следующий прототип:

```
VOID CALLBACK ResultCallBack(
    HWND hwnd,
    UINT uMsg,
    ULONG_PTR dwData,
    LRESULT lResult);
```

Адрес этой функции обратного вызова передается *SendMessageCallback* в параметре *pfnResultCallBack*. А при вызове *ResultCallBack* в первых двух параметрах передаются описатель окна, закончившего обработку сообщения, и код (значение) самого сообщения. Параметр *dwData* функции *ResultCallBack* всегда получает значение, переданное *SendMessageCallback* в одноименном параметре. (Система просто берет то, что указано там, и передает Вашей функции *ResultCallBack*.) Последний параметр функции *ResultCallBack* сообщает результат обработки сообщения, полученный от оконной процедуры.

Поскольку *SendMessageCallback*, передавая сообщение другому потоку, немедленно возвращает управление, *ResultCallBack* вызывается после обработки сообщения потоком-приемником не сразу, а с задержкой. Сначала поток-приемник асинхронно ставит сообщение в очередь ответных сообщений потока-отправителя. Затем при первом же вызове потоком-отправителем любой из функций *GetMessage*, *PeekMessage*, *WaitMessage* или одной из *Send*-функций сообщение извлекается из очереди ответных сообщений, и лишь потом вызывается Ваша функция *ResultCallback*.

Существует и другое применение функции *SendMessageCallback*. В Windows предусмотрен метод, позволяющий разослать сообщение всем перекрывающимся окнам (overlapped windows) в системе; он состоит в том, что Вы вызываете *SendMessage* и в параметре *hwnd* передаете ей *HWND_BROADCAST* (определенный как -1). Этот метод годится только для широковещательной рассылки сообщений, возвращаемые значения которых Вас не интересуют, поскольку функция способна вернуть лишь одно значение, *LRESULT*. Но, используя *SendMessageCallback*, можно получить результаты обработки «широковещательного» сообщения от каждого перекрытого окна. Ваша функция *SendMessageCallback* будет вызываться с результатом обработки сообщения от каждого из таких окон.

Если *SendMessageCallback* вызывается для отправки сообщения окну, созданному вызывающим потоком, система немедленно вызывает оконную процедуру, а после обработки сообщения — функцию *ResultCallBack*. После возврата из *ResultCallback* выполнение начинается со строки, следующей за вызовом *SendMessageCallback*.

Третья функция, предназначенная для передачи межпоточных сообщений:

```
BOOL SendNotifyMessage(
    HWND hwnd,
    UINT uMsg,
    WPARAM wParam,
    LPARAM lParam);
```

Поместив сообщение в очередь синхронных сообщений потока-приемника, она немедленно возвращает управление вызывающему потоку. Так ведет себя и *PostMessage*, помните? Но два отличия *SendNotifyMessage* от *PostMessage* все же есть.

Во-первых, если *SendNotifyMessage* посыпает сообщение окну, созданному другим потоком, приоритет данного синхронного сообщения выше приоритета асинхронных сообщений, находящихся в очереди потока-приемника. Иными словами, сообщения, помещаемые в очередь с помощью *SendNotifyMessage*, всегда извлекаются до выборки сообщений, отправленных через *PostMessage*.

Во-вторых, если сообщение посыпается окну, созданному вызывающим потоком, *SendNotifyMessage* работает точно так же, как и *SendMessage*, т. е. не возвращает управление до окончания обработки сообщения.

Большинство синхронных сообщений посыпается окну для уведомления — чтобы сообщить ему об изменении состояния и чтобы оно как-то отреагировало на это, прежде чем Вы продолжите свою работу. Например, WM_ACTIVATE, WM_DESTROY, WM_ENABLE, WM_SIZE, WM_SETFOCUS, WM_MOVE и многие другие сообщения — это просто уведомления, посыпаемые системой окну в синхронном, а не асинхронном режиме. Поэтому система не прерывает свою работу только ради того, чтобы оконная процедура могла их обработать. Прямо противоположный эффект дает отправка сообщения WM_CREATE — тогда система ждет, когда окно закончит его обработку. Если возвращено значение –1, значит, окно не создано.

И, наконец, четвертая функция, связанная с обработкой межпоточных сообщений:

```
BOOL ReplyMessage(LRESULT lResult);
```

Она отличается от трех описанных выше. В то время как *Send*-функции используются посыпающим сообщения потоком для защиты себя от зависания, *ReplyMessage* вызывается потоком, принимающим оконное сообщение. Вызвав ее, поток как бы говорит системе, что он уже получил результат обработки сообщения и что этот результат нужно упаковать и асинхронно отправить в очередь ответных сообщений потока-отправителя. Последний сможет пробудиться, получить результат и возобновить работу.

Поток, вызывающий *ReplyMessage*, передает результат обработки сообщения через параметр *lResult*. После вызова *ReplyMessage* выполнение потока-отправителя возобновляется, а поток, занятый обработкой сообщения, продолжает эту обработку. Ни один из потоков не приостанавливается — оба работают, как обычно. Когда поток, обрабатывающий сообщение, выйдет из своей оконной процедуры, любое возвращенное значение просто игнорируется.

Заметьте: *ReplyMessage* надо вызывать из оконной процедуры, получившей сообщение, но не из потока, вызвавшего одну из *Send*-функций. Поэтому, чтобы написать «защищенный от зависаний» код, следует заменить все вызовы *SendMessage* вызовами

одной из трех *Send*-функций и не полагаться на то, что оконная процедура будет вызывать именно *ReplyMessage*.

Учтите также, что вызов *ReplyMessage* при обработке сообщения, посланного этим же потоком, не влечет никаких действий. На это и указывает значение, возвращаемое *ReplyMessage*: TRUE — при обработке межпоточного сообщения и FALSE — при попытке вызова функции для обработки внутрипоточного сообщения.

Если Вас интересует, является ли обрабатываемое сообщение внутрипоточным или межпоточным, вызовите функцию *InSendMessage*:

```
BOOL InSendMessage();
```

Имя этой функции не совсем точно соответствует тому, что она делает в действительности. На первый взгляд, функция должна возвращать TRUE, если поток обрабатывает синхронное сообщение, и FALSE — при обработке им асинхронного сообщения. Но это не так. Она возвращает TRUE, если поток обрабатывает межпоточное синхронное сообщение, и FALSE — при обработке им внутрипоточного сообщения (синхронного или асинхронного). Возвращаемые значения функций *InSendMessage* и *ReplyMessage* идентичны.

Есть еще одна функция, позволяющая определить тип сообщения, которое обрабатывается Вашей оконной процедурой:

```
DWORD InSendMessageEx(PVOID pvReserved);
```

Вызывая ее, Вы должны передать NULL в параметре *pvReserved*. Возвращаемое значение указывает на тип обрабатываемого сообщения. Значение ISMEX_NOSEND (0) говорит о том, что поток обрабатывает внутрипоточное синхронное или асинхронное сообщение. Остальные возвращаемые значения представляют собой комбинацию битовых флагов, описанных в следующей таблице.

Флаг	Описание
ISMEX_SEND	Поток обрабатывает межпоточное синхронное сообщение, посланное через <i>SendMessage</i> или <i>SendMessageTimeout</i> ; если флаг ISMEX_REPLYED не установлен, поток-отправитель блокируется в ожидании ответа
ISMEX_NOTIFY	Поток обрабатывает межпоточное синхронное сообщение, посланное через <i>SendNotifyMessage</i> ; поток-отправитель не ждет ответа и не блокируется
ISMEX_CALLBACK	Поток обрабатывает межпоточное синхронное сообщение, посланное через <i>SendMessageCallback</i> ; поток-отправитель не ждет ответа и не блокируется
ISMEX_REPLYED	Поток обрабатывает межпоточное синхронное сообщение и уже вызвал <i>ReplyMessage</i> ; поток-отправитель не блокируется

Пробуждение потока

Когда поток вызывает *GetMessage* или *WaitMessage* и никаких сообщений для него или созданных им окон нет, система может приостановить выполнение потока, и тогда он уже не получает процессорное время. Как только потоку будет отправлено синхронное или асинхронное сообщение, система установит флаг пробуждения, указывающий, что теперь поток должен получать процессорное время и обработать сообщение. Если пользователь ничего не набирает на клавиатуре и не трогает мышь, то обычно в таких обстоятельствах никаких сообщений окнам не посыпается. А это значит, что большинство потоков в системе не получает процессорное время.

Флаги состояния очереди

Во время выполнения поток может опросить состояние своих очередей вызовом *GetQueueStatus*:

```
DWORD GetQueueStatus(UINT fuFlags);
```

Параметр *fuFlags* — флаг или группа флагов, объединенных побитовой операцией OR; он позволяет проверить значения отдельных битов пробуждения (wake bits). Допустимые значения флагов и их смысл описаны в следующей таблице.

Флаг	Сообщение в очереди
QS_KEY	WM_KEYUP, WM_KEYDOWN, WM_SYSKEYUP или WM_SYSKEYDOWN
QS_MOUSEMOVE	WM_MOUSEMOVE
QS_MOUSEBUTTON	WM_?BUTTON* (где знак вопроса заменяет букву L, M или R, а звездочка — DOWN, UP или DBLCLK)
QS_MOUSE	То же, что QS_MOUSEMOVE QS_MOUSEBUTTON
QS_INPUT	То же, что QS_MOUSE QS_KEY
QS_PAINT	WM_PAINT
QS_TIMER	WM_TIMER
QS_HOTKEY	WM_HOTKEY
QS_POSTMESSAGE	Асинхронное сообщение (отличное от события аппаратного ввода); этот флаг идентичен QS_ALLPOSTMESSAGE с тем исключением, что сбрасывается при отсутствии асинхронных сообщений в диапазоне действия фильтра сообщений
QS_ALLPOSTMESSAGE	Асинхронное сообщение (отличное от события аппаратного ввода); этот флаг идентичен QS_POSTMESSAGE с тем исключением, что сбрасывается лишь при полном отсутствии каких-либо асинхронных сообщений (вне зависимости от фильтра сообщений)
QS_ALLEVENTS	То же, что QS_INPUT QS_POSTMESSAGE QS_TIMER QS_PAINT QS_HOTKEY
QS_QUIT	Сообщает о вызове <i>PostQuitMessage</i> ; этот флаг не задокументирован, его нет в WinUser.h, и он используется самой системой
QS_SENDDMESSAGE	Синхронное сообщение, посланное другим потоком
QS_ALLINPUT	То же, что QS_ALLEVENTS QS_SENDDMESSAGE

При вызове *GetQueueStatus* параметр *fuFlags* сообщает функции, наличие каких типов сообщений в очереди следует проверить. Чем меньше идентификаторов QS_* объединено побитовой операцией OR, тем быстрее отрабатывается вызов. Результат сообщается в старшем слове значения, возвращаемого функцией. Возвращаемый набор флагов всегда представляет собой подмножество того набора, который Вы запросили от функции. Например, если Вы делаете такой вызов:

```
BOOL fPaintMsgWaiting = HIWORD(GetQueueStatus(QS_TIMER)) & QS_PAINT;
```

то значение *fPaintMsgWaiting* всегда будет равно FALSE независимо от наличия в очереди сообщения WM_PAINT, так как флаг QS_PAINT функции не передан.

Младшее слово возвращаемого значения содержит типы сообщений, которые помещены в очередь, но не обработаны с момента последнего вызова *GetQueueStatus*, *GetMessage* или *PeekMessage*.

Не все флаги пробуждения обрабатываются системой одинаково. Флаг QS_MOUSEMOVE устанавливается, если в очереди есть необработанное сообщение WM_MOUSE-

MOVE. Когда *GetMessage* или *PeekMessage* (с флагом PM_REMOVE) извлекают последнее сообщение WM_MOUSEMOVE, флаг сбрасывается и остается в таком состоянии, пока в очереди ввода снова не окажется сообщение WM_MOUSEMOVE. Флаги QS_KEY, QS_MOUSEBUTTON и QS_HOTKEY действуют при соответствующих сообщениях аналогичным образом.

Флаг QS_PAINT обрабатывается иначе. Он устанавливается, если в окне, созданном данным потоком, имеется недействительная, требующая перерисовки область. Когда область, занятая всеми окнами, созданными одним потоком, становится действительной (обычно в результате вызова *ValidateRect*, *ValidateRegion* или *BeginPaint*), флаг QS_PAINT сбрасывается. Еще раз подчеркну: данный флаг сбрасывается, только если становятся действительными все окна, принадлежащие потоку. Вызов *GetMessage* или *PeekMessage* на этот флаг пробуждения не влияет.

Флаг QS_POSTMESSAGE устанавливается, когда в очереди асинхронных сообщений потока есть минимум одно сообщение. При этом не учитываются аппаратные сообщения, находящиеся в очереди виртуального ввода потока. Этот флаг сбрасывается после обработки всех сообщений из очереди асинхронных сообщений.

Флаг QS_TIMER устанавливается после срабатывания таймера (созданного потоком). После того как функция *GetMessage* или *PeekMessage* вернет WM_TIMER, флаг сбрасывается и остается в таком состоянии, пока таймер вновь не сработает.

Флаг QS_SENDSMESSAGE указывает, что сообщение появилось в очереди синхронных сообщений. Он используется системой для идентификации и обработки межпоточных синхронных сообщений, а для внутрипоточных синхронных сообщений не применяется. Вы можете указывать флаг QS_SENDSMESSAGE, но необходимость в нем возникает крайне редко. Я ни разу не видел его ни в одном приложении.

Есть еще один (недокументированный) флаг состояния очереди — QS_QUIT. Он устанавливается при вызове потоком *PostQuitMessage*. Сообщение WM_QUIT при этом не добавляется к очереди сообщений. И учтите, что *GetQueueStatus* не возвращает состояние этого флага.

Алгоритм выборки сообщений из очереди потока

Когда поток вызывает *GetMessage* или *PeekMessage*, система проверяет флаги состояния очередей потока и определяет, какое сообщение надо обработать (рис. 26-2).

1. Если флаг QS_SENDSMESSAGE установлен, система отправляет сообщение соответствующей оконной процедуре. *GetMessage* и *PeekMessage* контролируют процесс обработки и не передают управление потоку сразу после того, как оконная процедура обработает сообщение; вместо этого обе функции ждут следующего сообщения.
2. Если очередь асинхронных сообщений потока не пуста, *GetMessage* и *PeekMessage* заполняют переданную им структуру MSG и возвращают управление. Цикл выборки сообщений (расположенный в потоке) в этот момент обычно обращается к *DispatchMessage*, чтобы соответствующая оконная процедура обработала сообщение.
3. Если флаг QS_QUIT установлен, *GetMessage* и *PeekMessage* возвращают сообщение WM_QUIT (параметр *wParam* которого содержит указанный код завершения) и сбрасывают этот флаг.
4. Если в очереди виртуального ввода потока есть какие-то сообщения, *GetMessage* и *PeekMessage* возвращают сообщение, связанное с аппаратным вводом.

5. Если флаг QS_PAINT установлен, *GetMessage* и *PeekMessage* возвращают сообщение WM_PAINT для соответствующего окна.
6. Если флаг QS_TIMER установлен, *GetMessage* и *PeekMessage* возвращают сообщение WM_TIMER.

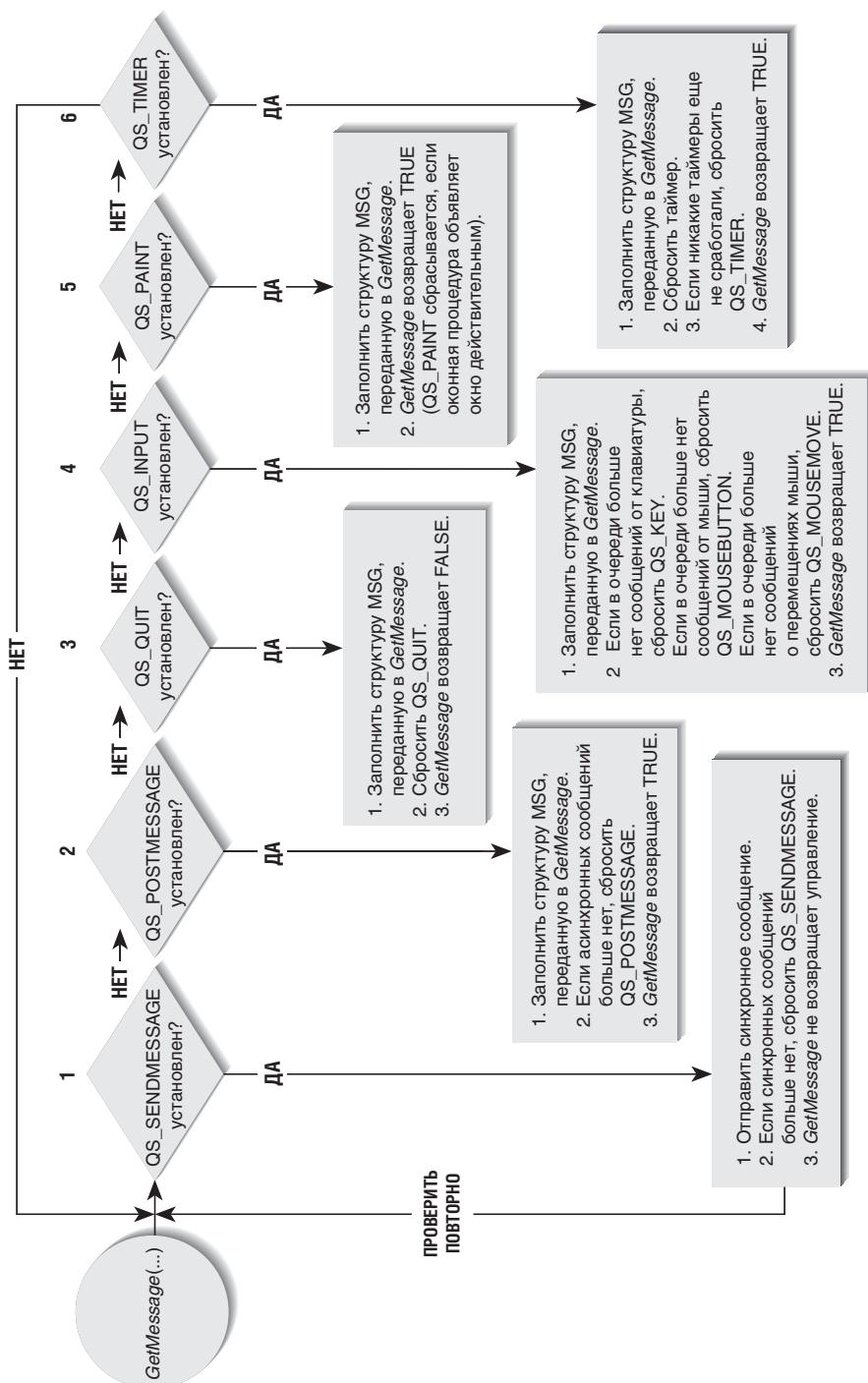


Рис. 26-2. Алгоритм выборки сообщений из очереди потока

Хоть и трудно в это поверить, но для такого безумия есть своя причина. Главное, из чего исходила Microsoft, разрабатывая описанный алгоритм, — приложения должны слушаться пользователя, и именно его действия (с клавиатурой и мышью) управляют программой, порождая события аппаратного ввода. Работая с программой, пользователь может нажать кнопку мыши, что приводит к генерации последовательности определенных событий. А программа порождает отдельные события, асинхронно отправляя сообщения в очередь потока.

Так, нажатие кнопки мыши могло бы заставить окно, которое обрабатывает сообщение WM_LBUTTONDOWN, послать три асинхронных сообщения разным окнам. Поскольку эти три программных события возникают в результате аппаратного события, система обрабатывает их до того, как принимает новое аппаратное событие, инициируемое пользователем. И именно поэтому очередь асинхронных сообщений проверяется раньше очереди виртуального ввода.

Прекрасный пример такой последовательности событий — вызов функции *TranslateMessage*, проверяющей, не было ли выбрано из очереди ввода сообщение WM_KEYDOWN или WM_SYSKEYDOWN. Если одно из этих сообщений выбрано, система проверяет, можно ли преобразовать информацию о виртуальной клавише в символьный эквивалент. Если это возможно, *TranslateMessage* вызывает *PostMessage*, чтобы поместить в очередь асинхронных сообщений WM_CHAR или WM_SYSCHAR. При следующем вызове *GetMessage* система проверяет содержимое очереди асинхронных сообщений и, если в ней есть сообщение, извлекает его и возвращает потоку. Возвращается либо WM_CHAR, либо WM_SYSCHAR. При следующем вызове *GetMessage* система обнаруживает, что очередь асинхронных сообщений пуста. Тогда она проверяет очередь ввода, где и находит сообщение WM_(SYS)KEYUP; именно оно и возвращается функцией *GetMessage*.

Поскольку система устроена так, а не иначе, последовательность аппаратных событий:

```
WM_KEYDOWN  
WM_KEYUP
```

генерирует следующую последовательность сообщений для оконной процедуры (при этом предполагается, что информацию о виртуальной клавише можно преобразовать в ее символьный эквивалент):

```
WM_KEYDOWN  
WM_CHAR  
WM_KEYUP
```

Вернемся к тому, как система решает, что за сообщение должна вернуть функция *GetMessage* или *PeekMessage*. Просмотрев очередь асинхронных сообщений, система, прежде чем перейти к проверке очереди виртуального ввода, проверяет флаг QS_QUIT. Вспомните: этот флаг устанавливается, когда поток вызывает *PostQuitMessage*. Вызов *PostQuitMessage* дает примерно тот же эффект, что и вызов *PostMessage*, которая помещает сообщение в конец очереди и тем самым заставляет обрабатывать его до проверки очереди ввода. Так почему же *PostQuitMessage* устанавливает флаг вместо того, чтобы поместить WM_QUIT в очередь сообщений? На то есть две причины.

Во-первых, в условиях нехватки памяти может получиться так, что асинхронное сообщение не удастся поместить в очередь. Но, если приложение хочет завершиться, оно должно завершиться — тем более при нехватке памяти. Вторая причина в том, что этот флаг позволяет потоку закончить обработку остальных асинхронных сообщений до завершения его цикла выборки сообщений. Поэтому в следующем фрагмен-

те кода сообщение WM_USER будет извлечено до WM_QUIT, даже если WM_USER асинхронно помещено в очередь после вызова *PostQuitMessage*.

```
case WM_CLOSE:
    PostQuitMessage(0);
    PostMessage(hwnd, WM_USER, 0, 0);
```

А теперь о последних двух сообщениях: WM_PAINT и WM_TIMER. Сообщение WM_PAINT имеет низкий приоритет, так как прорисовка экрана — операция не самая быстрая. Если бы это сообщение посыпалось всякий раз, когда окно становится недействительным, быстродействие системы снизилось бы весьма ощутимо. Но помещая WM_PAINT после ввода с клавиатуры, система работает гораздо быстрее. Например, из меню можно вызвать какую-нибудь команду, открывающую диалоговое окно, выбрать в нем что-то, нажать клавишу Enter — и проделать все это даже до того, как окно появится на экране. Достаточно быстро нажимая клавиши, Вы наверняка заметите, что сообщения об их нажатии извлекаются прежде, чем дело доходит до сообщений WM_PAINT. А когда Вы нажимаете клавишу Enter, подтверждая тем самым значения параметров, указанных в диалоговом окне, система разрушает окно и сбрасывает флаг QS_PAINT.

Приоритет WM_TIMER еще ниже, чем WM_PAINT. Почему? Допустим, какая-то программа обновляет свое окно всякий раз, когда получает сообщение WM_TIMER. Если бы оно поступало слишком часто, программа просто не смогла бы обновлять свое окно. Но поскольку сообщения WM_PAINT обрабатываются до WM_TIMER, такая проблема не возникает.



Функции *GetMessage* и *PeekMessage* проверяют флаги пробуждения только для вызывающего потока. Это значит, что потоки никогда не смогут извлечь сообщения из очереди, присоединенной к другому потоку, включая сообщения для потоков того же процесса.

Пробуждение потока с использованием объектов ядра или флагов состояния очереди

Функции *GetMessage* и *PeekMessage* приостанавливают поток до тех пор, пока ему не понадобится выполнить какую-нибудь задачу, связанную с пользовательским интерфейсом. Иногда то же самое было бы удобно и при обработке других задач. Для этого поток должен как-то узнавать о завершении операции, не относящейся к пользовательскому интерфейсу.

Чтобы поток ждал собственных сообщений, вызовите функцию *MsgWaitForMultipleObjects* или *MsgWaitForMultipleObjectsEx*:

```
DWORD MsgWaitForMultipleObjects(
    DWORD nCount,
    PHANDLE phObjects,
    BOOL fWaitAll,
    DWORD dwMilliseconds,
    DWORD dwWakeMask);

DWORD MsgWaitForMultipleObjectsEx(
    DWORD nCount,
    PHANDLE phObjects,
    DWORD dwMilliseconds,
```

см. след. стр.

```
DWORD dwWakeMask,  
DWORD dwFlags);
```

Эти функции аналогичны *WaitForMultipleObjects* (см. главу 9). Разница в том, что при их использовании поток становится планируемым, когда освобождается какой-нибудь из указанных объектов ядра или когда оконное сообщение нужно переслать окну, созданному этим потоком.

Внутренне система просто добавляет объект «событие» в массив описателей ядра. Параметр *dwWakeMask* сообщает системе, в какой момент объект-событие должно переходить в свободное состояние. Его допустимые значения идентичны тем, которые можно передавать в функцию *GetQueueStatus*.

WaitForMultipleObjects обычно возвращает индекс освобожденного объекта (в диапазоне от *WAIT_OBJECT_0* до *WAIT_OBJECT_0 + nCount - 1*). Задание параметра *dwWakeMask* равносильно добавлению еще одного описателя. При выполнении условия, определенного маской пробуждения, *MsgWaitForMultipleObjects(Ex)* возвращает значение *WAIT_OBJECT_0 + nCount*.

Вот пример вызова *MsgWaitForMultipleObjects*:

```
MsgWaitForMultipleObjects(0, NULL, TRUE, INFINITE, QS_INPUT);
```

Описатели синхронизирующих объектов в этом операторе не передаются — параметры *nCount* и *phObjects* равны соответственно 0 и NULL. Мы указываем функции ждать освобождения всех объектов. Но в действительности задан лишь один объект, и с тем же успехом параметру *fWaitAll* можно было бы присвоить значение FALSE. Мы также сообщаем, что будем ждать — сколько бы времени это ни потребовало — появления в очереди ввода потока сообщения от клавиатуры или мыши.

Начав пользоваться функцией *MsgWaitForMultipleObjects* в своих программах, Вы быстро поймете, что она лишена многих важных качеств. Вот почему Microsoft пришлось создать более совершенную функцию *MsgWaitForMultipleObjectsEx*, которая позволяет задать в параметре *dwFlags* любую комбинацию следующих флагов.

Флаг	Описание
MWMO_WAITALL	Функция ждет освобождения всех объектов ядра и появления в очереди потока указанных сообщений (без этого флага функция ждет освобождения одного из объектов ядра или появления в очереди одного из указанных сообщений)
MWMO_ALERTABLE	Функция ждет в «тревожном» состоянии
MWMO_INPUTAVAILABLE	Функция ждет появления в очереди потока одного из указанных сообщений

Если Вам не нужны эти дополнительные возможности, передайте в *dwFlags* нулевое значение.

При использовании *MsgWaitForMultipleObjects(Ex)* учитывайте, что:

- эти функции лишь включают описатель внутреннего объекта ядра «событие» в массив описателей объектов ядра, и значение параметра *nCount* не должно превышать 63 (*MAXIMUM_WAIT_OBJECTS - 1*);
- если в параметре *fWaitAll* передается FALSE, функции возвращают управление при освобождении объекта ядра *или* при появлении в очереди потока сообщения заданного типа;
- если в параметре *fWaitAll* передается TRUE, функции возвращают управление при освобождении всех объектов ядра *и* появлении в очереди потока сообще-

ния заданного типа. Такое поведение этих функций преподносит сюрприз многим разработчикам. Ведь очень часто поток надо пробуждать при освобождении всех объектов ядра *или* при появлении сообщения указанного типа. Но функции, действующей именно так, нет;

- при вызове любая из этих функций на самом деле проверяет в очереди потока только *новые* сообщения заданного типа.

Заметьте, что и последняя особенность этих функций — не очень приятный сюрприз для многих разработчиков. Возьмем простой пример. Допустим, в очереди потока находятся два сообщения о нажатии клавиш. Если теперь вызвать *MsgWaitForMultipleObjectsEx* и задать в *dwWakeMask* значение *QS_INPUT*, поток пробудится, извлечет из очереди первое сообщение и обработает его. Но на повторный вызов *MsgWaitForMultipleObjectsEx* поток никак не отреагирует — ведь *новых* сообщений в очереди нет.

Этот механизм создал столько проблем разработчикам, что Microsoft пришлось добавить в *MsgWaitForMultipleObjectsEx* поддержку флага *MWMO_INPUTAVAILABLE*.

Вот как надо писать цикл выборки сообщений при использовании *MsgWaitForMultipleObjectsEx*:

```
BOOL fQuit = FALSE; // надо ли завершить цикл?

while (!fQuit) {

    // поток пробуждается при освобождении объекта ядра ИЛИ
    // для обработки сообщения от пользовательского интерфейса
    DWORD dwResult = MsgWaitForMultipleObjectsEx(1, &hEvent,
                                                INFINITE, QS_ALLEVENTS, MWMO_INPUTAVAILABLE);

    switch (dwResult) {
        case WAIT_OBJECT_0:           // освободилось событие
            break;

        case WAIT_OBJECT_0 + 1:       // в очереди появилось сообщение
            // разослать все сообщения
            MSG msg;

            while (PeekMessage(&msg, NULL, 0, 0, PM_REMOVE)) {
                if (msg.message == WM_QUIT) {
                    // сообщение WM_QUIT - выходим из цикла
                    fQuit = TRUE;
                } else {
                    // транслируем и пересыпаем сообщение
                    TranslateMessage(&msg);
                    DispatchMessage(&msg);
                }
            } // наша очередь пуста
            break;
    }
} // конец цикла while
```

Передача данных через сообщения

Здесь мы обсудим, как система обеспечивает передачу данных между процессами с помощью сообщений. В некоторых оконных сообщениях параметр *lParam* задает адрес блока памяти. Например, сообщение WM_SETTEXT использует *lParam* как указатель на строку (с нулевым символом в конце), содержащую новый текст для окна. Рассмотрим такой вызов:

```
SendMessage(FindWindow(NULL, "Calculator"), WM_SETTEXT,  
0, (LPARAM) "A Test Caption");
```

Вроде бы все достаточно безобидно: определяется описатель окна Calculator и делается попытка изменить его заголовок на «A Test Caption». Но приглядимся к тому, что тут происходит.

В *lParam* передается адрес строки (с новым заголовком), расположенной в адресном пространстве Вашего процесса. Получив это сообщение, оконная процедура программы Calculator берет *lParam* и пытается манипулировать чем-то, что, «по ее мнению», является указателем на строку с новым заголовком.

Но адрес в *lParam* указывает на строку в адресном пространстве Вашего процесса, а не программы Calculator. Вот Вам и долгожданная неприятность — нарушение доступа к памяти. Но если Вы все же выполните показанную ранее строку, все будет работать нормально. Что за наваждение?

А дело в том, что система отслеживает сообщения WM_SETTEXT и обрабатывает их не так, как большинство других сообщений. При вызове *SendMessage* внутренний код функции проверяет, не пытаетесь ли Вы послать сообщение WM_SETTEXT. Если это так, функция копирует строку из Вашего адресного пространства в проекцию файла и делает его доступным другому процессу. Затем сообщение посыпается потоку другого процесса. Когда поток-приемник готов к обработке WM_SETTEXT, он определяет адрес общей проекции файла (содержащей копию строки) в адресном пространстве своего процесса. Параметру *lParam* присваивается значение именно этого адреса, и WM_SETTEXT направляется нужной оконной процедуре. После обработки этого сообщения, проекция файла уничтожается. Не слишком ли тут накрученено, а?

К счастью, большинство сообщений не требует такой обработки — она осуществляется, только если сообщение посыпается другому процессу. (Заметьте: описанная обработка выполняется и для любого сообщения, параметры *wParam* или *lParam* которого содержат указатель на какую-либо структуру данных.)

А вот другой случай, когда от системы требуется особая обработка, — сообщение WM_GETTEXT. Допустим, Ваша программа содержит код:

```
char szBuf[200];  
SendMessage(FindWindow(NULL, "Calculator"), WM_GETTEXT,  
Sizeof(szBuf), (LPARAM) szBuf);
```

WM_GETTEXT требует, чтобы оконная процедура программы Calculator поместила в буфер, на который указывает *szBuf*, заголовок своего окна. Когда Вы посыпаете это сообщение окну другого процесса, система должна на самом деле послать два сообщения. Сначала — WM_GETTEXTLENGTH. Оконная процедура возвращает число символов в строке заголовка окна. Это значение система использует при создании проекции файла, разделяемой двумя процессами.

Создав проекцию файла, система посыпает для его заполнения сообщение WM_GETTEXT. Затем переключается обратно на процесс, первым вызвавший функцию *SendMes-*

sage, копирует данные из общей проекции файла в буфер, на который указывает *szBuf*, и заставляет *SendMessage* вернуть управление.

Что ж, все хорошо, пока Вы посыпаете сообщения, известные системе. А если мы определим собственное сообщение (*WM_USER + x*), собираясь отправить его окну другого процесса? Система не «поймет», что нам нужна общая проекция файла для корректировки указателей при их пересылке. Но выход есть — это сообщение *WM_COPYDATA*:

```
COPYDATASTRUCT cds;
SendMessage(hwndReceiver, WM_COPYDATA, (WPARAM) hwndSender, (LPARAM) &cds);
```

COPYDATASTRUCT — структура, определенная в *WinUser.h*:

```
typedef struct tagCOPYDATASTRUCT {
    ULONG_PTR dwData;
    DWORD cbData;
    PVOID lpData;
} COPYDATASTRUCT;
```

Чтобы переслать данные окну другого процесса, нужно сначала инициализировать эту структуру. Элемент *dwData* резервируется для использования в Вашей программе. В него разрешается записывать любое значение. Например, передавая в другой процесс данные, в этом элементе можно указывать тип данных.

Элемент *cbData* задает число байтов, пересылаемых в другой процесс, а *lpData* указывает на первый байт данных. Адрес, идентифицируемый элементом *lpData*, находится, конечно же, в адресном пространстве отправителя.

Увидев, что Вы посыпаете сообщение *WM_COPYDATA*, *SendMessage* создает проекцию файла размером *cbData* байтов и копирует данные из адресного пространства Вашей программы в эту проекцию. Затем отправляет сообщение окну-приемнику. При обработке этого сообщения принимающей оконной процедурой параметр *lParam* указывает на структуру *COPYDATASTRUCT*, которая находится в адресном пространстве процесса-приемника. Элемент *lpData* этой структуры указывает на проекцию файла в адресном пространстве процесса-приемника.

Вам следует помнить о трех важных вещах, связанных с сообщением *WM_COPYDATA*.

- Отправляйте его всегда синхронно; никогда не пытайтесь делать этого асинхронно. Последнее просто невозможно: как только принимающая оконная процедура обработает сообщение, система должна освободить проекцию файла. При передаче *WM_COPYDATA* как асинхронного сообщения появится неопределенность в том, когда оно будет обработано, и система не сможет освободить память, занятую проекцией файла.
- На создание копии данных в адресном пространстве другого процесса неизбежно уходит какое-то время. Значит, пока *SendMessage* не вернет управление, нельзя допускать изменения содержимого общей проекции файла каким-либо другим потоком.
- Сообщение *WM_COPYDATA* позволяет 16-разрядным приложениям взаимодействовать с 32-разрядными (и наоборот), как впрочем и 32-разрядным — с 64-разрядными (и наоборот). Это удивительно простой способ общения между новыми и старыми приложениями. К тому же, *WM_COPYDATA* полностью поддерживается как в Windows 2000, так и в Windows 98. Но, если Вы все еще пишете 16-разрядные Windows-приложения, учтите, что сообщение *WM_COPY-*

DATA и структура COPYDATASTRUCT в Microsoft Visual C++ версии 1.52 не определены. Вам придется добавить их определения самостоятельно:

```
// включите этот код в свою 16-разрядную Windows-программу
#define WM_COPYDATA 0x004A

typedef VOID FAR* PVOID;
typedef struct tagCOPYDATASTRUCT {
    DWORD dwData;
    DWORD cbData;
    PVOID lpData;
} COPYDATASTRUCT, FAR* PCOPYDATASTRUCT;
```

Сообщение WM_COPYDATA — мощный инструмент, позволяющий разработчикам экономить массу времени при решении проблем связи между процессами. И очень жаль, что применяется оно нечасто. Насколько полезно это сообщение, иллюстрирует программа-пример LastMsgBoxInfo из главы 22.

Программа-пример CopyData

Эта программа, «26 CopyData.exe» (см. листинг на рис. 26-3), демонстрирует применение сообщения WM_COPYDATA при пересылке блока данных из одной программы в другую. Файлы исходного кода и ресурсов этой программы находятся в каталоге 26-CopyData на компакт-диске, прилагаемом к книге. Чтобы увидеть программу CopyData в действии, запустите минимум две ее копии; при этом каждая копия открывает диалоговое окно, показанное ниже.



Если Вы хотите посмотреть, как данные копируются из одного приложения в другое, то сначала измените содержимое полей Data1 и Data2. Затем щелкните одну из двух кнопок Send Data* To Other Windows. Программа отправит данные всем выполняемым экземплярам CopyData, и в их полях появятся новые данные.

А теперь обсудим принцип работы программы. Щелчок одной из двух кнопок приводит к:

1. Инициализации элемента *dwData* структуры COPYDATASTRUCT нулевым значением (если выбрана кнопка Send Data1 To Other Windows) или единицей (если выбрана кнопка Send Data2 To Other Windows).
2. Подсчету длины текстовой строки (в символах) из соответствующего поля с добавлением единицы, чтобы учесть нулевой символ в конце. Полученное число символов преобразуется в количество байтов умножением на *sizeof(TCHAR)*, и результат записывается в элемент *cbData* структуры COPYDATASTRUCT.
3. Вызову *_alloc*, чтобы выделить блок памяти, достаточный для хранения строки с учетом концевого нулевого символа. Адрес этого блока записывается в элемент *lpData* все той же структуры.
4. Копированию текста из поля в выделенный блок памяти.

Теперь все готово для пересылки в другие окна. Чтобы определить, каким окнам следует посыпать сообщение WM_COPYDATA, программа вызывает *FindWindowEx* и передает заголовок своего диалогового окна — благодаря этому перечисляются только другие экземпляры данной программы. Найдя окна всех экземпляров, программа пересыпает им сообщение WM_COPYDATA, что заставляет их обновить содержимое своих полей.



CopyData.cpp

```

 ****
 Модуль: CopyData.cpp
 Автор: Copyright (c) 2000, Джейфри Рихтер (Jeffrey Richter)
 ****

#include "..\CmnHdr.h"      /* см. приложение A */
#include <windowsx.h>
#include <tchar.h>
#include <malloc.h>
#include "Resource.h"

////////

// в WindowsX.h нет прототипа Cls_OnCopyData, поэтому определяем его здесь
/* BOOL Cls_OnCopyData(HWND hwnd, HWND hwndFrom, PCOPYDATASTRUCT pcds) */

////////

BOOL Dlg_OnCopyData(HWND hwnd, HWND hwndFrom, PCOPYDATASTRUCT cds) {
    Edit_SetText(GetDlgItem(hwnd, cds->dwData ? IDC_DATA2 : IDC_DATA1),
                 (PTSTR) cds->lpData);

    return(TRUE);
}

////////

BOOL Dlg_OnInitDialog(HWND hwnd, HWND hwndFocus, LPARAM lParam) {
    chSETDLGICONS(hwnd, IDI_COPYDATA);

    // инициализируем поля тестовыми данными
    Edit_SetText(GetDlgItem(hwnd, IDC_DATA1), TEXT("Some test data"));
    Edit_SetText(GetDlgItem(hwnd, IDC_DATA2), TEXT("Some more test data"));
    return(TRUE);
}

////////

void Dlg_OnCommand(HWND hwnd, int id, HWND hwndCtl, UINT codeNotify) {

```

Рис. 26-3. Программа-пример *CopyData*

см. след. стр.

Рис. 26-3. продолжение

```

switch (id) {
    case IDCANCEL:
        EndDialog(hwnd, id);
        break;

    case IDC_COPYDATA1:
    case IDC_COPYDATA2:
        if (codeNotify != BN_CLICKED)
            break;

        HWND hwndEdit = GetDlgItem(hwnd,
            (id == IDC_COPYDATA1) ? IDC_DATA1 : IDC_DATA2);

        // создаем экземпляр структуры COPYDATASTRUCT
        COPYDATASTRUCT cds;

        // указываем, из какого поля мы пересылаем данные (0=ID_DATA1, 1=ID_DATA2)
        cds.dwData = (DWORD) ((id == IDC_COPYDATA1) ? 0 : 1);

        // получаем длину пересылаемого блока данных (в байтах)
        cds.cbData = (Edit_GetTextLength(hwndEdit) + 1) * sizeof(TCHAR);

        // выделяем блок памяти для хранения строки
        cds.lpData = _alloca(cds.cbData);

        // копируем текст из поля ввода в выделенный блок
        Edit_GetText(hwndEdit, (PTSTR) cds.lpData, cds.cbData);

        // получаем заголовок нашего окна
        TCHAR szCaption[100];
        GetWindowText(hwnd, szCaption, chDIMOF(szCaption));

        // перечисляем все окна верхнего уровня с тем же заголовком
        HWND hWndT = NULL;
        do {
            hWndT = FindWindowEx(NULL, hWndT, NULL, szCaption);
            if (hWndT != NULL) {
                FORWARD_WM_COPYDATA(hWndT, hwnd, &cds, SendMessage);
            }
        } while (hWndT != NULL);
        break;
    }

///////////////////////////////
INT_PTR WINAPI Dlg_Proc(HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam) {

    switch (uMsg) {
        chHANDLE_DLGMMSG(hwnd, WM_INITDIALOG, Dlg_OnInitDialog);
        chHANDLE_DLGMMSG(hwnd, WM_COMMAND, Dlg_OnCommand);
        chHANDLE_DLGMMSG(hwnd, WM_COPYDATA, Dlg_OnCopyData);
    }
}

```

Рис. 26-3. продолжение

```

    }
    return(FALSE);
}
/////////////////////////////// Конец файла ///////////////////////////////
int WINAPI _tWinMain(HINSTANCE hinstExe, HINSTANCE, PTSTR pszCmdLine, int) {
    DialogBox(hinstExe, MAKEINTRESOURCE(IDD_COPYDATA), NULL, Dlg_Proc);
    return(0);
}
/////////////////////////////// Конец файла ///////////////////////////////

```

Как Windows манипулирует с ANSI/Unicode-символами и строками

WINDOWS 98 Windows 98 поддерживает классы и процедуры окон только в формате ANSI.

Регистрируя новый класс окна, Вы должны сообщить системе адрес оконной процедуры, которая отвечает за обработку сообщений для этого класса. В некоторых сообщениях (например, WM_SETTEXT) параметр *lParam* является указателем на строку. Для корректной обработки сообщения система должна заранее знать, в каком формате оконная процедура принимает строки — ANSI или Unicode.

Выбирая конкретную функцию для регистрации класса окна, Вы сообщаете системе формат, приемлемый для Вашей оконной процедуры. Если Вы создаете структуру WNDCLASS и вызываете *RegisterClassA*, система считает, что процедура ожидает исключительно ANSI-строки и символы. А регистрация класса окна через *RegisterClassW* заставит систему полагать, что процедуре нужен Unicode. И, конечно же, в зависимости от того, определен ли UNICODE при компиляции модуля исходного кода, макрос *RegisterClass* будет раскрыт либо в *RegisterClassA*, либо в *RegisterClassW*.

Располагая описателем окна, Вы можете выяснить, какой формат символов и строк требует оконная процедура. Для этого вызовите функцию:

```
BOOL IsWindowUnicode(HWND hwnd);
```

Если оконная процедура ожидает передачи данных только в Unicode, эта функция возвращает TRUE; в ином случае — FALSE.

Если Вы сформировали ANSI-строку и посыпаете сообщение WM_SETTEXT окну, чья процедура принимает только Unicode-строки, то система перед отсылкой сообщения автоматически преобразует его в нужный формат. Так что необходимость в вызове *IsWindowUnicode* возникает нечасто.

Система автоматически выполняет все преобразования и при создании подкласса окна. Допустим, что для заполнения своего поля ввода оконная процедура ожидает передачи символов и строк в Unicode. Кроме того, где-то в программе Вы создаете поле ввода и подкласс оконной процедуры, вызывая:

```
LONG_PTR SetWindowLongPtrA(
    HWND hwnd,
    int nIndex,
    LONG_PTR dwNewLong);
```

или

```
LONG_PTR SetWindowLongPtrW(
    HWND hwnd,
    int nIndex,
    LONG_PTR dwNewLong);
```

При этом Вы передаете в параметре *nIndex* значение GCLP_WNDPROC, а в параметре *dwNewLong* — адрес своей процедуры подкласса. Но что будет, если Ваша процедура ожидает передачи символов и строк в формате ANSI? В принципе, это чревато проблемами. Система определяет, как преобразовывать строки и символы в зависимости от функции, вызванной Вами для создания подкласса. Используя *SetWindowLongPtrA*, Вы сообщаете Windows, что новая оконная процедура (Вашего подкласса) принимает строки и символы только в ANSI. (Вызвав *IsWindowUnicode* после *SetWindowLongPtrA*, Вы получили бы FALSE, так как новая процедура не принимает строки и символы в Unicode.)

Но теперь у нас новая проблема: как сделать так, чтобы исходная процедура получала символы и строки в своем формате? Для корректного преобразования системе нужно знать две вещи. Во-первых, текущий формат символов и строк. Эту информацию мы предоставляем, вызывая одну из двух функций — *CallWindowProcA* или *CallWindowProcW*:

```
LRESULT CallWindowProcA(
    WNDPROC wndprcPrev,
    HWND hwnd,
    UINT uMsg,
    WPARAM wParam,
    LPARAM lParam);

LRESULT CallWindowProcW(
    WNDPROC wndprcPrev,
    HWND hwnd,
    UINT uMsg,
    WPARAM wParam,
    LPARAM lParam);
```

При передаче исходной оконной процедуре ANSI-строк процедура подкласса должна вызывать *CallWindowProcA*, а при передаче Unicode-строк — *CallWindowProcW*.

Второе, о чем должна знать система, — тип символов и строк, ожидаемый исходной оконной процедурой. Система получает эту информацию по адресу этой процедуры. Когда Вы вызываете *SetWindowLongPtrA* или *SetWindowLongPtrW*, система проверяет, создаете ли Вы ANSI-подкласс Unicode-процедуры окна или наоборот. Если при создании подкласса тип строк не меняется, *SetWindowLongPtr* просто возвращает адрес исходной процедуры. В ином случае *SetWindowLongPtr* вместо этого адреса возвращает описатель внутренней структуры данных.

Эта структура содержит адрес исходной оконной процедуры и значение, которое указывает на ожидаемый ею формат строк. При вызове *CallWindowProc* система проверяет, что Вы передаете — адрес оконной процедуры или описатель внутренней структуры данных. В первом случае система сразу обращается к исходной оконной процедуре, так как никаких преобразований не требуется, а во втором случае система сначала преобразует символы и строки в соответствующую кодировку и только потом вызывает исходную оконную процедуру.

Модель аппаратного ввода и локальное состояние ввода

В этой главе мы рассмотрим модель аппаратного ввода. В частности, я расскажу, как события от клавиатуры и мыши попадают в систему и пересылаются соответствующим оконным процедурам. Создавая модель ввода, Microsoft стремилась главным образом к тому, чтобы ни один поток не мог нарушить работу других потоков. Вот пример из 16-разрядной Windows: задача (так в этой системе назывались выполняемые программы), зависшая в бесконечном цикле, приводила к тому, что зависали и остальные задачи — их дальнейшее выполнение становилось невозможным. Пользователю ничего не оставалось, как только перезагрузить компьютер. А все потому, что операционная система слишком много разрешала отдельно взятой задаче. Отказоустойчивые операционные системы вроде Windows 2000 и Windows 98 не дают зависшему потоку блокировать другим потокам прием аппаратного ввода.

Поток необработанного ввода

Общая схема модели аппаратного ввода в системе показана на рис. 27-1. При запуске система создает себе особый *поток необработанного ввода* (raw input thread, RIT) и *системную очередь аппаратного ввода* (system hardware input queue, SHIQ). RIT и SHIQ — это фундамент, на котором построена вся модель аппаратного ввода.

Обычно RIT бездействует, ожидая появления какого-нибудь элемента в SHIQ. Когда пользователь нажимает и отпускает клавишу на клавиатуре или кнопку мыши, либо перемещает мышь, соответствующий драйвер устройства добавляет аппаратное событие в SHIQ. Тогда RIT пробуждается, извлекает этот элемент из SHIQ, преобразует его в сообщение (WM_KEY*, WM_?BUTTON* или WM_MOUSEMOVE) и ставит в конец очереди виртуального ввода (virtualized input queue, VIQ) нужного потока. Далее RIT возвращается в начало цикла и ждет появления следующего элемента в SHIQ. RIT никогда не перестает реагировать на события аппаратного ввода — весь его код написан самой Microsoft и очень тщательно протестирован.

Как же RIT узнает, в чью очередь надо пересыпать сообщения аппаратного ввода? Ну, с сообщениями от мыши все ясно: RIT просто выясняет, в каком окне находится ее курсор, и, вызвав *GetWindowThreadProcessId*, определяет поток, создавший это окно. Поток с данным идентификатором и получит сообщение от мыши.

В случае сообщений от клавиатуры все происходит несколько иначе. В любой момент с RIT «связан» лишь какой-то один поток, называемый *активным* (foreground thread). Именно ему принадлежит окно, с которым работает пользователь в данное время.

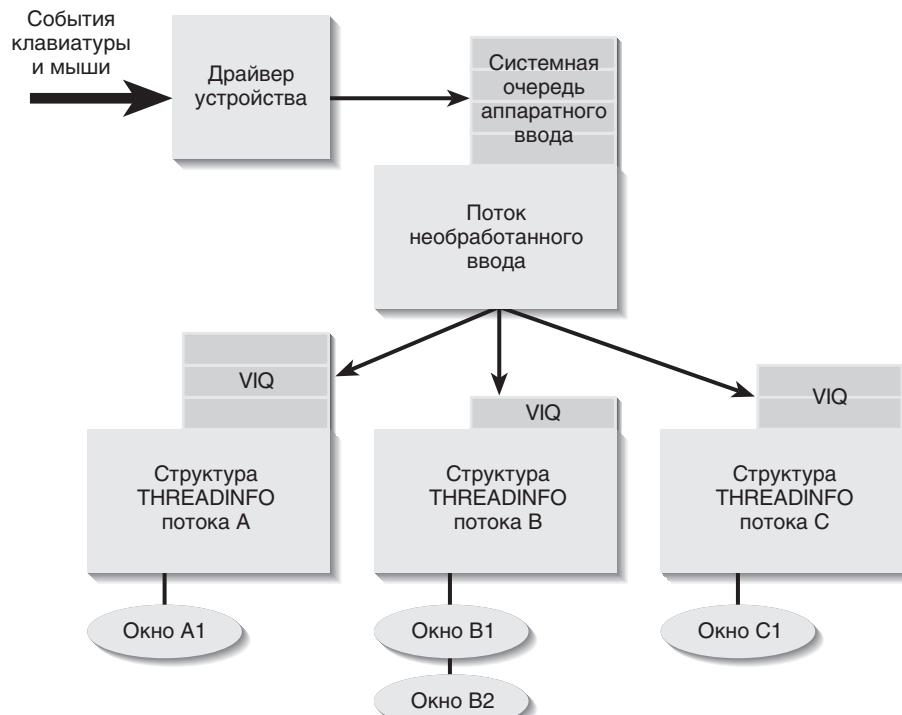


Рис. 27-1. Модель аппаратного ввода

Когда пользователь входит в систему, процесс Windows Explorer порождает поток, который создает панель задач и рабочий стол. Этот поток привязывается к RIT. Если Вы запустите Calculator, то его поток, создавший окно, немедленно подключится к RIT. После этого поток, принадлежащий Explorer, отключается от RIT, так как единовременно с RIT может быть связан только один поток. При нажатии клавиши в SHIQ появится соответствующий элемент. Это приведет к тому, что RIT пробудится, преобразует событие аппаратного ввода в сообщение от клавиатуры и поместит его в VIQ потока Calculator.

Каким образом различные потоки подключаются к RIT? Если при создании процесса его поток создает окно, последнее автоматически появляется на переднем плане (становится активным), и этот поток присоединяется к RIT. Кроме того, RIT отвечает за обработку особых комбинаций клавиш: Alt+Tab, Alt+Esc и Ctrl+Alt+Del. Поскольку эти комбинации клавиш RIT обрабатывает самостоятельно, пользователи могут в любой момент активизировать соответствующие окна с клавиатуры; ни одно приложение не в состоянии перехватить упомянутые комбинации клавиш. Как только пользователь нажимает одну из таких комбинаций клавиш, RIT активизирует выбранное окно, и в результате его поток подключается к RIT. Кстати, в Windows есть функции, позволяющие программно активизировать окно, присоединив его поток к RIT. Мы обсудим их несколько позже.

На рис. 27-1 видно, как работает механизм защиты потоков друг от друга. Посыпая сообщение в окно B1 или B2, RIT помещает его в очередь виртуального ввода потока B. Обрабатывая это сообщение, поток — при синхронизации на каком-либо объекте ядра — может войти в бесконечный цикл или попасть в ситуацию взаимной блокировки. Если так и случится, он все равно останется присоединенным к RIT, и сообщения будут поступать именно в его очередь виртуального ввода.

Однако пользователь, заметив, что ни окно В1, ни окно В2 не реагируют на его действия, может переключиться, например, в окно А1 нажатием клавиш Alt+Tab. Поскольку RIT сам обрабатывает комбинацию клавиш Alt+Tab, переключение пройдет без всяких проблем. После активизации окна А1 к RIT будет подключен поток А. Теперь пользователь может спокойно работать с окном А1, даже несмотря на то что поток В и оба его окна зависли.

Локальное состояние ввода

Независимая обработка ввода потоками, предотвращающая неблагоприятное воздействие одного потока на другой, — это лишь часть того, что обеспечивает отказоустойчивость модели аппаратного ввода. Но этого недостаточно для надежной изоляции потоков друг от друга, и поэтому система поддерживает дополнительную концепцию — *локальное состояние ввода* (local input state).

Каждый поток обладает собственным состоянием ввода, сведения о котором хранятся в структуре THREADINFO (глава 26). В информацию об этом состоянии включаются данные об очереди виртуального ввода потока и группа переменных. Последние содержат управляющую информацию о состоянии ввода.

Для клавиатуры поддерживаются следующие сведения:

- какое окно находится в фокусе клавиатуры;
- какое окно активно в данный момент;
- какие клавиши нажаты;
- состояние курсора ввода.

Для мыши учитывается такая информация:

- каким окном захвачена мышь;
- какова форма курсора мыши;
- видим ли этот курсор.

Так как у каждого потока свой набор переменных состояния ввода, то и представления об окне, находящемся в фокусе, об окне, захватившем мышь, и т. п. у них тоже сугубо свои. С точки зрения потока, клавиатурный фокус либо есть у одного из его окон, либо его нет ни у одного окна во всей системе. То же самое относится и к мыши: либо она захвачена одним из его окон, либо не захвачена никем. В общем, перечислять можно еще долго. Так вот, подобный сепаратизм приводит к некоторым последствиям — о них мы и поговорим.

Ввод с клавиатуры и фокус

Как Вы уже знаете, ввод с клавиатуры направляется потоком необработанного ввода (RIT) в очередь виртуального ввода какого-либо потока, но только не в окно. RIT помещает события от клавиатуры в очередь потока безотносительно конкретному окну. Когда поток вызывает *GetMessage*, событие от клавиатуры извлекается из очереди и перенаправляется окну (созданному потоком), на котором в данный момент сосредоточен фокус ввода (рис. 27-2).

Чтобы направить клавиатурный ввод в другое окно, нужно указать, в очередь какого потока RIT должен помещать события от клавиатуры, а также «сообщить» переменным состояния ввода потока, какое окно будет находиться в фокусе. Одним вызовом *SetFocus* эти задачи не решить. Если в данный момент ввод от RIT получает поток 1, то вызов *SetFocus* с передачей описателей окон А, В или С приведет к смене фокуса.

Окно, теряющее фокус, убирает используемый для обозначения фокуса прямоугольник или гасит курсор ввода, а окно, получающее фокус, рисует такой прямоугольник или показывает курсор ввода.

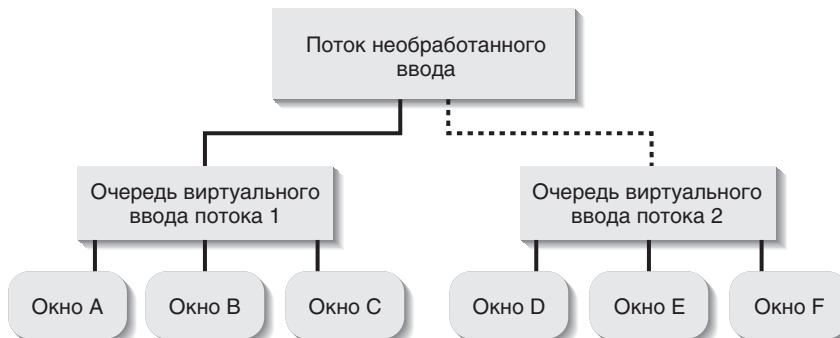


Рис. 27-2. RIT направляет пользовательский ввод с клавиатуры в очередь виртуального ввода только одного из потоков единовременно

Предположим, однако, что поток 1 по-прежнему получает ввод от RIT и вызывает *SetFocus*, передавая ей описатель окна E. В этом случае система не дает функции что-либо сделать, так как окно, на которое Вы хотите перевести фокус, не использует очередь виртуального ввода, подключенную в данный момент к RIT. Когда поток 1 выполнит этот вызов, на экране не произойдет ни смены фокуса, ни каких-либо изменений.

Возьмем другую ситуацию: поток 1 подключен к RIT, а поток 2 вызывает *SetFocus*, передавая ей описатель окна E. На этот раз значения переменных локального состояния ввода потока 2 изменятся так, что — когда RIT в следующий раз направит события от клавиатуры этому потоку — ввод с клавиатуры получит окно E. Этот вызов не заставит RIT направить клавиатурный ввод в очередь виртуального ввода потока 2.

Так как фокус теперь сосредоточен на окне E потока 2, оно получает сообщение WM_SETFOCUS. Если окно E — кнопка, на нем появляется прямоугольник, обозначающий фокус, и в результате на экране могут появиться два окна с такими прямоугольниками (окна А и Е). Сами понимаете, это вряд ли кому понравится. Поэтому вызывать *SetFocus* следует с большой осторожностью — чтобы не создавать подобных ситуаций. Вызов *SetFocus* безопасен, только если Ваш поток подключен к RIT.

Кстати, если Вы переведете фокус на окно, которое, получив сообщение WM_SETFOCUS, показывает курсор ввода, не исключено одновременное появление на экране нескольких окон с таким курсором. Это тоже вряд ли кому обрадует.

Когда фокус переводится с одного окна на другое обычным способом (например, щелчком окна), теряющее фокус окно получает сообщение WM_KILLFOCUS. Если окно, получающее фокус, принадлежит другому потоку, переменные локального состояния ввода потока, который владеет окном, теряющим фокус, обновляются так, чтобы показать: окон в фокусе нет. И вызов *GetFocus* возвращает при этом NULL, заставляя поток считать, что окон в фокусе нет.

Функция *SetActiveWindow* активизирует в системе окно верхнего уровня и переводит на него фокус:

```
HWND SetActiveWindow(HWND hwnd);
```

Как и *SetFocus*, эта функция ничего не делает, если поток вызывает ее с описателем окна, созданного другим потоком.

Функцию *SetActiveWindow* дополняет *GetActiveWindow*:

```
HWND GetActiveWindow();
```

Она работает так же, как и *GetFocus*, но возвращает описатель активного окна, указанного в переменных локального состояния ввода вызывающего потока. Так что, если активное окно принадлежит другому потоку, функция возвращает NULL.

Есть и другие функции, влияющие на порядок размещения окон, их статус (активно или неактивно) и фокус:

```
BOOL BringWindowToTop(HWND hwnd);
```

```
BOOL SetWindowPos(
    HWND hwnd,
    HWND hwndInsertAfter,
    int x,
    int y,
    int cx,
    int cy,
    UINT fuFlags);
```

Обе эти функции работают одинаково (фактически *BringWindowToTop* вызывает *SetWindowPos*, передавая ей *HWND_TOP* во втором параметре). Когда поток, вызывающий любую из этих функций, не связан с RIT, они ничего не делают. В ином случае (когда поток связан с RIT) система активизирует указанное окно. Обратите внимание, что здесь не имеет значения, принадлежит ли это окно вызвавшему потоку. Окно становится активным, а к RIT подключается тот поток, который создал данное окно. Кроме того, значения переменных локального состояния ввода обоих потоков обновляются так, чтобы отразить эти изменения.

Иногда потоку нужно вывести свое окно на передний план. Например, Вы запланировали какую-то встречу, используя Microsoft Outlook. Где-то за полчаса до назначенного времени Outlook выводит на экран диалоговое окно с напоминанием о встрече. Если поток Outlook не связан с RIT, это диалоговое окно появится под другими окнами, и Вы его не увидите. Поэтому нужен какой-то способ, который позволил бы привлекать внимание к определенному окну, даже если в данный момент пользователь работает с окном другого приложения.

Вот функция, которая выводит окно на передний план и подключает его поток к RIT:

```
BOOL SetForegroundWindow(HWND hwnd);
```

Одновременно система активизирует окно и переводит на него фокус. Функция, парная *SetForegroundWindow*:

```
HWND GetForegroundWindow();
```

Она возвращает описатель окна, находящегося сейчас на переднем плане.

В более ранних версиях Windows функция *SetForegroundWindow* срабатывала всегда. То есть поток, вызвавший ее, всегда мог перевести указанное окно на передний план (даже если оно было создано другим потоком). Однако разработчики стали злоупотреблять этой функцией и нагромождать окна друг на друга. Представьте, я пишу журнальную статью, и вдруг высакивает окно с сообщением о завершении печати. Если бы я не смотрел на экран, то начал бы вводить текст не в свой документ, а в это окно. Еще больше раздражает, когда пытаешься выбрать команду в меню, а на экране появляется какое-то окно и закрывает меню.

Чтобы прекратить всю эту неразбериху, Microsoft сделала *SetForegroundWindow* чуть поумнее. В частности, эта функция срабатывает, только если вызывающий поток уже подключен к RIT или если поток, связанный с RIT в данный момент, не получал ввода на протяжении определенного периода (который задается функцией *SystemParametersInfo* и значением SPI_SETFOREGROUNDLOCKTIMEOUT). Кроме того, *SetForegroundWindow* терпит неудачу, когда активно какое-нибудь меню.

Если *SetForegroundWindow* не удается переместить окно на передний план, то его кнопка на панели задач начинает мигать. Заметив это, пользователь будет в курсе, что окно требует его внимания. Чтобы выяснить, в чем дело, пользователю придется активизировать это окно вручную. Управлять режимом мигания окна позволяет функция *SystemParametersInfo* со значением SPI_SETFOREGROUNDFLASHCOUNT.

Из-за такого поведения *SetForegroundWindow* в систему встроено несколько новых функций. Первая из них, *AllowSetForegroundWindow*, разрешает потоку указанного процесса успешно вызвать *SetForegroundWindow*, но только если и вызывающий ее поток может успешно вызвать *SetForegroundWindow*. Чтобы любой процесс мог выводить окно «поверх» остальных окон, открытых Вашим потоком, передайте в параметре *dwProcessId* значение ASFW_ANY (определенное как -1):

```
BOOL AllowSetForegroundWindow(DWORD dwProcessId);
```

Кроме того, можно полностью заблокировать работу *SetForegroundWindow*, вызвав *LockSetForegroundWindow*:

```
BOOL LockSetForegroundWindow(UINT uLockCode);
```

В параметре *uLockCode* она принимает либо LSFW_LOCK, либо LSFW_UNLOCK. Данная функция вызывается системой, когда на экране активно какое-нибудь системное меню, — чтобы никакое окно не могло его закрыть. (Поскольку меню Start не является встроенным, то при его открытии Windows Explorer сам вызывает эти функции.)

Система автоматически снимает блокировку с функции *SetForegroundWindow*, когда пользователь нажимает клавишу Alt или активизирует какое-либо окно. Так что приложение не может навечно заблокировать *SetForegroundWindow*.

Другой аспект управления клавиатурой и локальным состоянием ввода связан с массивом синхронного состояния клавиш (synchronous key state array). Этот массив включается в переменные локального состояния ввода каждого потока. В то же время массив асинхронного состояния клавиш (asynchronous key state array) — только один, и он разделяется всеми потоками. Эти массивы отражают состояние всех клавиш на данный момент, и функция *GetAsyncKeyState* позволяет определить, нажата ли сейчас заданная клавиша:

```
SHORT GetAsyncKeyState(int nVirtKey);
```

Параметр *nVirtKey* задает код виртуальной клавиши, состояние которой нужно проверить. Старший бит результата определяет, нажата в данный момент клавиша (1) или нет (0). Я часто пользовался этой функцией, определяя при обработке сообщения, отпустил ли пользователь основную (обычно левую) кнопку мыши. Передав значение VK_LBUTTON, я ждал, когда обнулится старший бит. Заметьте, что *GetAsyncKeyState* всегда возвращает 0 (не нажата), если ее вызывает другой поток, а не тот, который создал окно, находящееся сейчас в фокусе ввода.

Функция *GetKeyState* отличается от *GetAsyncKeyState* тем, что возвращает состояние клавиатуры на момент, когда из очереди потока извлечено последнее сообщение от клавиатуры:

```
SHORT GetKeyState(int nVirtKey);
```

Эту функцию можно вызвать в любой момент; для нее неважно, какое окно в фокусе.

Управление курсором мыши

В концепцию локального состояния ввода входит и управление состоянием курсора мыши. Поскольку мышь, как и клавиатура, должна быть доступна всем потокам, Windows не позволяет какому-то одному потоку монопольно распоряжаться курсором мыши, изменяя его форму или ограничивая область его перемещения. Посмотрим, как система управляет этим курсором.

Один из аспектов управления курсором мыши заключается в его отображении или гашении. Если поток вызывает *ShowCursor(FALSE)*, то система скрывает курсор, когда он оказывается на любом окне, созданном этим потоком, и показывает курсор всякий раз, когда он попадает в окно, созданное другим потоком.

Другой аспект управления курсором мыши — возможность ограничить его перемещение каким-либо прямоугольным участком. Для этого надо вызвать функцию *ClipCursor*:

```
BOOL ClipCursor(CONST RECT *prc);
```

Она ограничивает перемещение курсора мыши прямоугольником, на который указывает параметр *prc*. И опять система разрешает потоку ограничить перемещение курсора заданным прямоугольником. Но, когда возникает событие асинхронной активизации, т. е. когда пользователь переключается в окно другого приложения, нажимает клавиши Ctrl+Esc или же поток вызывает *SetForegroundWindow*, система снимает ограничения на передвижение курсора, позволяя свободно перемещать его по экрану.

И здесь мы подошли к концепции *захвата мыши* (*mouse capture*). «Захватывая» мышь (вызовом *SetCapture*), окно требует, чтобы все связанные с мышью сообщения RIT отправлял в очередь виртуального ввода вызывающего потока, а из нее — установившему захват окну до тех пор, пока программа не вызовет *ReleaseCapture*.

Как и в предыдущих случаях, это тоже снижает отказоустойчивость системы, но без компромиссов, увы, не обойтись. Вызывая *SetCapture*, поток заставляет RIT помечать все сообщения от мыши в свою очередь виртуального ввода. При этом *SetCapture* соответственно настраивает переменные локального состояния ввода данного потока.

Обычно приложение вызывает *SetCapture*, когда пользователь нажимает кнопку мыши. Но поток может вызвать эту функцию, даже если нажатия кнопки мыши не было. Если *SetCapture* вызывается при нажатой кнопке, захват действует для всей системы. Как только система определяет, что все кнопки мыши отпущены, RIT перестает направлять сообщения от мыши исключительно в очередь виртуального ввода данного потока. Вместо этого он передает сообщения в очередь ввода, связанную с окном, «поверх» которого курсор находится в данный момент. И это нормальное поведение системы, когда захват мыши не установлен.

Однако для вызвавшего *SetCapture* потока ничего не меняется. Всякий раз, когда курсор оказывается на любом из окон, созданных установившим захват потоком, сообщения от мыши направляются в окно, применительно к которому этот захват и установлен. Иначе говоря, когда пользователь отпускает все кнопки мыши, захват осуществляется на уровне лишь данного потока, а не всей системы.

Если пользователь попытается активизировать окно, созданное другим потоком, система автоматически отправит установившему захват потоку сообщения о нажатии и отжатии кнопок мыши. Затем она изменит переменные локального состояния ввода потока, чтобы отразить тот факт, что поток более не работает в режиме захвата. Словом, Microsoft считает, что захват мыши чаще всего применяется для выполнения таких операций, как щелчок и перетаскивание экранного объекта.

Последняя переменная локального состояния ввода, связанная с мышью, относится к форме курсора. Всякий раз, когда поток вызывает *SetCursor* для изменения формы курсора, переменные локального состояния ввода соответствующим образом обновляются. То есть переменные локального состояния ввода всегда запоминают последнюю форму курсора, установленную потоком.

Допустим, пользователь перемещает курсор мыши на окно Вашей программы, окно получает сообщение WM_SETCURSOR, и Вы вызываете *SetCursor*, чтобы преобразовать курсор в «песочные часы». Вызвав *SetCursor*, программа начинает выполнять какую-то длительную операцию. (Бесконечный цикл — лучший пример длительной операции. Шутка.) Далее пользователь перемещает курсор из окна Вашей программы в окно другого приложения, и это окно может изменить форму курсора.

Для такого изменения переменные локального состояния ввода не нужны. Но переведем курсор обратно в то окно, поток которого по-прежнему занят обработкой. Системе «хочется» послать окну сообщения WM_SETCURSOR, но процедура этого окна не может выбрать их из очереди, так как его поток продолжает свою операцию. Тогда система определяет, какая форма была у курсора в прошлый раз (информация об этом содержится в переменных локального состояния ввода данного потока), и автоматически восстанавливает ее (в нашем примере — «песочные часы»). Теперь пользователю четко видно, что в этом окне работа еще не закончена и придется подождать.

Подключение к очередям виртуального ввода и переменным локального состояния ввода

Как Вы уже убедились, отказоустойчивость модели ввода достигается благодаря тому, что у каждого потока имеются собственные переменные локального состояния ввода, а подключение потока к RIT и отключение от него происходит по мере необходимости. Иногда нужно, чтобы два потока (или более) разделяли один набор переменных локального состояния ввода или одну очередь виртуального ввода.

Вы можете заставить два и более потока совместно использовать одну и ту же очередь виртуального ввода и переменные локального состояния ввода с помощью функции *AttachThreadInput*:

```
BOOL AttachThreadInput(
    DWORD idAttach,
    DWORD idAttachTo,
    BOOL fAttach);
```

Параметр *idAttach* задает идентификатор потока, чьи переменные локального состояния ввода и очередь виртуального ввода Вам больше не нужны, а параметр *idAttachedTo* — идентификатор потока, чьи переменные локального состояния ввода и виртуальная очередь ввода должны совместно использоваться потоками. И, наконец, параметр *fAttach* должен быть или TRUE, чтобы инициировать совместное использование одной очереди, или FALSE — тогда каждый поток будет вновь использовать свои переменные состояния ввода и очередь. А чтобы одну очередь (и переменные состояния ввода) разделяли более двух потоков, вызовите *AttachThreadInput* соответствующее число раз.

Вернемся к одному из предыдущих примеров и допустим, что поток А вызывает *AttachThreadInput*, передавая в первом параметре свой идентификатор, во втором — идентификатор потока В и в последнем — TRUE:

```
AttachThreadInput(idThreadA, idThreadB, TRUE);
```

Теперь любое событие аппаратного ввода, адресованное окну A1, B1 или B2, будет добавлено в конец очереди виртуального ввода потока B. Аналогичная очередь потока A больше не получит новых событий, если только Вы не разъедините очереди, повторно вызвав *AttachThreadInput* с передачей FALSE в параметре *fAttach*.

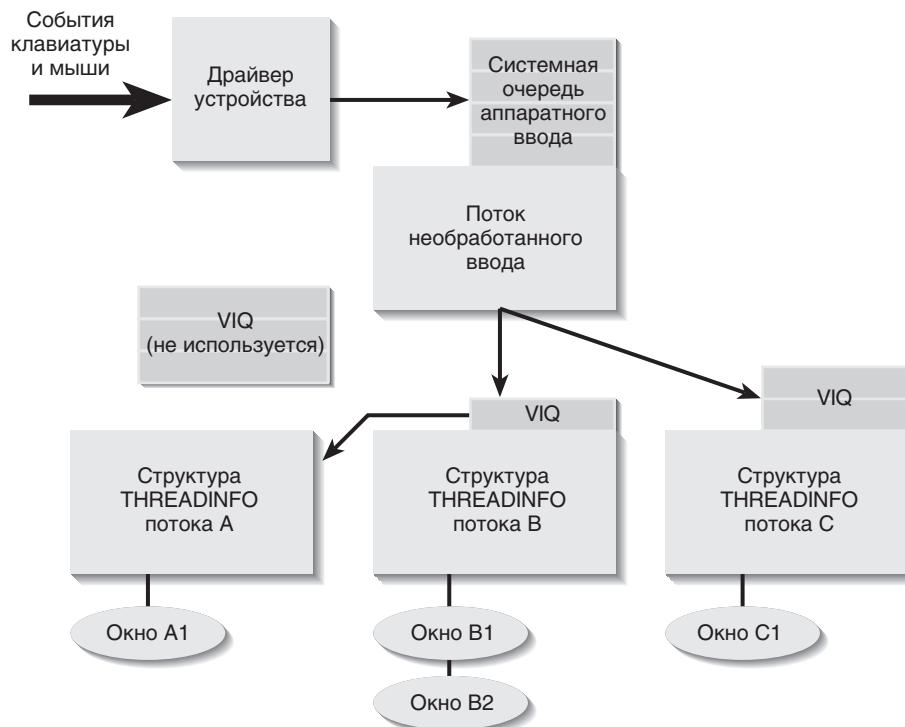


Рис. 27-3. Аппаратные сообщения для окон A1, B1 и B2 помещаются в очередь виртуального ввода потока B

Потоки, присоединенные к одной очереди виртуального ввода (VIQ), сохраняют индивидуальные очереди сообщений (синхронных, асинхронных и ответных), а также флаги пробуждения. Однако Вы серьезно снизите надежность системы, если заставите все потоки использовать одну очередь сообщений. Если какой-нибудь поток зависнет при обработке нажатия клавиши, другие потоки не получат никакого ввода. Поэтому использования *AttachThreadInput* следует по возможности избегать.

Система неявно соединяет очереди виртуального ввода двух потоков, если какой-то из них устанавливает ловушку регистрации (journal record hook) или ловушку воспроизведения (journal playback hook). Когда ловушка снимается, система восстанавливает схему организации ввода, существовавшую до установки ловушки.

Установкой ловушки регистрации поток сообщает, что хочет получать уведомления о всех аппаратных событиях, вызываемых пользователем. Поток обычно сохраняет или регистрирует эту информацию в файле. Так как пользовательский ввод должен быть зарегистрирован в том порядке, в котором он происходил, все потоки в системе начинают разделять одну очередь виртуального ввода для синхронизации обработки ввода.

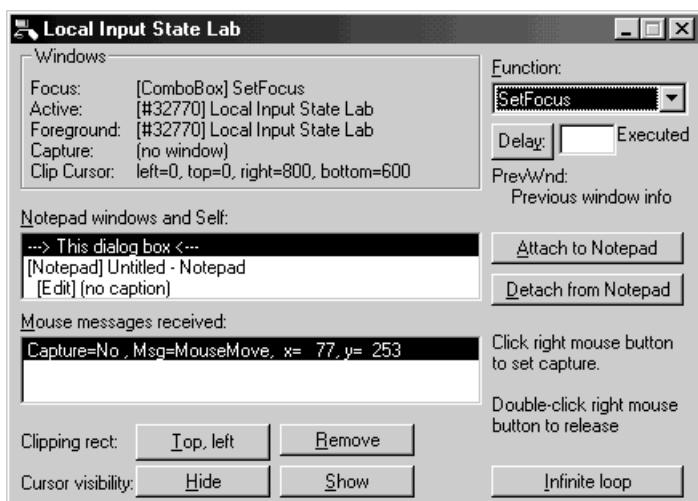
Есть еще один случай, когда система неявно вызывает *AttachThreadInput*. Допустим, приложение создает два потока. Первый открывает на экране диалоговое окно. Затем второй поток вызывает *CreateWindow*, указывая стиль WS_CHILD и передавая описа-

тель этого диалогового окна, чтобы оно стало «родителем» дочернего окна. Тогда система сама вызывает *AttachThreadInput*, чтобы поток (которому принадлежит дочернее окно) использовал ту же очередь ввода, что и поток, создавший исходное диалоговое окно. Это приводит к синхронизации ввода во всех дочерних окнах исходного диалогового окна.

Программа-пример LISLab

Эта программа, «27 LISLab.exe» (см. листинг на рис. 27-4), — своего рода лаборатория, в которой Вы сможете поэкспериментировать с локальным состоянием ввода. Файлы исходного кода и ресурсов этой программы находятся в каталоге 27-LISLab на компакт-диске, прилагаемом к книге.

В качестве подопытных кроликов нам понадобятся два потока. Один поток есть в нашей LISLab, а вторым будет Notepad. Если на момент запуска LISLab программа Notepad не выполняется, LISLab сама запустит эту программу. После инициализации LISLab Вы увидите следующее диалоговое окно.



В левом верхнем углу окна — раздел Windows; его поля обновляются дважды в секунду, т. е. дважды в секунду диалоговое окно получает сообщение WM_TIMER и в ответ вызывает функции *GetFocus*, *GetActiveWindow*, *GetForegroundWindow*, *GetCapture* и *GetClipCursor*. Первые четыре функции возвращают описатели окна (считываемые из переменных локального состояния ввода моего потока), через которые я могу определить класс и заголовок окна и вывести эту информацию на экран.

Если я активизирую другое приложение (тот же Notepad), названия полей Focus и Active меняются на (No Window), а поля Foreground — на [Notepad] Untitled - Notepad. Обратите внимание, что активизация Notepad заставляет LISLab считать, что ни активных, ни находящихся в фокусе окон нет.

Теперь поэкспериментируем со сменой фокуса. Выберем SetFocus в списке Function — в правом верхнем углу диалогового окна. Затем в поле Delay введем время (в секундах), в течение которого LISLab будет ждать, прежде чем вызвать *SetFocus*. В данном случае, видимо, лучше установить нулевое время задержки. Позже я объясню, как используется поле Delay.

Выберем окно (описатель которого мы хотим передать функции *SetFocus*) в списке Notepad Windows And Self, расположенным в левой части диалогового окна. Для эксперимента укажем [Notepad] Untitled - Notepad. Теперь все готово к вызову *SetFocus*.

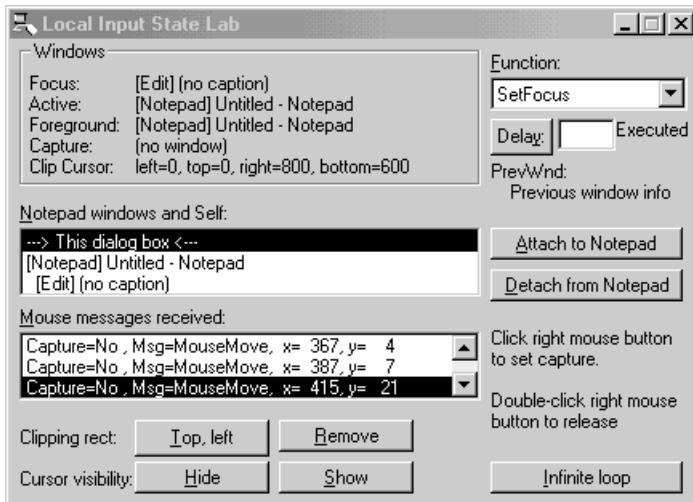
Щелкните кнопку Delay и понаблюдайте, что произойдет в разделе Windows. Ничего. Система отказалась менять фокус.

Если Вы действительно хотите перевести фокус на Notepad, щелкните кнопку Attach To Notepad, что заставит LISLab вызвать:

```
AttachThreadInput(GetWindowThreadProcessId(g_hwndNotepad, NULL),
    GetCurrentThreadId(), TRUE);
```

В результате этого вызова поток LISLab станет использовать ту же очередь виртуального ввода, что и Notepad. Кроме того, поток LISLab «разделит» переменные локального состояния ввода Notepad.

Если после щелчка кнопки Attach To Notepad Вы щелкнете окно Notepad, диалоговое окно LISLab примет следующий вид.



Теперь, когда очереди ввода соединены друг с другом, LISLab способна отслеживать изменения фокуса, происходящие в Notepad. В приведённом выше диалоговом окне показано, что в данный момент фокус установлен на поле ввода. А если мы откроем в Notepad диалоговое окно File Open, то LISLab, продолжая следить за Notepad, покажет нам, какое окно Notepad получило фокус ввода, какое окно активно и т. д.

Теперь можно вернуться в LISLab, щелкнуть кнопку Delay и вновь попытаться заставить SetFocus перевести фокус на Notepad. На этот раз все пройдет успешно, потому что очереди ввода соединены.

Если хотите, поэкспериментируйте с SetActiveWindow, SetForegroundWindow, BringWindowToTop и SetWindowPos, выбирая нужную функцию в списке Function. Попробуйте вызывать их и когда очереди соединены, и когда они разъединены; при этом обращайте внимание на различия в поведении функций.

А сейчас я поясню, зачем предусмотрена задержка. Она заставляет LISLab вызывать указанную функцию только по истечении заданного числа секунд. Для иллюстрации возьмем такой пример. Но прежде отключите LISLab от Notepad, щелкнув кнопку Detach From Notepad. Затем в списке Notepad Windows And Self выберите --->This Dialog Box<---, а в списке Function — SetFocus и установите задержку на 10 секунд. Наконец «нажмите» кнопку Delay и быстро щелкните окно Notepad, чтобы оно стало активным. Вы должны активизировать Notepad до того, как истекут заданные 10 секунд.

Пока идет отсчет времени задержки, справа от счетчика высвечивается слово Pending. По истечении 10 секунд слово Pending меняется на Executed, и появляется

способом питания ловят насекомых и переваривают их с помощью специальных ферментов. Некоторые одноклеточные, в частности эвглена зеленая, могут не только получать питательные вещества с помощью фотосинтеза, но и усваивать готовые органические продукты. Такие организмы относят к *миксотрофным*, или организмам со смешанным типом питания.

Помимо различий в питании между животными и растениями, существует ряд других различий, например, животные клетки не имеют твердой целлюлозной оболочки; животным свойствен интенсивный обмен веществ и ограниченный рост; у высших животных имеются системы органов: нервная, кровеносная, дыхательная, двигательная, пищеварительная, выделительная и половая.

Все перечисленные различия между животными и растениями относительны, что обусловлено генетическим единством живой природы. Однако для удобства изучения живых организмов биологию подразделяют на несколько крупных дисциплин: микробиологию, микологию, ботанику и зоологию.

История зоологии. На ранних этапах развития первобытного общества человек неразрывно был связан с животным миром. Человек охотился, приручал и разводил животных, изучал их строение, образ жизни и болезни. Обобщение сведений о животных привело к возникновению самостоятельной науки — *зоологии* (от греч. zoon — животное, logos — учение). Зоология входит в комплекс биологических дисциплин, изучающих живую природу (от греч. bios — жизнь).

Основоположником зоологии считают древнегреческого ученого и философа Аристотеля (384—322 гг. до н. э.), разделившего известных ему животных на две группы: животные, не имеющие крови, и животные, имеющие кровь и спинной хребет. Такое деление соответствует в какой-то мере делению животных на беспозвоночных и позвоночных. Аристотель был и блестящим анатомом: в своих трудах по анатомии животных он определил новое направление в зоологической науке.

Изобретение А. Левенгука (1632—1723) микроскопа позволило изучать ранее неизвестный человеку огромный по своему разнообразию мир одноклеточных организмов, в том числе и представителей животного царства. Были открыты половые гаметы животных и человека, изучены клетки крови и т. д. В результате сформировалась новая отрасль биологии — микроскопическая анатомия.

Английский исследователь Д. Рей (1627—1705) вводит понятие *вид*, определяя его как группу морфологически сходных особей, подобных своим родителям. Но наибольший вклад в классификацию внес шведский ученый Карл Линней (1707—1778). Его система легла в основу современной систематики растений и животных. В качестве исходной единицы классификации животных был предложен вид. К. Линней считал, что вид — это совокупность особей, сходных по морфологическим признакам, скрещивающихся между собой и дающих плодовитое потомство. К. Линней предложил иерархию систематических категорий у животных: класс, отряд, род и вид. По Линнею, каждое животное

должно иметь двойное (бинарное) латинское название: первое — название рода, в который входит данный вид, второе — собственно наименование вида, например: мышь домовая — *Mus musculus*. Система животных К. Линнея включала только шесть классов: черви, насекомые, рыбы, гады, птицы и млекопитающие.

Основоположником сравнительной анатомии и палеонтологии по праву считается французский ученый Жорж Кювье (1769—1832), сформулировавший принцип корреляции; он развел учение о целостности организации животных, реконструировал облик вымерших животных по их ископаемым остаткам.

Одним из крупнейших теоретиков в области сравнительной эмбриологии и анатомии был Этьен Жоффруа Сент-Илер (1772—1844), который развел представление о едином плане строения всех животных и положил начало учению о гомологии и аналогии органов.

Жан Батист Ламарк (1744—1829) был создателем первой естественной системы животных и эволюционной теории. Система Ламарка включала уже 14 классов, что отражало усложнение организации животных и преемственность в эволюционном развитии; эти классы располагались по ступеням эволюционной лестницы. Систему, разработанную Ж. Б. Ламарком, считают первой естественной системой, поскольку в ней учитывалась степень родства между классами животных. Основными факторами эволюции ученый считал изменчивость под влиянием среды, наследуемость приобретенных признаков и стремление к прогрессу.

В первой половине XIX в. появляются фундаментальные труды К. Бэра по сравнительной эмбриологии, А. Гумбольдта по биологической географии, Ч. Лайеля по исторической геологии; в это же время М. Шлейден и Т. Шванн создают клеточную теорию, в которой убедительно показывают единство микроскопического строения животных и растений; в трудах К. Ф. Рулье была опровергнута теория неизменяемости видов. Успехи естественных наук способствовали разработке научно обоснованной эволюционной теории Ч. Дарвином (1809—1882), который внес значительный вклад в развитие зоологии, биогеографии, палеонтологии и эмбриологии. Им был открыт основной движущий фактор эволюции — естественный отбор.

Во второй половине XIX в. под влиянием работ Ч. Дарвина развивается эволюционное направление в зоологии. Немецкие ученые Э. Геккель и Ф. Мюллер сформулировали «биогенетический закон» о соответствии индивидуального (онтогенеза) и исторического (филогенеза) развития. Формируются новые научные направления по эволюционной эмбриологии (Ф. Мюллер, И. И. Мечников, А. О. Ковалевский), по эволюционной палеонтологии (В. О. Ковалевский), по эволюционной физиологии животных (И. И. Сеченов), по филогенетике и эволюционной систематике (Э. Геккель), появляются работы по генетике (Г. Мендель, А. Вейсман), экологии (Н. А. Северцов), зоогеографии (Семенов-Тян-Шанский) и ряд других.

Зоология — комплексная наука. Новейшие достижения и научно-технический прогресс оказали огромное влияние на развитие мировой и отечественной зоологической науки. Зоология превратилась в сложную систему дисциплин. Сформировалось множество научных направлений, каждое из которых имеет свои задачи и в исследованиях использует свои методические подходы. Вместе с тем эти науки не обособились друг от друга, а составляют единую систему наук, поскольку у всех этих наук общий объект познания — животный мир.

Морфология (от греч. *morphe* — форма) изучает и объясняет внешнее и внутреннее строение организма. Описание морфологии животных — основа всякого зоологического исследования. С помощью метода сравнительной морфологии выявляют наиболее характерные черты в строении различных животных и степень сложности их организации. Кроме того, с помощью этого метода устанавливают исторические связи между группами животных.

Одним из достижений сравнительной и экологической морфологии является разработка учения об аналогичных и гомологичных органах. *Аналогичные органы* различаются по своему строению, но выполняют сходные (аналогичные) функции; примером могут служить крылья насекомых и крылья птиц. У перечисленных животных нет сходства в строении крыльев, но они выполняют сходные функции. *Гомологичные органы* имеют коренное сходство в строении, развиваются из сходных зачатков, но функции могут выполнять различные. Примером гомологичных органов служат конечности у позвоночных (рис. 1).

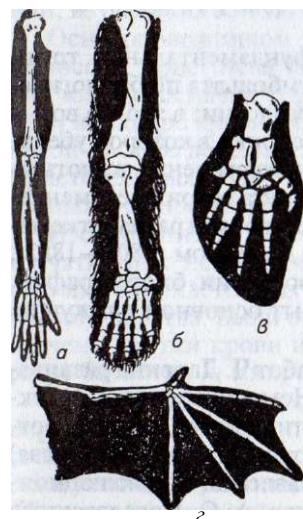


Рис. 1. Гомологичные органы на примере передней конечности позвоночных:
а — рука человека; б — лапа медведя; в — ласт кита; г — крыло летучей мыши

Сравнительный метод в морфологии позволил разработать учение о *конвергенции*, т. е. о сходстве в строении организмов, живущих в сходных условиях и ведущих сходный образ жизни. Определяющим в обозначении конвергентной изменчивости является независимое приобретение организмами общих приспособлений (адаптаций) и приспособительных (адаптивных) черт, формирующихся на базе изменчивости, которая была сохранена и закреплена отбором. Так, на рис. 2 изображены дельфин (млекопитающее), ихтиозавр (пресмыкающееся) и акула (рыба), которые характеризуются конвергентным сходством в форме тела. Но конвергенция может проявляться и в утрате органов, например в редукции глаз у обитателей пещер и подземных водоемов.

Физиология (от греч. *fazis* — природа) — наука о процессах жизнедеятельности организма и деятельности его органов в связи с их строением. Закон о единстве формы и

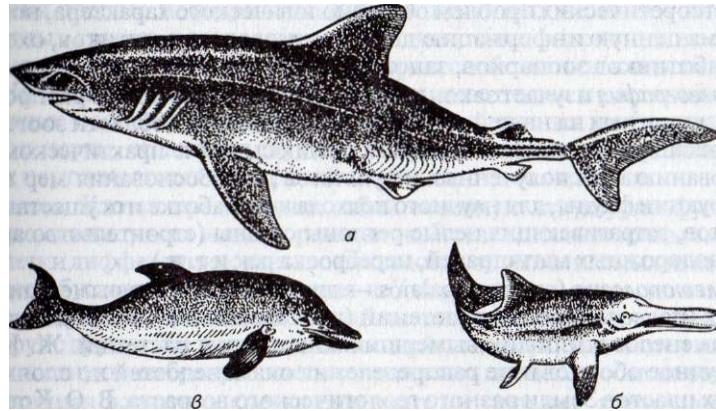


Рис. 2. Конвергенция:
а — акула; б — ихтиозавр; в — дельфин

фу 11 кции находит подтверждение в морфологии и физиологии. В настоя-
11 исе время физиология как наука имеет множество направлений, базирую-
щихся на учении И. П. Павлова о нервной системе, объединяющей орга-
низм в единое целое.

Эмбриология (от греч. *embryon* — зародыш) — наука о зародышевом развитии животных. Сравнительная эмбриология позволяет установить закономерности в развитии зародышей разных животных, понять взаимосвязь онтогенеза и филогенеза. Пройденные исторические этапы в эволюционном развитии животных в какой-то степени проявляются у современных видов на эмбриональных стадиях их развития. Менно в раннем периоде развития у зародышей разных видов животных наблюдается наибольшее сходство. Такая закономерность была сформулирована К. Бэрром как закон зародышевого сходства. Сущность этого закона заключается в том, что в эмбриональном периоде развития раньше других закладываются признаки типа, к которому принадлежит животное, затем признаки класса, позднее признаки отряда и, наконец, оформляются признаки вида. Эта закономерность повторения в онтогенезе стадий филогенеза была обобщена Э. Геккелем в форме биогенетического закона: онтогенез есть краткое повторение филогенеза.

Экология (от греч. *oikos* — жилище, местообитание) изучает животных в связи с местом и условиями их жизни, закономерности во взаимосвязях организмов со средой обитания. Одной из задач экологии является исследование приспособительных черт в строении животных, в их жизненных направлениях, в поведении. Появляется новая наука — **этология** (от греч. *Sthos* — характер, нрав), которая позволяет раскрыть многие закономерности жизнедеятельности сообществ животных, взаимоотношения отдельных видов, и в частности конкуренцию, паразитизм и т. п. Данные экологии и этологии ценны не только для разра-

ботки теоретических проблем общебиологического характера, но несут и весьма ценную информацию для животноводов-практиков, охотников, работников зоопарков, заповедников и т. д.

Зоогеография изучает закономерности географического распространения животных на нашей планете. Исследования в области зоогеографии, экологии и этологии создают предпосылки к практическому использованию всех полученных результатов для обоснования мер по реконструкции фауны, для научного подхода к разработке и осуществлению проектов, затрагивающих целые регионы страны (строительство авто- и железнодорожных магистралей, переброска рек и т. п.).

Палеонтология (от греч. *palaios* — древний) изучает вымерших животных (палеозоология) и растений (палеоботаника), восстанавливает по окаменелостям облик вымерших животных и растений. Ж. Кювье дал научное обоснование распределения окаменелостей по слоям осадочных пластов Земли разного геологического возраста. В. О. Ковалевский призывал палеонтологов не только констатировать различия в фаунах разных эпох, но и искать среди ископаемых остатков сходство между животными смежных напластований.

Филогенетика (от греч. *phylon* — племя, род; *genesis* — происхождение) занимается изучением исторических связей в мире животных и растений. В результате изучения родственных отношений животных было построено родословное древо животного мира.

Систематика (от греч. *systematikos* — упорядоченный) на основе данных всех зоологических наук разрабатывает классификацию животных и естественную систему животного мира, отражающую родственные связи различных групп животных. Естественная система базируется на том, что все организмы постоянно развиваются. Эта система предназначена отражать генеалогические связи организмов, ибо для объединения животных в ту или иную категорию основанием служит степень сходства как отражение родства. К одному виду относят животных, наиболее близко родственных (что и объясняет их морфологическое сходство), свободно скрещивающихся между собой с получением плодовитого потомства и имеющих общее географическое распространение. Видовые признаки стойкие и отличаются исключительной консервативностью.

В результате внутривидовой изменчивости образуются подвиды и разновидности. Общеизвестны различия в окраске, опущенности, в размерах и т. п. у животных одного вида, но обитающих в разных географических зонах. Особенно большая пластичность свойственна моллюскам и насекомым.

Особенности организма животных. Эволюция живой природы на Земле привела к образованию животных и растений. Важное различие между ними заключается в характере обмена веществ, который обусловлен типом питания. Если растения в большинстве своем автотрофные организмы, то животные, как правило, гетеротрофные организмы. В темноте растения погибают, а животные способны жить.

Большинство животных ведут активный образ жизни, свободно перемещаясь в пространстве или совершая разнообразные движения.

Гас гения обычно неподвижны. Клетки животных не имеют плотных неточных стенок, построенных из целлюлозных волокон, и не содержат вакуолей с клеточным соком, которые свойственны клеткам растений! Но провести резкую границу между животными и растениями невозможно; особенно это трудно сделать для низших их форм, сохраняющих черты, общие для двух царств природы. Одна из кардинальных ощущений черт животных и растений — их клеточное строение.

Клетка. Клетка представляет собой основную структурно-функциональную единицу всех живых организмов. В теле многоклеточных животных клетки дифференцированы в зависимости от выполняемых ими функций, что обусловливает их различия не только по размерам, но также по форме и строению. Все клетки организма взаимосвязаны и взаимодействуют друг с другом; такая связь и взаимодействие осуществляются через их плазматическую мембрану. В клетках протекают процессы обмена веществ.

Типичная животная клетка содержит цитоплазму, ядро (или ядра) и различные органоиды, или органеллы; сама клетка ограничена наружной мембраной, которую называют плазматической мембраной или плазмалеммой (рис. 3).

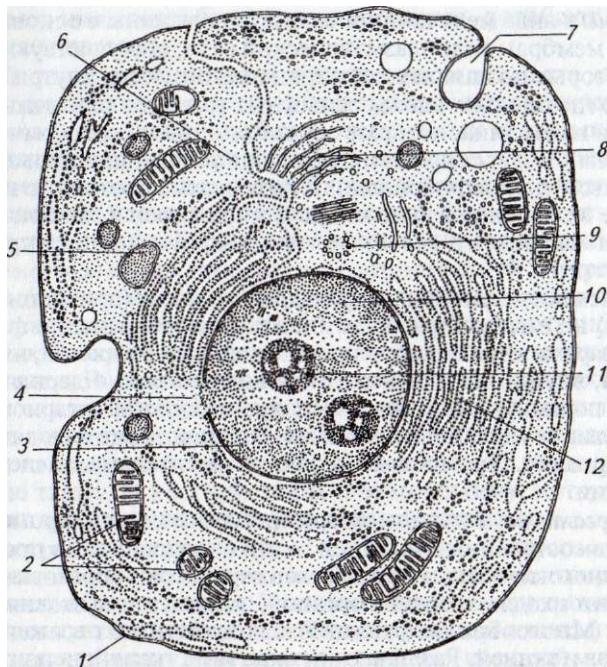


Рис. 3. Строение клетки эукариот:
1 — плазматическая мембрана; 2 — митохондрии; 3 — кариоплазма; 4 — ядерная оболочка;
5 — лизосома; 6 — цитоплазма; 7 — пиноцитозный пузырек; 8 — аппарат Гольджи;
9 — центриоли; 10 — ядро; 11 — ядрышко; 12 — эндоплазматическая сеть

Плазматическая мембрана представляет собой сложный белко-во-липидный комплекс. Она очень тонка и не только защищает клетку от внешних воздействий, но и участвует в обмене веществами между клеткой и окружающей средой.

Цитоплазма — это сложная коллоидная система, в которой находятся структурные образования, такие как митохондрии, эндоплазматическая сеть, аппарат Гольджи, рибосомы и растворенные вещества.

Митохондрии, имеющие вид мелких удлиненных телец, служат энергетическими центрами клетки, регулирующими биохимические реакции превращения энергии.

Эндоплазматическая сеть, штэндоплазматический ретикулум, представляет собой систему тончайших трубочек, пронизывающих всю цитоплазму клетки, и пузырьков. По тончайшим каналцам эндоплазматической сети осуществляется внутриклеточный обмен веществ.

Аппарат Гольджи близок по своему строению к строению эндоплазматической сети и служит для временного хранения продуктов внутриклеточного синтеза (в основном гормонов и ферментов) и передачи их через эндоплазматическую сеть для вовлечения в обменные процессы всего организма.

Рибосомы в виде мельчайших зерен расположены в основном на поверхности мембран эндоплазматической сети; они участвуют в синтезе белков, которые по каналам сети транспортируются внутри клетки.

Часто в цитоплазме клеток животных имеются различные тончайшие нити и волоконца, которые служат опорным каркасом клеток (тонофибриллы), могут сокращаться (миофибриллы) или проводить нервные импульсы (нейрофибриллы). В цитоплазме постоянно наблюдают временные включения в виде капелек жира, зерен и глыбок резервных белков, пигментов и т. д., которые возникают и исчезают в процессе обмена веществ.

Лишь немногие специализированные клетки (эритроциты млекопитающих) не имеют ядра. *Ядра* клеток разнообразны по форме и величине. Снаружи ядро ограничено двухслойной мембраной, или ядерной мембраной, внутри которой находится кариоплазма. Ядерная мембрана пронизана порами, от которых в сторону цитоплазмы и кариоплазмы отходят небольшие тончайшие каналцы. В кариоплазме находятся хромосомы и ядрышко. Хромосомы являются носителями наследственной информации.

Ткани организма. В животном организме все клетки, кроме половых, находятся в составе тканей. Ткани — это сложившиеся в процессе филогенеза многоклеточных организмов структуры, образованные клетками. Ткани входят в состав органов, участвуя в выполняемых ими функциях. Многообразие функций тела животного отражено в строении органов и тканей. Различают четыре типа тканей: нервную, эпителиальную, соединительную и мышечную.

Нервная ткань воспринимает и передает раздражения, поступающие как из внешней, так и из внутренней среды организма. Раздражимость — одно из свойств, характеризующих живую материю. Нервная

II .in I, состоит из нервных клеток, или нейронов, клеток глии и межклеточного вещества. В зависимости от выполняемой функции нейроны делятся на чувствительные и двигательные. Каждый нейрон имеет • •/•nil или несколько отростков. Короткие разветвленные отростки называются *дendritами*, а один длинный отросток называется *нейритом* и *in аксоном*. Концевые разветвления аксона чувствительной клетки поглощают раздражение и называются *рецепторами*. Их много на поверхности тела и во внутренних органах. От рецептора возбуждение передается по аксону к телу нейрона, а затем по его дендритам — к дендритам двигательного нейрона. В результате возникает соответствующая реакция — двигательная, секреторная и т. п.

Эпителиальная ткань представляет собой пластины клеток, плотно прилегающих друг к другу и соединенных межклеточными контактами. Под эпителиальным пластом располагается слой межклеточного вещества, называемого базальной мембраной. Эпителий бывает однолистовым (его клетки лежат в один ряд) и многослойным (его клетки располагаются в несколько рядов). Для эпителия характерна высокая способность к регенерации, поскольку его клетки из-за своего положения (на поверхности тела или на внутренней поверхности полых органов, например пищевода) быстро изнашиваются, погибают и должны заменяться новыми.

Кожный эпителий находится в постоянном контакте с внешней средой, что определяет его специфику у разных видов животных. Например, кожа рыб обильно снабжена слизистыми клетками, а у насекомых эпителий пропитан хитиновым веществом, защищающим тело от высыхания. Вместе с тем энтодермальный эпителий кишечника выложен в один слой клетками цилиндрической формы, через этот эпителий осуществляется избирательное всасывание переваренной пищи.

Соединительная ткань относится к системе тканей внутренней среды. Для соединительной ткани характерно наличие большого количества межклеточного вещества и сравнительно небольшого числа клеток. Их межклеточное вещество плотное, а в крови — жидкое. Соединительная ткань выполняет многообразные функции: трофическую (связанную с питанием организма), опорную, защитную и др. В зависимости от выполняемых функций соединительную ткань делят на: 1) собственно соединительную ткань, 2) жировую ткань, 3) кровь, 4) хрящевую ткань, 5) костную ткань и 6) мезенхиму, которую можно рассматривать как соединительную ткань зародыша.

В состав крови входят ее жидкая часть (плазма) и форменные элементы — клетки крови. Форменные элементы у животных разных типов существенно различаются. У позвоночных это эритроциты (красные кровяные клетки), лейкоциты (белые кровяные клетки) и тромбоциты. Главным органом кроветворения является костный мозг. Особая роль принадлежит эритроцитам, поглощающим кислород и транспортирующим его к тканям. Лейкоциты наиболее разнообразны у позвоночных по форме и по выполняемым функциям, среди которых важное место принадлежит нейтрализации ядов и чужеродных тел.

Наряду с кровеносной системой у позвоночных имеется *лимфатическая система*. Представлена она лимфатическими узлами и лимфатическими сосудами. Лимфоциты, циркулирующие по лимфатической системе, играют важную роль в защитной функции организма.

Ретикулярная ткань в виде рыхлого скопления звездчатых клеток является основой селезенки. Наличие большого количества лимфоцитов способствует процессам фагоцитоза, определяя защитные функции этой ткани.

Рыхлая волокнистая соединительная ткань входит в состав многих органов и подкожной клетчатки. Плотная волокнистая соединительная ткань составляет основу связок (эластиновые волокна) и нижнего слоя кожи (коллагеновые волокна).

Хрящевая ткань входит в состав скелета позвоночных и ряда беспозвоночных животных. Она составляет основу ушных раковин, образует межпозвоночные диски. Из костной ткани сформированы кости. Костная ткань состоит из клеток и межклеточного вещества, построенного из коллагеновых волокон и аморфной массы, пропитанной минеральными солями. Костная ткань — это живая ткань; в ней находятся кровеносные сосуды и нервы. Кости в организме служат депо кальция, фосфора и других минеральных элементов.

Мышечная ткань. Мышцы делятся на гладкие и поперечно-полосатые. Основу мышц составляют тончайшие мышечные волоконца — миофибриллы. Гладкие мышцы характеризуются плавностью сокращения и расслабления. Они находятся во внутренних органах. Поперечно-полосатые мышцы способны совершать быстрые сокращения и выносить большую нагрузку. При этом скорость их сокращения может колебаться в значительных пределах. Поперечно-полосатые мышцы обычно прикреплены к костям наружного или внутреннего скелета и относятся к мышцам произвольного сокращения.

Из тканей формируются различные органы многоклеточного животного. Функционирование любого органа происходит в тесном взаимодействии со всеми другими органами, что свидетельствует о целостности всего организма. То же самое наблюдается и в деятельности различных органелл у одноклеточных животных; особенно четко это прослеживается у высших простейших — инфузорий. Каждый орган функционирует как неразрывная часть единого организма.

Особенности строения и функционирования различных органов изучаются в разделах систематики животных.

Размножение — это свойство живых организмов воспроизводить себе подобных особей. Животные размножаются бесполым и половым путем. *Бесполое размножение* характерно в основном для низших животных. При бесполом размножении от материнской особи либо отделяется часть ее тела, либо вся материнская особь делится на две или большее число частей. При этом каждая часть в дальнейшем развивается в самостоятельное животное. Существует несколько способов бесполого размножения: деление, почкование и шизогония.

При размножении *простым делением* ма ге р и некая особь делится на две одинаковые дочерние особи (рис. 4). У одних простейших (жгутиконосцы) тело делится п продольном направлении, у других (инфузории) — в поперечном, у третьих, облагающих шарообразной или изменчивой формой (например, амебы), деление может происходить в любом направлении.

При *почковании* на теле материнского организма образуется вырост — почка, которая постепенно приобретает форму и строение взрослой особи. После отделения (ограничения) от материнского организма новая (дочерняя) особь начинаетести самостоятельную жизнь. При почковании у многих видов кишечнополостных животных, ведущих сидячий образ жизни, в результате неполного отделения дочернего организма (почки) от материнского и сохранения связи с последним образуются колонии. При этом могут возникать целые коралловые «города» — рифы.

У ряда паразитических форм простейших наблюдается множественное деление — *шизогония*. В этом случае ядро материнского организма многократно делится и образуется многоядерный *шизонт*. Вокруг каждого ядра внутри шизонта обособляется участок цитоплазмы. Шизонт распадается (делится) на многочисленные мелкие дочерние особи — *мерашиты*. Такое множественное деление позволяет паразиту в очень короткий срок достичь высокой численности в организме своего хозяина.

Половое размножение свойственно всем типам животных. При размноженииовым путем новый организм развивается из зиготы, которая образуется в результате слияния женской (яйца) и мужской (спермии) половых клеток.

Спермий — мужская гаплоидная половая клетка — обычно состоит из головки, шейки и хвоста. Последний служит для передвижения спермии в жидкой или вязкой среде (рис. 5).

Яйцеклетка — женская половая клетка — имеет округлую форму и состоит из цитоплазмы и ядра. По своим размерам яйцеклетка пре-восходит спермий во много раз.

У некоторых животных наблюдается наружное оплодотворение, т. е. женские и мужские гаметы выделяются в воду, где и происходит их слияние. Другим животным свойственно внутреннее оплодотворение: спермии в составе спермы вводятся в половые пути самки, где и происходит их слияние с яйцеклеткой.

Имеют место случаи развития организмов из неоплодотворенных яйцеклеток. Такое размножение, называемое *партеногенетическим* или *девственным* размножением, часто встречается у членистоногих.

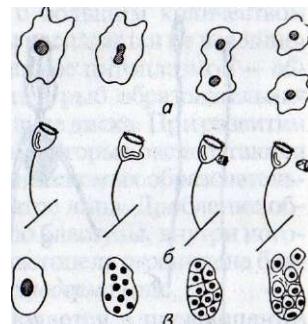


Рис. 4. Бесполое размножение одноклеточных животных:
а — деление амебы; б — почкование инфузории суворки;
в — шизогония малярийного плазмодия

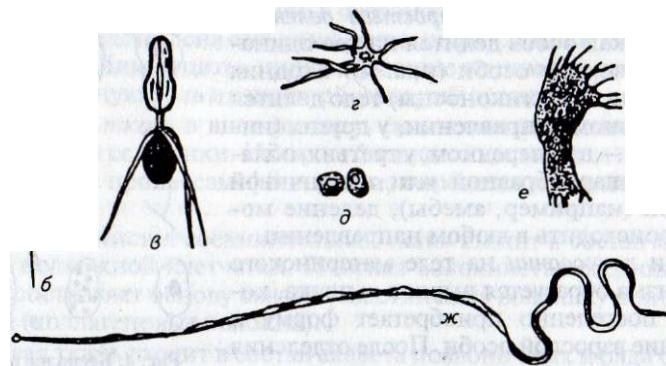


Рис. 5. Различные формы спермиев животных:
а — улитки; *б* — медузы; *в* — рака; *г* — круглого червя; *д, е* — низших раков; *ж* — жука

Начальные этапы развития многоклеточных животных. Развитие организма начинается с делений дробления зиготы (оплодотворенного яйца) на ряд клеток — *blastomeres*. Дробление бывает полным и неполным, что обусловлено количеством желтка в яйце. Полное дробление происходит тогда, когда в яйце содержится мало желтка и он распределен в цитоплазме относительно равномерно (такие яйца называют *гомолецитальными*). При полном дроблении все яйцо делится сначала на 2 blastomera, затем на 4, 8, 16, 32 и т. д. В результате такого полного дробления яйца образуется комочек из клеток — *морула*. В последующем в центре морулы возникает полость и морула превращается в однослоистый зародыш — *blastula* (рис. 6).

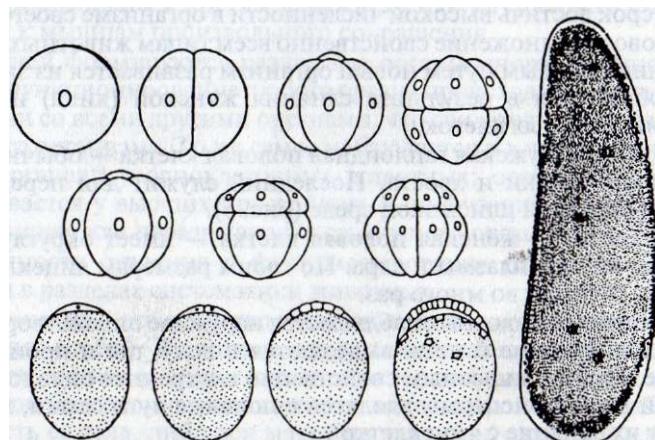


Рис. 6. Типы дробления яйца:
а — полное равномерное; *б* — полное неравномерное; *в* — дискоидальное; *г* — поверхностное

Неполное дробление наблюдается в яйцах с большим количеством яиц. При этом типе дробления на бластомеры распадается не все яйцо, а лишь его часть, где содержится ядро, окруженное цитоплазмой — оболочкой новотельной плазмой. Например, в яйцах птиц и рыб образовательная яйцеклетка расположена на одном из полюсов яйца в виде диска. При развитии зародыша этот диск дробится на ряд бластомеров, которые располагаются в один слой, лежащий на массе желтка. В яйцах насекомых образовательная плазма окружает желток, находящийся в центре яйца. Дробление оболочки новотельной плазмы приводит к возникновению бластулы, внутри которой имеется первичная полость — бластоцель. Бластоцель ограничена бластодермами, расположенными в один слой — бластодермой (рис. 7).

Следующий этап развития зародыша заключается в превращении однослоистого зародыша — бластулы — в двухслойный — гаструлу. Этот процесс называется гаструляцией, который у разных животных протекает неодинаково. Но как бы ни шел процесс гаструляции, он

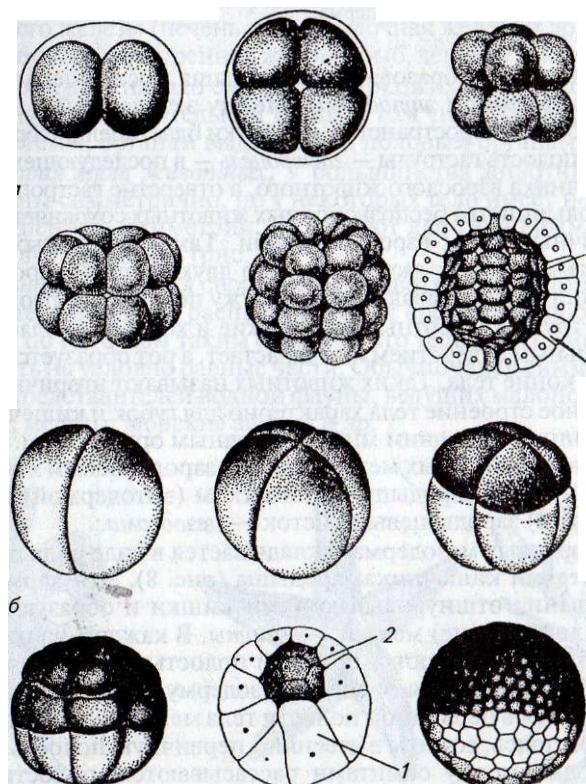


Рис. 7. Схема полного дробления яйца до стадии гаструлы:
а — равномерное дробление яйца голотурии; б — неравномерное дробление яйца лягушки; 1 — бластодерма; 2 — бластоцель

Рис. 8. Продольные и поперечный разрезы через различные эмбриональные стадии ланцетника:

a — бластула; *b, e, g* — гастрula на разных стадиях развития; *d, e, ж* — образование мезодермы, хорды и нервной системы; *f* — анистомальный полюс; *2* — вегетативный полюс; *3* — гастрапльная полость; *4* — гастропор; *5* — первая трубка; *6* — нервно-кишечный канал; *7* — невропор; *8* — складка мезодермы; *9* — целомические мешки; *10* — хорда; *11* — место будущего рта; *12* — место будущего заднего прохода; *13,14* — мезодермальные карманы; *15* — эктодерма; *16* — энтодерма

всегда заканчивается образованием зародыша, состоящего из двух слоев клеток: *эктодермы* и *энтодермы*. Между эктодермой и энтодермой остается небольшое пространство — остатки бластоцеля. Образовавшаяся кишечная полость гаструллы — *гастроцель* — в последующем станет полостью кишечника взрослого животного, а отверстие гастроцеля — *блестопор* — у большинства беспозвоночных животных сохраняется и становится первичным ртом взрослой особи. Таких животных называют первичноротыми. Итак, гастрula — это двухслойный зародыш с кишечной полостью, открывающейся наружу первичным ртом.

У другой группы животных (иглокожие и хордовые) бластопор становится анальным отверстием или застает, а рот образуется на противоположном конце тела. Таких животных называют вторичноротыми.

Двухслойное строение тела характерно для губок и кишечнополостных, относящихся к низшим многоклеточным организмам. У высокоорганизованных животных между первым зародышевым листком (эктодермой) и вторым зародышевым листком (энтодермой) образуется средний (третий) зародышевый листок — *мезодерма*.

У части хордовых мезодерма закладывается в виде ряда парных выпячиваний стенки кишечника зародыша (рис. 8). Эти карманообразные выпячивания отшнуровываются от кишки и образуют мезодермальные (целомические) мешки — *сомиты*. В каждом из них имеется полость, которую называют вторичной полостью тела или *целомом*, а стенки сомитов представляют собой мезодерму. Таким образом, сомиты располагаются в первичной полости тела между эктодермой и энтодермой. Разрастаясь, сомиты вытесняют первичную полость тела, а когда перегородки между сомитами рассасываются, полости сомитов сливаются, образуя вторичную полость тела. Остатки первичной полости сохраняются в виде лакун и каналов. Животные, имеющие целом, называются вторичнopolостными.

Первичнополостные животные не имеют кровеносных сосудов и функцию крови в их организме выполняет полостная жидкость, омывающая внутренние органы.

У примитивных вторичнополостных животных (моллюски, членистоногие) первичная полость тела частично сливается с целомом, образуя смешанную полость тела — миксоцель. Кровеносная система у таких животных незамкнутая, т. е. кровь у них движется то по сосудам, то по лакунам.

В процессе развития эмбриона из зародышевых листков путем дифференцировки образуются ткани и органы. В формировании эпителиальной ткани участвуют все три зародышевых листка. Эктодерма дает начало наружным покровам и их производным (кожные железы, волосы, перья, чешуя, когти), наружному скелету беспозвоночных животных, эпителию переднего и заднего отделов пищеварительной системы, нервной системе и органам чувств, мочевым протокам, висцеральному скелету и наружным жабрам.

Из энтодермы образуются средний (пищеварительный) отдел кишечника и все его железы (печень, поджелудочная железа и др.), хорда, плавательный пузырь и внутренние жабры у рыб, легкие у высших животных.

Мезодерма дает начало скелету у иглокожих и позвоночных животных, мышцам, соединительной ткани, в том числе крови, части кровеносной системы, органам выделения и половым органам.

Симметрия тела животных. У большинства животных части тела расположены симметрично, но у некоторых тело имеет неправильную форму, лишенную симметрии. Различают два типа симметрии: радиальную и двустороннюю (рис. 9).

Радиальная симметрия свойственна животным, у которых одинаковые части тела и органы располагаются от срединной продольной оси животного по радиусам таким образом, что тело таких животных можно разделить на условно равные части. Обычно такая симметрия характерна для представителей водной фауны, ведущих малоподвижный образ жизни: медузы, морские звезды и др.

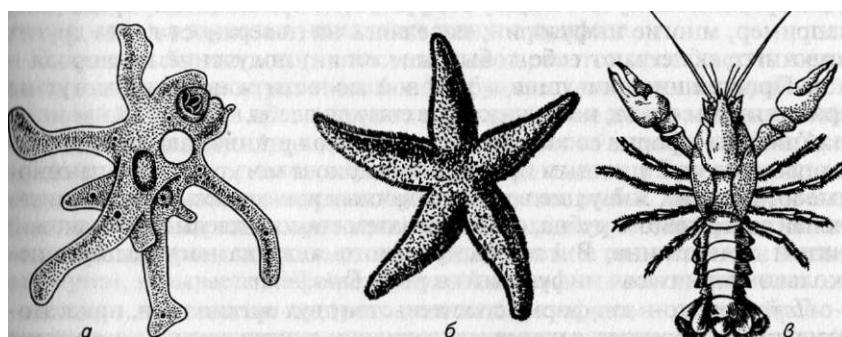


Рис. 9. Типы симметрии:
а — асимметричная (амеба); б — радиальная (морская звезда); в - - двусторонняя (речной рак)

Двусторонняя симметрия характеризуется тем, что тело животного может быть разделено плоскостью только на две равные половины, т. е. у таких животных можно различить левую и правую половины или передний и задний концы тела. Этот тип симметрии всегда бывает относительным, так как в расположении многих внутренних органов ее нельзя достигнуть (сердце у приматов лежит в левой части грудной клетки, а у кур-несушек яичник расположен в левой части таза и т. п.).

Тип симметрии является важным систематическим признаком различных групп животных. Встречаются случаи, когда с возрастом животного меняется симметрия его тела. Например, плавающие личинки морских ежей обладают двусторонней симметрией, а взрослая особь — радиальной.

Симбиоз и паразитизм в животном мире. Отношения животных, обитающих в разных условиях среды, строятся на прямом, непосредственном или косвенном взаимном влиянии. Питание — основной фактор, который определяет взаимоотношения животных. Именно на этой основе складываются пищевые связи между животными и растениями, а также между разными видами животных. По характеру питания одни животные являются плотоядными, другие — растительноядными, третьи — всеядными.

Животные делятся также на *монофагов* и *полифагов*. Монофаги питаются растениями или животными, принадлежащими к одному виду (тля филлоксера питается только соком виноградной лозы). Для полифагов пищей служат растения или животные разных видов (полевка обыкновенная поедает до 100 видов растений). Многие животные используют в пищу существенно ограниченный круг животных или растений. Это так называемые *олигофаги* (например, комары рода анофелес питаются кровью только крупных стадных млекопитающих).

В процессе эволюции органического мира возникли различные формы тесного сожительства разных видов животных, животных и растений. Существуют три формы такого сожительства: комменсаллизм, симбиоз и паразитизм.

Комменсаллизм — такая форма сожительства, при которой только один организм получает пользу от другого, не причиняя ему вреда. Так, например, многие инфузории, поселяясь на поверхности тела других животных, облегчают себе добывание пищи, получение кислорода и т. п. Простейшие, живущие в ротовой полости животных, могут не причинять им вреда, но извлекать пользу для себя.

Симбиоз — форма сожительства, когда оба организма получают взаимную пользу. Типичным примером симбиоза могут служить панцирные инфузории, живущие в рубце жвачных животных: питаясь бактериями содержимого рубца, они сами затем становятся высокобелковой пищей для хозяина. В 1 см³ содержимого желудка насчитывают несколько сотен тысяч инфузорий из рода *Entodinium*.

Паразитизм — это форма сожительства двух организмов, при котором паразит поселяется на поверхности или внутри организма хозяина и питается частями его тела или продуктами пищеварения хозяина. Наружные паразиты называются *экто паразитами*, а паразитирующие

ми у гри организма хозяина — эндопаразитами. Паразитами и их хозяевами могут быть как растения, так и животные. В отличие от хищников паразиты, существуя за счет хозяина, обычно не убивают его, но могут и способствовать его гибели, подрывая его здоровье. Как правило, паразиты по размерам всегда меньше своего хозяина и его организация в равнении с организацией свободноживущих сородичей резко упрощена; особенно это касается нервной системы, органов движения и выхания. Обычно паразит и его хозяин принадлежат не только к разным видам, но даже к разным типам и полцарствам животного мира.

Паразитические животные произошли, вероятно, от свободноживущих предков в процессе постепенного и длительного приспособления к жизни в других организмах. Зачастую предками паразитов были коменсальсы и симбионты, которые постепенно переходили к питанию исключительно за счет хозяев. Огромная масса свободноживущих организмов, которые с пищей попадали в пищеварительный тракт животных, погибли. Но какая-то часть из них, приспособившись к жизни в кишечнике, перешла к паразитическому образу жизни. Из кишечного тракта паразиты могли попадать в кровь, лимфу, различные органы и ткани, становясь постоянными обитателями организма хозяина. Некоторые эктопаразиты, проникав через кожный покров тела хозяина, могли превратиться в илопаразитов. Таким образом, переход свободноживущих форм к паразитическому образу жизни мог быть осуществлен разными путями.

Формы и проявления паразитизма в животном мире разнообразны. Ирременный паразитизм характеризуется тем, что паразит вступает в контакт с хозяином только тогда, когда принимает пищу: кровососущие насекомые (комары, слепни, блохи, постельный клоп и др.). В остальное время эти паразиты остаются свободноживущими.

Постоянному паразитизму свойственно то, что паразит проводит на теле хозяина или внутри него всю жизнь на всех стадиях своего развития или только на строго определенной стадии развития. Например, опаснейший паразит круглый червь трихинелла ни на одной стадии развития не покидает своего хозяина — крысу, свинью, человека и др. У оводов же только личинки проходят свое развитие в организме хозяина (лошадь, корова, овца), а взрослые насекомые живут в природе.

Жизненный цикл многих постоянных паразитов протекает не в одном, а в двух и даже в трех хозяевах. Возбудитель малярии малярийный плазмодий обитает в крови человека и в организме комара. Ленточный червь лентец широкий в половозрелом состоянии живет в кишечнике хищных животных и человека, а личиночные стадии его развития проходят в организмах мелких ракообразных и затем рыбы. Таким образом, развитие этого паразита проходит со сменой хозяев. Хозяин, в организме которого паразит достигает половой зрелости и размножается половым путем, называется *дeфинитивным* или *окончательным* хозяином.

Хозяин, в организме которого этот же паразит размножается бесполым путем или проходит стадии своего развития, не достигая половой зрелости, называется *промежуточным*. Следовательно, пораженный широким лентецом человек является дефинитивным хозяином, в ко-

тором происходит половое размножение паразита; мелкие ракообразные — промежуточным, а рыба — дополнительным.

Обитая в организме хозяина, паразитические животные оказывают на него разносторонние и обычно отрицательные воздействия: питаясь за счет организма хозяина, паразит истощает его; выделяемые паразитом продукты обмена вызывают у хозяина токсикозы; механически повреждая ткани и органы, паразиты способствуют проникновению в организм хозяина патогенных микроорганизмов и пр.; паразиты могут разрушать ткани и органы хозяина, закупоривать воздухоносные пути, просвет кишечника, выводные протоки желез; ослабленный паразитами организм хозяина подвержен всевозможным инфекциям.

Паразитарные заболевания, возбудителями которых являются животные организмы, называются *инвазионными* в отличие от *инфекционных* заболеваний, возбудителями которых служат вирусы, бактерии и грибы.

В фауне нашей планеты описано более 80 тыс. видов паразитических животных, из которых половина приходится на перепончатокрылых насекомых — наездников. На втором месте по численности находятся паразитические круглые и плоские черви, и лишь на третьем месте — паразитические простейшие. Среди хордовых паразитов очень мало.

Паразиты локализуются во всех системах органов животных, преобладая в пищеварительной. У растений животные-паразиты, а это в основном нематоды, локализуются также во всех системах органов, но главным образом в корневой системе. Организм хозяина для паразита — это среда его обитания. Внешняя же среда также воздействует на паразита, но через изменения внутренней среды организма хозяина. Удивительную чувствительность к состоянию организма демонстрируют даже эктопаразиты. Паразитическая моногенетическая trematoda многоустка начинает свое развитие на жабрах головастика. Развитие паразита и хозяина — головастика — происходит синхронно. Как только головастик превращается в лягушонка, паразит переходит в мочевой пузырь животного. Если развитие головастика по какой-либо причине задерживается, то замедляется и развитие многоустки.

Распространению паразитов способствуют *резервуарные* хозяева, попадая в организм которых паразиты не развиваются, а накапливаются, сохраняя свою жизнеспособность. Такие резервуарные хозяева известны для многих паразитов из разных классов. Например, установлена роль земляных червей в распространении аскарид: черви заглатывают яйца аскарид и в их кишечнике из яиц выходят личинки, сохраняющие жизнеспособность до года. Съедая такого дождевого червя свиньи или другие животные заражаются аскаридами.

Система животного мира. Животный мир отличается большим разнообразием, на Земле насчитывается около 2 млн видов. Создание системы животного мира является важнейшей задачей зоологии. Решением ее занимается одна из отраслей этой науки — систематика (таксономия), которая разрабатывает теорию и практику классификации и определения животных. Без систематики и ее конечного результата — классификации — все огромное разнообразие видов воспринималось

бы как хаос, недоступный пониманию. Естественная система животного мира строится на основе всестороннего изучения животных, что позволяет выявить не только черты сходства и различия между ними, но и доказать их исторические связи и установить степень родства.

Основной таксономической единицей в систематике является *вид* (*species*) — реально существующая категория. Вид — это обособленная группа сходных особей, обитающих на определенном пространстве (ареале), свободно скрещивающихся между собой и дающих плодовое потомство. Особи разных видов, как правило, между собой не скрещиваются, но если такое скрещивание произойдет, то полученное потомство обычно не способно к дальнейшему размножению.

Каждый вид населяет определенное пространство, называемое областью распространения вида или его ареалом. Особи одного вида, населяющие различные районы ареала, находятся обычно в неодинаковых условиях среды, что приводит к возникновению изменчивости, т. е. приобретению этими особями своеобразных черт. Такие несколько отличные местные группы особей вида, населяющие часть ареала, называются *подвидами*. В отличие от видов подвиды связаны друг с другом переходными формами, обладающими признаками промежуточного характера.

У большинства видов сельскохозяйственных животных выделяют породы, выведенные человеком и отличающиеся, в первую очередь, продуктивностью, но также экsterьерными и интерьерными показателями. Например, во всем мире насчитывают около 400 официально зарегистрированных пород домашних собак.

В современной систематике принято именовать различные виды животных на латинском (или латинизированном греческом) языке, что делает эти названия интернациональными. Впервые двойное (бинарное) название ввел великий шведский ученый К. Линней еще в XVIII в. В соответствии с правилом бинарной номенклатуры каждому виду присваивается название, состоящее из двух слов, первое означает род, второе — собственно вид. Например, различные виды кошек составляют один род *Felis*. Отдельные же виды этого рода будут называться уже двумя словами: например, кот лесной — *Felis silvestris*, кот степной — *F. libysca*, кот камышовый — *F. chaus* и др. После названия вида животного обычно указывается фамилия (полностью или сокращенно) ученого, впервые описавшего данный вид, и год, когда это было сделано. Например, собака домашняя *Canis familiaris* L., 1758., в данном случае L. — это К. Линней.

В современной систематике животных используются следующие таксономические группы (таксоны). Близкие виды объединяются в род (*genus*), близкие роды — в семейство (*familia*), семейства — в отряд (*ordo*), отряды — в класс (*classis*), классы — в тип (*phylum*). Типы образуют царство (*regnum*) животных. Часто устанавливаются промежуточные категории — подрод (между родом и видом), подсемейство (между семейством и родом), подотряд (между отрядом и семейством), подкласс (между классом и отрядом), подтип (между типом и классом). Кроме того, выделяют надсемейства (между семейством и подотрядом), надотряд (между отрядом и подклассом), надкласс (между классом и подтиповом).

Высшая систематическая категория — это тип. Каждый тип характеризуется определенным планом строения, общим для всех групп, входящих в его состав, и общим происхождением.

Тип подразделяется на следующие основные категории: ТИП—подтип—над класс—КЛАСС—подкласс—надотряд—ОТРЯД—подотряд—над семейство—СЕМЕЙСТВО—подсемейство—РОД—подрод—ВИД—подвид.

Царство Животные (Animalia, или Zoa)

Подцарство Одноклеточные, или Простейшие (Protozoa)

- Тип Саркомастигофоры (Sarcomastigophora)
- Тип Апикомплексы (Apicomplexa)
- Тип Миксоспоридии (Mixozoa)
- Тип Микроспоридии (Microspora)
- Тип Асцетоспоридии (Ascetospora)
- Тип Лабиринтулы (Labyrinthomorpha)
- Тип Инфузории (Ciliophora)

Подцарство Многоклеточные (Metazoa)

Надраздел Фагоцителлозои (Phagocytellozoa)

- Тип Пластинчатые (Placozoa)

Надраздел Паразои (Parazoa)

- Тип Губки (Porifera, или Spongia)

Надраздел Эуметазои (Eumetazoa)

Раздел Лучистые (Radiata)

- Тип Кишечнополостные (Coelenterata)
- Тип Гребневики (Ctenophora)
- Тип Мезозой (Mesozoa)

Раздел Двустороннесимметричные (Bilateria)

- Тип Плоские черви (Plathelminthes)
- Тип Круглые черви (Nemathelminthes)
- Тип Немертины (Nemertini)
- Тип Кольчатые черви (Annelida)
- Тип Моллюски (Mollusca)
- Тип Онихофоры (Onychophora)
- Тип Членистоногие (Arthropoda)
- Тип Погонофоры (Pogonophora)
- Тип Щупальцевые (Tentaculata)
- Тип Щетинкочелюстные (Chaetognatha)
- Тип Иглокожие (Echinodermata)
- Тип Полухордовые (Hemichordata)
- Тип Хордовые (Chordata)

Современная систематика выделяет большое число типов животных. В данном учебнике дается описание только тех типов, которые имеют важное значение либо для познания эволюции животных, либо с практической точки зрения.

Царство Животные (Animalia) делят на два полцарства: Простейшие, или Одноклеточные (Protozoa) и Многоклеточные (Metazoa).

ПОДЦАРСТВО ОДНОКЛЕТОЧНЫЕ, ИЛИ ПРОСТЕЙШИЕ (Protozoa)

К одноклеточным относят животных, у которых тело морфологически соответствует одной клетке, но одновременно представляет самостоятельный организм со всеми присущими живому существу функциями. Известно значительное число представителей простейших, образующих колонии из нескольких или многих клеток, но эти Простейшие не могут быть отнесены к многоклеточным организмам, поскольку каждая клетка такой колонии выполняет все функции, хотя в некоторых случаях и намечается разделение отдельных функций между клетками колонии. В многоклеточном же организме каждая клетка выполняет определенную функцию — двигательную, нервную и т. п.

Деление клеток у многоклеточных животных приводит к росту организма, а деление клетки у простейших приводит к увеличению их численности, т. е. деление у простейших — это по сути дела размножение этих организмов.

Известно более 39 тыс. видов одноклеточных. Это мелкие организмы. Минимальными размерами характеризуются простейшие, ведущие паразитический образ жизни внутри клеток растений и животных (эндопаразиты) — всего 2—4 мкм. Тело простейших ограничено снаружи тончайшей мембраной (или более плотной и эластичной пелликулой), под которой находятся цитоплазма и ядро (одно или несколько). Цитоплазма представлена двумя слоями: наружным светлым и плотным — эктоплазмой, и внутренним менее плотным с многочисленными включениями — эндоплазмой. В эндоплазме сосредоточены все основные органеллы клетки: митохондрии, рибосомы, лизосомы, аппарат Гольджи, эндоплазматическая сеть и пр.

У простейших имеются специальные органеллы: пищеварительные и сократительные вакуоли, опорные и сократительные фибриллы.

Простейшие, тело которых ограничено мембраной, не имеют постоянной формы (амебы). У ряда видов клеточная мембра уплотняется за счет эктоплазмы и становится плотной и эластичной, образуя так называемую пелликулу. В этом случае животные имеют определенную форму тела (инфузории) и одновременно сохраняют достаточную гибкость. Часть одноклеточных имеет постоянную форму тела благодаря укреплению оболочки за счет различных включений.

Функцию скелета у простейших могут выполнять раковины, формирующие наружный скелет, или специальные иглы и капсулы, формирующие внутренний скелет. Раковины образуются из веществ, выделяемых эктоплазмой, а внутренний скелет возникает в эндоплазме клетки. Основу скелетных образований составляют органические и минеральные вещества (CaCO_3 , SiO_2 , SrSO_4).

Самый простой способ движения простейших, не имеющих постоянной формы тела, — движение с помощью ложноножек, или псевдоподий (амебоидное движение). Псевдоподии — это выросты клетки, в которые перетекает цитоплазма. Более сложное движение характерно для простейших, обладающих жгутиками или ресничками. Строение жгутиков и ресничек сходно, но для жгутиков свойственно вращательное движение, а для ресничек — гребной тип движения. Внутриклеточные паразиты, как правило, не имеют органелл, обеспечивающих движение.

Разнообразен тип питания простейших. Среди них встречаются автотрофы, которые способны к фотосинтезу (одноклеточные жгутиконосцы). Но большая часть простейших — гетеротрофы, питающиеся готовыми органическими веществами. Одним простейшим свойствен голозойный способ питания путем проглатывания оформленных частиц пищи, другим — сапрофитный способ за счет поглощения растворенных органических соединений.

Когда в клетку простейшего поступают оформленные пищевые частицы, вокруг них образуются пищеварительные вакуоли, в которых эти частицы перевариваются. Такой захват частиц клеткой получил название фагоцитоза. При сапрофитном способе питания пищеварительных вакуолей в организме простейших не образуется. Захват клеточной поверхностью растворенных органических веществ называется пиноцитозом.

Небольшое число простейших обладает смешанным (миксотрофным) типом питания. В одних условиях они способны к фотосинтезу, в других — к питанию органическими веществами, т. е., имея в цитоплазме хлорофилловые зерна, они могут образовывать и пищеварительные вакуоли.

У пресноводных простейших процессы осморегуляции и выделения осуществляются с помощью сократительных вакуолей. У паразитических и морских форм сократительные вакуоли отсутствуют, так как среда, в которой обитают эти животные, и их внутреннее содержимое изотоничны. Выделение продуктов обмена у большинства простейших происходит через поверхность клетки, а также через сократительные вакуоли. Кислород поступает в клетку путем диффузии через клеточную мембрану.

В цитоплазме большинства простейших находится одно, реже два или несколько ядер, которые регулируют обмен веществ и размножение. Ядра одноклеточных имеют те же структуры и компоненты, что и ядра клеток многоклеточных животных, хотя и характеризуются морфологическим многообразием. У части многоядерных простейших ядра выполняют различные функции: репродуктивные и вегетативные. Такое явление называют ядерным дуализмом (у инфузорий). При бесполом размножении деление ядра у одноклеточных происходит по типу митоза

(непрямого деления), позволяющего сохранять преемственность в ряду Клеточных поколений. Ядра простейших, которым свойствен половой **процесс**, делятся путем мейоза, или редукционного деления.

Размножаются простейшие бесполым и половым путями. Бесполое размножение происходит путем деления клетки на две или множество **клеток**. Половой процесс заключается в образовании половых клеток — **гамет** (женских — макрогамет и мужских — микрогамет) и их слияния. В результате образовавшаяся зигота дает начало новому дочернему организму. У инфузорий половой процесс происходит путем коньюгации — слияния генеративных ядер двух особей, а не половых клеток.

Жизнь одноклеточных животных представлена рядом стадий, которые чередуются с определенной закономерностью. Период жизни организма между двумя одинаковыми стадиями называется жизненным циклом данного вида. Обычно жизненный цикл начинается со стадии зиготы (соответствует оплодотворенной яйцеклетке многоклеточных Животных), затем следуют стадия бесполого размножения делением, стадия образования половых клеток-гамет, которые, сливаясь попарно, дают новую зиготу. При бесполом размножении жизненный цикл — это период от деления до деления, только при половом размножении **жизненный цикл** — это период от зиготы до зиготы.

Важнейшая биологическая особенность одноклеточных — образование цист, или инцистирование. Для сохранения жизнеспособности в неблагоприятных для организма условиях животные округляются, теснясь воду, образуют плотную оболочку и переходят в состояние покоя, в таком состоянии простейшие могут длительный период сохранять жизнеспособность, пассивно перемещаться на большие расстояния с воздушными массами, водой и т. п., а в благоприятных условиях вновь перейти к активному образу жизни.

Одноклеточные животные приспособлены к обитанию в разнообразных средах, но для этих животных, как правило, необходимо наличие воды: морские и пресные водоемы, влажные почвы; есть одноклеточные, перешедшие к паразитическому образу жизни в растениях, животных и человеке.

Подцарство Protozoa делят на семь типов, из которых наибольший интерес представляют следующие: Саркомастигофоры (*Sarcomastigophora*), Апикомплексы (*Apicomplexa*), Миксоспоридии (*Mixozoa*), Микроспоридии (*Microspora*) и Инфузории (*Ciliophora*).

ТИП САРКОМАСТИГОФОРЫ (*Sarcomastigophora*)

К саркомастигофорам относят свободноживущих или паразитических одноклеточных животных, которые передвигаются с помощью особых временных выростов цитоплазмы (псевдоподий) или бичевидных выростов (жгутиков). Некоторые одноклеточные могут перемещаться как с помощью псевдоподий, так и с помощью жгутиков. Большинству

видов одноклеточных свойственно только бесполое размножение. Для некоторых простейших свойствен половому процессу путем копуляции.

Тип *Sarcomastigophora* представлен двумя подтипами: Жгутиконосцы (*Mastigophora*) и Саркодовые (*Sarcodina*).

Жгутиконосцы, видимо, стоят ближе к предковым группам простейших. Они разнообразнее по типам питания, органелл движения, типам оболочек клеток и т. п. О первичности жгутиковых форм свидетельствует то, что саркодовые, которые размножаются половым путем, проходят жгутиковую стадию гамет. Среди жгутиконосцев есть переходные формы между одноклеточными растительными и животными организмами.

ПОДТИП ЖГУТИКОНОСЦЫ (*Mastigophora*)

Жгутиконосцы обитают в морских и пресных водах, в почве и в организме растений и животных; среди них есть опасные паразиты животных и человека. Насчитывают более 8 тыс. видов жгутиконосцев. Растительные и животные жгутиконосцы являются важным звеном в пищевых цепях водных экосистем. Некоторые жгутиконосцы находятся в симбиотических отношениях с различными животными.

Представители жгутиконосцев характеризуются наличием особых органелл — жгутиков, которые служат для передвижения; число жгутиков колеблется от 1,2,4,8 до нескольких тысяч. При этом жгутики имеются постоянно в течение большей части жизненного цикла. У некоторых видов жгутиконосцев помимо жгутиков могут иметься временные или постоянные ложноножки — псевдоподии; этот признак сближает их с представителями подтипа Саркодовые.

Размеры и форма тела жгутиконосцев разнообразны. Они покрыты довольно плотной и сложной по строению оболочкой — пелликулой, что позволяет им сохранять более или менее постоянную форму. Цитоплазма делится на два слоя: эктоплазму и эндоплазму. У видов, способных образовывать псевдоподии, тело покрыто тонкой и эластичной мембраной. У растительных жгутиконосцев оболочка может состоять из клетчатки; у некоторых видов образуется панцирь разнообразной формы, нередко несущий отростки.

От переднего полюса тела отходят жгутики; если жгутиков много (несколько тысяч), то они могут покрывать все тело простейшего. У некоторых жгутиконосцев жгутик тянется вдоль тела, соединяясь с ним с помощью тонкой цитоплазматической мембранны — ундулирующей мембранны. Эта мембрана обеспечивает поступательное движение простейшего в вязкой среде, которое как бы ввинчивается в эту среду. Нижняя часть жгутика, погруженная в эктоплазму, называется базальным тельцем или кинетосомой.

Размножаются жгутиконосцы путем продольного деления клетки на две дочерние; у некоторых представителей существует половому процессу с образованием гамет и последующей их копуляцией.

Среди жгутиконосцев есть автотрофы, которые способны к фотосинтезу, гетеротрофы, характеризующиеся животным типом питания, И, наконец, миксотрофы, сочетающие растительный и животный способы питания. Гетеротрофным жгутиконосцам свойственно либо голозойное (анимальное) питание путем заглатывания частиц органической пищи, либо сапрофитное — за счет всасывания жидкой органической пищи всей поверхностью тела.

Жгутики служат не только для движения, но и помогают захватывать пищевые частицы. В результате движения жгутика в воде возникает водоворот, увлекающий мелкие пищевые частицы к основанию жгутика, где у некоторых видов находится клеточный рот, ведущий в глотку. У видов, не имеющих клеточного рта, у основания жгутика есть участок липкой цитоплазмы, не покрытый пелликулой, через которую пища попадает в организм одноклеточного. Поступившая в цитоплазму пища заключается в образовавшиеся пищеварительные вакуоли. Непереваренные остатки пищи выбрасываются из тела простейшего во внешнюю среду в любом участке клетки.

КЛАСС РАСТИТЕЛЬНЫЕ ЖГУТИКОНОСЦЫ (*Phytomastigophorea*)

Для представителей этого класса характерен автотрофный или миксотрофный типы питания, реже среди растительных жгутиконосцев встречаются виды с гетеротрофным типом питания. Обитают эти жгутиконосцы в соленой и пресной воде. У них имеются хроматофоры, содержащие хлорофилл, а многие имеют светочувствительный глазок — стигму, позволяющий выбирать наиболее освещенные участки водоема для оптимизации процессов фотосинтеза. У пресноводных форм имеется сократительная вакуоль. Встречаются колониальные формы. Колонии образуются при неполном делении, в результате которого не полностью отделившиеся друг от друга особи остаются связанными друг с другом. Колонии могут различаться по форме (шаровидные, древовидные) и по характеру развития (монотомические и палинтомические колонии).

При монотомическом развитии после бесполого размножения путем деления дочерние клетки растут и снова периодически делятся, тем самым увеличивая число особей в колонии, которая в свою очередь периодически делится пополам. При палинтомическом развитии все клетки колонии или только их часть последовательно делятся, но без стадий роста и увеличения их объема, в результате чего образуется сразу несколько молодых колоний. Материнская колония затем распадается на дочерние, число которых соответствует числу клеток старой материнской колонии.

Половой процесс распространен преимущественно у растительных форм, имеющих палинтомический тип колоний. Наиболее сходен с половым процессом многоклеточных половой процесс у колоний *Volvox* (рис. 10). В колонии только немногие клетки дают начало мужским (микрогаметы) и женским (макрогаметы) гаметам. Среди вольвоксовых встречаются раздельнополые виды, в колонии которых обра-

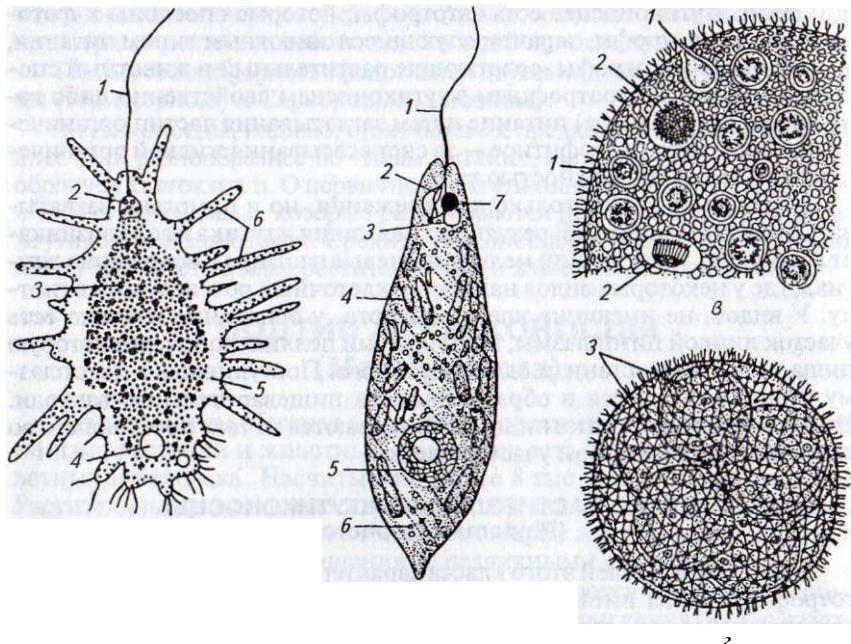


Рис. 10. Растительные жгутиконосцы:
 а — жгутиконосец *Mastigamoeba aspera* (по Шульце); 1 — жгутик; 2 — ядро; 3 — эндоплазма; 4 — сократительная вакуоль; 5 — эктоплазма; 6 — псевдоподии; б — эвглена зеленая (*Euglena viridis*) (по Дофлейну); 1 — жгутик; 2 — резервуар сократительной вакуоли; 3 — сократительная вакуоль; 4 — хроматофоры; 5 — ядро; 6 — зерна парамиля; 7 — глазок; в — вольвокс (*Volvox globator*), участок колонии с половыми клетками (по Кону); 1 — макрогаметы; 2 — микрограметы; 3 — дочерние колонии

зуются или только мужские, или только женские гаметы, и герmafродитные виды, в колонии которых одновременно развиваются и мужские, и женские гаметы. Макрогаметы неподвижны. Микрограметы отыскивают их и сливаются с ними. Оплодотворенная макрогамета (зигота) дает начало новой колонии путем последовательных плинтомических делений.

К растительным жгутиконосцам относятся: хризомонады, панцирные жгутиконосцы, эвгленовые и вольвоксовые (рис. 10). Растительные свободноживущие морские и пресноводные жгутиконосцы входят в состав планктона, среди них имеются симбионты коралловых полипов и паразитические виды. Пресноводные окрашенные жгутиконосцы при массовом размножении вместе с микроскопическими водорослями могут вызывать «цветение» воды в пресных водоемах, вследствие чего иногда возникают ночные заморы рыбы.

КЛАСС ЖИВОТНЫЕ ЖГУТИКОНОСЦЫ (Zoomastigophorea)

Всем животным жгутиконосцам свойствен гетеротрофный тип питания. Большая их часть являются паразитами растений и животных. Особенно опасны эндопаразиты животных и человека, относящиеся к отряду Кинетопластиды (Kinetoplastida). В плазме крови животных и человека паразитируют различные виды трипаносом (*Trypanosoma*), имеющих лентовидное тело с одним (реже с двумя) жгутиком (рис. 11). В эктоплазме жгутик связан с кинетопластом, а снаружи свободным концом направлен вдоль тела жгутиконосца, срастаясь с ним с помощью цитоплазматической ундулирующей мембранны. На переднем конце жгутик остается свободным. У ряда форм трипаносомных жгутиконосцев жгутик может отходить от середины клетки, у некоторых он начинается на переднем конце тела, а у лейшманий жгутик отсутствует.

Трипаносомы паразитируют в основном в крови и спинномозговой жидкости животных и человека, вызывая тяжелейшие заболевания, называемые трипаносомозами. В тропической Африке трипаносомы *Trypanosoma rhodesiense* и *T. brucei gambiense* вызывают «сонную болезнь» человека. Это длительное и тяжелое заболевание, унесшее свыше миллиона жизней, начинается небольшой лихорадкой, сопровождается сонливостью и постепенно приводит к полному истощению организма человека. Без лечения болезнь заканчивается смертью.

Переносчиками возбудителей сонной болезни являются кровососущие мухи цеце (*Glossina palpalis* и *Gl. morsitans*). Вместе с кровью больного муха засасывает трипаносом, которые размножаются в ее кишечнике и [акапливаются в слюнных железах и хоботке насекомого, являющемся одновременно переносчиком и вторым хозяином паразита. При укусе мухой цеце здорового человека трипаносомы вместе со слюной попадают в его кровь. Природным резервуаром трипаносом являются антилопы и другие животные, почти не страдающие от этих жгутиконосцев, но являющиеся их носителями.

В Южной Америке трипаносома *T. cruzi* вызывает у людей болезнь Чагаса. Переносчиком и вторым хозяином возбудителя являются кровососущие клопы. Трипаносомы выделяются с экскрементами клопа и, попадая в ранку на коже человека, могут заразить его. Паразиты живут в крови, а затем проникают в клетки внутренних органов, где размножаются и снова попадают в кровь.

Есть немало видов трипаносом, вызывающих тяжелые заболевания у крупного рогатого скота и верблюдов. В Африке *T. brucei* поражает рогатый скот, вызывая болезнь Нагана. Переносчиком возбудителя

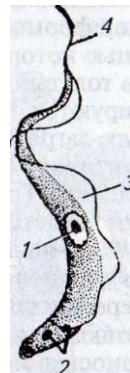


Рис. 11. Трипаносома:
1 ~ ядро; 2 ~ базальный диск;
3 ~ ундулирующая мембра--/
4 ~ жгутик

являются мухи цеце. В Южной Азии и Африке кровососущие слепни переносят трипаносому *T. evansi* — возбудителя болезни верблюдов. *T. equiperdum* вызывает случную болезнь лошадей, заражение которых происходит при случке. Паразиты поражают нервную систему животных. Случная болезнь встречается в Средиземноморье, Азии и Африке.

Среди представителей Kinetoplastida есть родичи трипаносом — лейшмании (*Leishmania*), вызывающие лейшманиоз у человека. Это внутриклеточные паразиты с редуцированным жгутиковым аппаратом. Переносчиком лейшманий являются кровососущие москиты, в кишечнике которых паразит размножается и проходит стадию со жгутиком, но без ундулирующей мембранны. При укусе москитами в кровь человека попадают лейшмании, имеющие жгутики. После внедрения в клетки печени и селезенки паразиты утрачивают жгутик.

Один вид лейшманий — *L. donovani* — вызывает у человека заболевание, называемое висцеральным лейшманиозом (кала-азар). Распространенное в Средней Азии, Иране, Южной Америке и Индокитае это заболевание поражает в основном детей, у которых увеличиваются в размерах печень и селезенка. Болезнь сопровождается лихорадкой, истощением и малокровием. Природным резервуаром паразита являются в основном бродячие собаки.

В средней Азии и Закавказье *L. tropica* вызывает восточную язву, или пендинку. Переносчиком и вторым хозяином являются москиты рода *Phlebotomus*, в желудке которых лейшмании размножаются, образуя жгутиконосную форму. В местах укусов москитами образуются изъязвления. Внутри лейкоцитов, находящихся в язве, обнаруживается множество паразитов, лишенных жгутика. Через 1—2 года язвы зарубцовываются. Носителем кожного лейшманиоза могут быть различные грызуны, чаще всего большие песчанки. Висцеральный и кожный лейшманиозы дают стойкий иммунитет.

В кишечнике и желчных протоках человека паразитирует лямблия *Giardia intestinalis*, вызывая болезнь лямблиоз. Тело этого паразита грушевидной формы, имеет несколько жгутиков и вооружено присоской, с помощью которой лямблия прикрепляется к слизистой кишечника. Попав в толстый отдел кишечника, лямблии отбрасывают жгутики и инфицируются. Человек (чаще всего дети) заражается, потребляя пищу и воду, загрязненные цистами паразитов.

Среди отряда Трихомонадовые (Trichomonadida) есть опасные паразиты человека. Трихомонады имеют четыре—шесть жгутиков, из которых один является рулевым и образующим ундулирующую мембрану. *Trichomonas hominis* вызывает хронические поносы, *T. vaginalis* паразитирует в мочеполовых путях, вызывая трудноизлечимые заболевания.

Интересны представители отряда Многожгутиковые (Hypotermastigida), обитающие в кишечнике насекомых, в частности термитов. У этих жгутиконосцев много жгутиков, и они являются полезными симбионтами термитов. Без многожгутиковых хозяева не могут самостоятельно переваривать клетчатку, так как жгутиконосцы вырабатывают фермент целлюлазу, расщепляющую потребленную термитами клетчатку до легко усвояемых углеводов.

ПОДТИП САРКОДОВЫЕ (*Sarcodina*)

Представители саркодовых на протяжении жизненного цикла или большей его части передвигаются с помощью псевдоподий; жгутиками саркодовые могут обладать лишь на кратковременных стадиях развития, это гаметы и зооспоры. Основная масса саркодовых размножается бесполым путем: делением на две клетки или на множество клеток. Половое размножение присуще немногим видам и осуществляется путем слияния гамет. Большинство саркодовых — свободноживущие виды, обитающие в соленых водах, часть из них живет в пресных водоемах, застеляет почву, участвуя в почвообразовательных процессах, и немногие являются паразитами животных и человека. Всего насчитывают около 10 тыс. видов саркодовых.

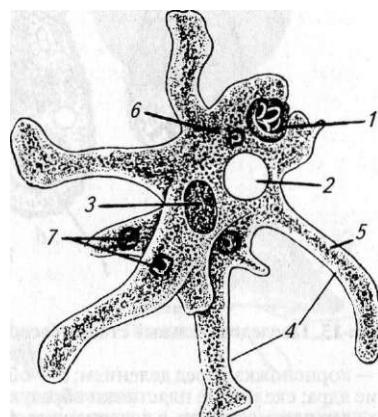
Из 12 классов саркодовых можно выделить три класса, к которым принадлежит основная масса представителей этого подтипа: класс Корненожки (*Rhizopoda*), класс Лучевики (*Radiolaria*) и класс Солнечники (*Heliozoa*).

Наиболее просто устроены представители голых амеб (класс Корненожки — *Rhizopoda*, отр. Амебы — *Amoebina*), населяющие пресные водоемы и почвы, где они питаются мелкими простейшими, одноклеточными водорослями, микроорганизмами и гниющими остатками. Тончайшая мембрана, покрывающая тело этих микроскопических животных, позволяет им образовывать временные выросты — псевдоподии, по своей форме напоминающие корни растений, что и определило название класса. С помощью псевдоподий амебы медленно перетекают с одного места на другое. При этом они обтекают мелкие пищевые частицы (одноклеточные водоросли, бактерии и др.) со всех сторон, и те оказываются внутри цитоплазмы амебы, где образуются пищеварительные вакуоли. С помощью пищеварительных ферментов пищевые частицы перевариваются (внутриклеточное пищеварение). Жидкие продукты переваривания поступают в эндоплазму, а непереваренные остатки транспортируются к поверхности тела и выбрасываются наружу через мембрану клетки. Типичным представителем отряда является пресноводная амeba (*Amoeba proteus*) (рис. 12). Подобный способ захвата

Рис. 12. Амеба (*Amoebaproteus*), захватывающая пищу:

1 — захваченная псевдоподиями пищевая частица; 2 — сократительная вакуоль; 3 — ядро; 4 — псевдоподии; 5 — эктоплазма; 6 — эндоплазма; 7 — пищеварительные вакуоли

3 - 6407



33

пищевых частиц с помощью псевдоподий называют фагоцитозом. Наряду с ним существует способ поступления жидких веществ в тело амебы — пиноцитоз. При этом внутрь цитоплазмы впячивается тонкий канал, в который засасывается капелька жидкости с растворенными в ней органическими веществами. Образовавшаяся вокруг этой капельки вакуоль с жидкостью отшнуровывается от канала и после всасывания жидкости эта вакуоль прекращает свое существование.

Для поддержания осмотического давления у амеб, особенно обитающих в пресных водах и почве, есть особый аппарат для удаления излишков воды из организма — сократительная вакуоль; обычно сократительная вакуоль бывает одна, реже две. У морских и паразитических амеб сократительные вакуоли отсутствуют или пульсируют очень редко. Помимо регулирования осмотического давления сократительные вакуоли участвуют в процессах выделения продуктов обмена и дыхания, обеспечивая организм кислородом из окружающей воды.

Амебам присущее бесполое размножение путем деления на две клетки или на несколько дочерних особей (рис. 13).

При неблагоприятных условиях амебы инфицируются, выделяя вокруг тела плотную оболочку. При наступлении благоприятных условий среды цисты разрушаются и амебы начинают вести активный образ жизни.

В кишечнике позвоночных животных, в том числе домашних животных и человека, обитает множество видов амеб, не принося вреда своим хозяевам, а просто являясь их квартирантами. Большая их часть питается содержимым кишечника, в том числе бактериями (*Entamoeba coli*). Среди

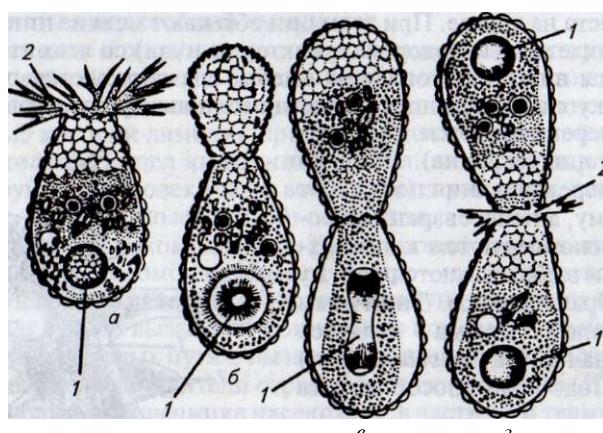


Рис. 13. Последовательные стадии бесполого размножения пресноводной корненожки *Euglypha alveolata*:

а — корненожка перед делением; *б* — образование цитоплазматической почки; *в* — деление ядра; скелетные пластинки образуют новую раковину; *г* — окончание деления; одно из ядер переместилось в дочернюю особь: 1 — ядро; 2 — псевдоподии

таких в толстом кишечнике человека, есть виды, которые могут вызвать тяжелые заболевания, например дизентерийная амеба *E. histolytica* (хронично пищей ей служат бактерии, но в ряде случаев эта амеба может паразитировать под слизистую оболочку кишечника, где питается и размножается, вызывая кровавый понос (кишечный амебиаз). Симптомы заболевания схожи с симптомами дизентерии, поэтому эту амебу называют дизентерийной. С каловыми массами наружу выходит множество цист, которые в наши времена остаются инвазионными (сохраняют способность к заражению). Некоторые люди могут быть носителями дизентерийных амеб.

У представителей отряда Раковинные амебы (Testacea) тело заключено в раковину, образованную органическими рогоподобными веществами, немыми цитоплазмой; зачастую в такую раковину включены песчинки и другие посторонние частицы. Раковины имеют отверстие — устье, из которого амебы выдвигают псевдоподии. Раковинные и голые амебы в Олипопом количестве населяют пресные водоемы, сфагновые мхи и почву, участвуя в процессах почвообразования. Благодаря своим микроскопическим размерам они способны существовать в тончайшем водном слое, окружающем частички почвы. При пересыхании почвы амебы инцистируются и в виде цист могут переноситься ветром с пылью на значительные расстояния. В благоприятных условиях почвенные амебы быстро размножаются и делением надвое: одна из клеток остается в материнской раковине, а другая строит себе новую раковину. У увлажненных и заболоченных почв наиболее многочисленны арцелла и диффлюгия (рис. 14).

Более сложно устроены обитатели морей из отряда Фораминиферы (Foraminifera), у которых раковина образуется из веществ (близких по

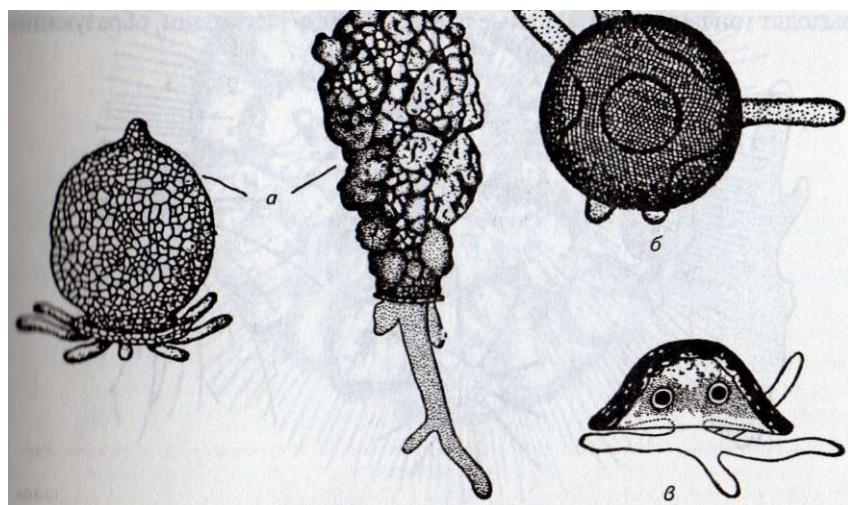


Рис. 14. Раковинные корненожки:
а — диффлюгия (*Diffugia*); б — арцелла (*Arcella*), вид сверху; в — арцелла, вид сбоку

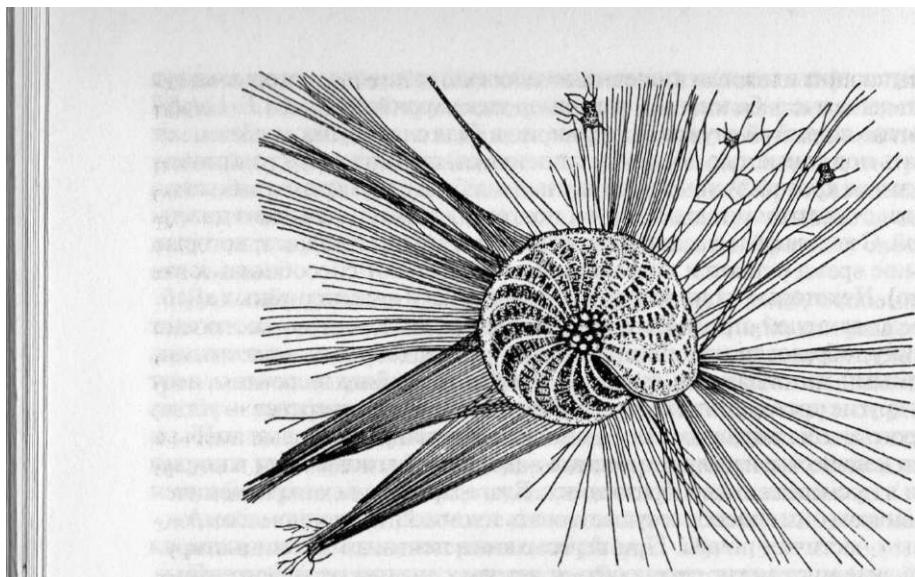


Рис. 15. Фораминифера. Раковина и сеть тончайших псевдоподий

природе к псевдохитину), выделяемых эктоплазмой. Это самая многочисленная и разнообразная группа саркодовых, встречающихся во всех морях на значительной глубине. У некоторых видов псевдохитиновые раковины инкрустированы песчинками, у других пропитаны углекислым кальцием (рис. 15). Форма раковин разнообразна, внутри они имеют одну или несколько сообщающихся камер, в которых находится тело корненожки. Кроме устья у раковины имеется множество пор, через которые наружу выходят тончайшие нитевидные псевдоподии — изоподии, образующие

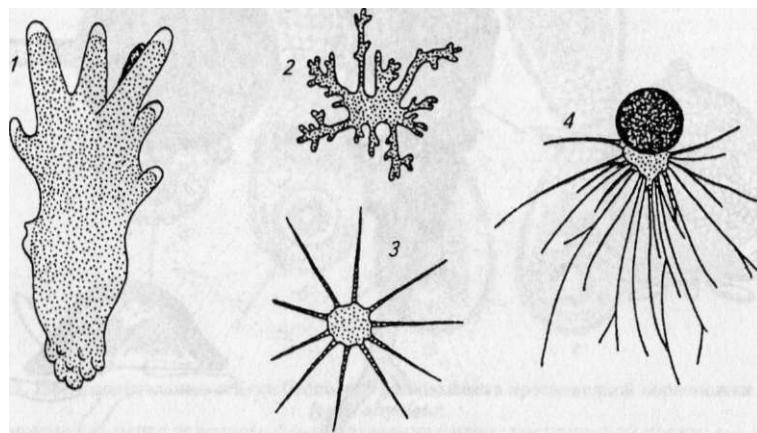


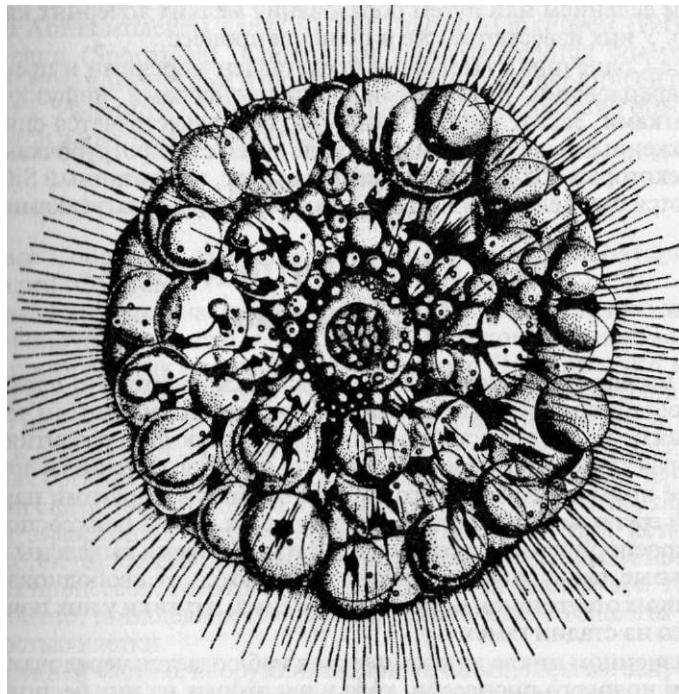
Рис. 16. Типы псевдоподий у саркодовых:
1 — лобоподии; 2 — ризоподии; 3 — аксоподии; 4 — филоподии

• и it ꙗї раковины сложную цитоплазматическую сеть; с помощью этой се-
м I прмсможки передвигаются и питаются (рис. 16). Пищей им служат
п,д ігрті, мелкие простейшие и даже некоторые многоклеточные.

II приду с бесполым размножением фораминиферы размножаются и
нши.иным путем. Сначала тело корненожки распадается на множество
нм>1н1||д||ых клеток, которые покидают материнскую раковину, растут
и пїаю вокруг себя новую раковину. Дочерние корненожки дают
иічнико /ругому поколению раковинных корненожек: путем множест-
и о деления образуют гаметы — мелкие клетки с двумя жгутиками.
'Ігрт I поры раковины одинаковые по форме и размерам гаметы выхо-
14 1 н иоду и попарно сливаются, образуя зиготу, которая дает начало
•____v поколению. Таким образом, в жизненном цикле фораминифер
Происходит чередование бесполого и полового размножения.

1.1нышая часть фораминифер живет в придонном слое морей и Океа-
нии, входя в состав бентоса и питаясь мелкими организмами. Немногие
шнин!, обладающие легкой раковиной, входят в состав планктона.

II иерхних слоях морей живут саркодовые со сложным внутренним
• и¹ не том (класс Радиолярии, или Лучевики — Radiolaria). Болынинст-
1н и I них имеет окружной формы тело, от которого в виде лучей отходят
ммочисленные тонкие псевдоподии (рис. 17). Часто у радиолярий в



І'm 17. Радиолярия *Thalassophysa pelagica*. В центре видны крупное ядро и центральная капсула

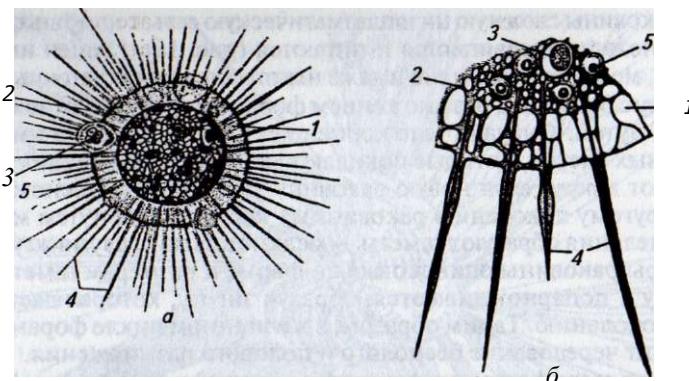


Рис. 18. Солнечник *Actinosphaerium eichhorni*:
а — общий вид; б — участок тела при большом увеличении; 1 — эктоплазма; 2 — эндоплазма; 3 — пищевой комок; 4 — аксоподии; 5 — ядро

цитоплазме находятся симбионты — водоросли, снабжающие хозяина кислородом и сами служащие ему пищей. Радиолярии размножаются простым делением или путем образования мелких дочерних клеток зооспор. У них известен также и половой процесс.

Класс Солнечники (Heliozoa) представлен морскими и пресноводными саркодовыми, питающимися жгутиконосцами, инфузориями и коловратками. Захват пищи у солнечников осуществляется спирально расположенным псевдоподиями в виде лучей с микротрубочками (рис. 18). У некоторых солнечников имеется скелет, пропитанный SiO_2 . Размножаются они делением или образуют зооспоры со жгутиками.

ТИП АПИКОМПЛЕКСЫ (Apicomplexa)

Тип насчитывает около 4,8 тыс. видов исключительно паразитических простейших, среди которых много паразитов животных и человека. Основная масса этих простейших проходит особую фазу развития — спору, которая служит для перехода паразита от одного хозяина к другому.

Ранее представителей этого типа объединяли с другими паразитическими группами простейших, образующих споры (миксоспоридиями, микроспоридиями), которые в настоящее время выделены в самостоятельные типы. Апикомплексы отличаются от свободноживущих простейших отсутствием органелл движения, жгутики у них появляются только на стадии гамет.

В жизненном цикле апикомплексов наблюдается чередование бесполого и полового процессов, хотя у некоторых из них бесполое размножение отсутствует. Бесполое размножение осуществляется путем

Множественного деления — шизогонии, в результате чего образуются **меротиты**. Мерозоиты инфицируют здоровые клетки хозяина. В **последующем** новые поколения мерозоитов дают начало поколению **половых особей** — гамонтов, формирующих половые гаметы.

Половой процесс протекает в форме копуляции гамет, которые у Оомыиинства апикомплексов различаются по размерам, т. е. образуются **Микро-** и микрогаметы. Зигота одевается плотной оболочкой и называется **Ооцистой**. В ооцисте начинается процесс спорогонии — образование множества спорозоитов, которые находятся внутри спор, покрытых **ООственной оболочкой**. Образованием спорозоитов заканчивается ШИ ишший цикл апикомплексов.

Таким образом, шизогония ведет к увеличению числа паразитов в тканях хозяина, а спорогония способствует росту числа паразитов в период их расселения в виде ооцист со спорами. Ооцисты и споры покрыты плотными оболочками, защищающими спорозоиты от внешней среды.

В жизненном цикле части споровиков происходит смена хозяев. Метаюнты и спорозоиты для проникновения в клетки хозяина имеют особый апикальный комплекс органелл на переднем конце тела (отсюда и название типа — Апикомплексы), представляющий собой упругую ОНириль, которая проникает в клетку хозяина после растворения оболочки этой клетки особым секретом, синтезируемым паразитом.

Тип Апикомплексы делят на два класса: Перкинсеи (Perkinsea) и Споровики (Sporozoea). Споровики характеризуются совершенным **Апикальным** комплексом и наличием в отличие от перкинсеи полового процесса. Именно к споровикам относится большинство опаснейших **Паразитов** животных и человека.

КЛАСС СПОРОВИКИ (*Sporozoea*)

Класс Споровики включает два отряда: Грегарини (Gregarinida) и Кокцидии (Coccidia).

О **граде** Грегарини насчитывает более 500 видов — паразитов многих **Органов** позвоночных животных, в основном насекомых и кольчатых червей, а также водных моллюсков и иглокожих. Паразитируют в кишечнике, а также в полости тела и гонадах. Тело кишечных грегарин разд蹭ено на три участка: передний, средний и задний (эпимерит, протомерит и дейтомерит). Грегарини, паразитирующие в полости тела и в половых органах, не обладают трехчленностью, их тело червеобразной или сферической формы. Грегарини — эндопаразиты, характеризующиеся анаэробным (бескислородным) дыханием, при котором паратикоген расщепляется с выделением энергии, используемой для обменных процессов. Тело грегарин одето плотной пелликулой. Питаются опрофитно, поглощая растворенные органические вещества всей поверхностью клетки.

Передней частью, вооруженной крючочками, паразит закрепляется и стенке кишечника. Ядро находится в задней части клетки, длина которой может достигать 16 мм. Длина самых мелких видов не превышает

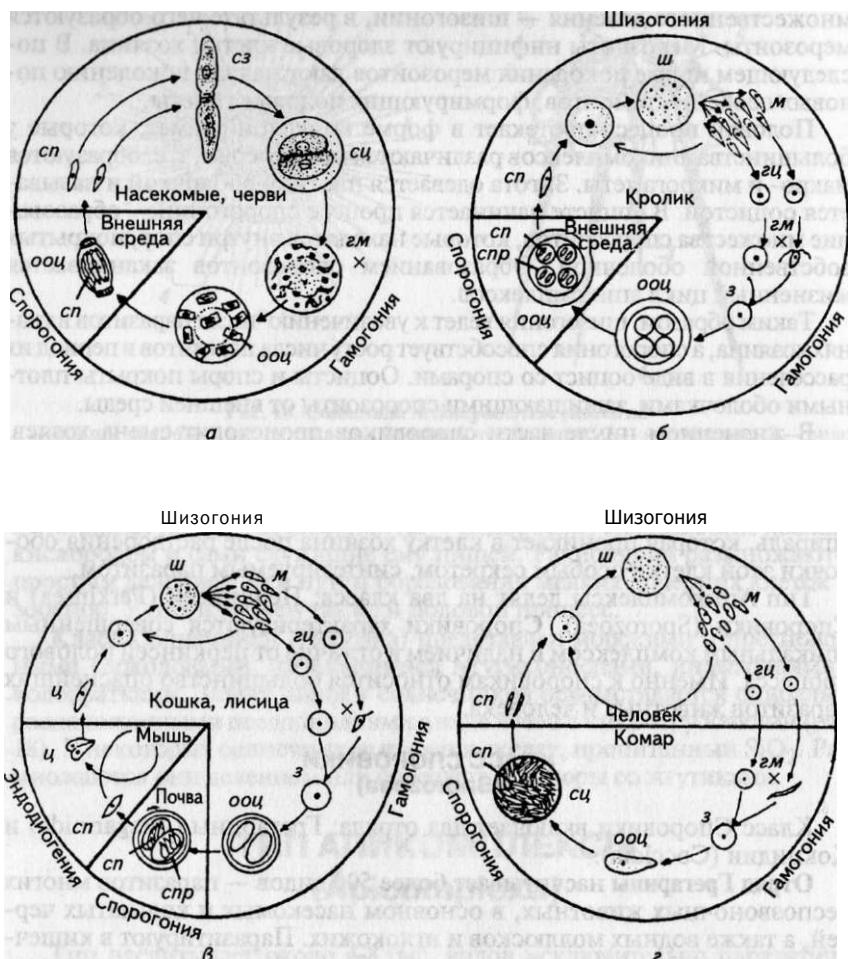


Рис. 19. Схемы жизненных циклов споровиков:
 a — грегарини; b — кокцидии; c — токсоплазмы; d — малярийного плазмодия; gm — гаметы; zg — гаметоциты; z — зигота; m — мерозоиты; oo — ооциста; cs — сизигий; sp — спорозоит; sp — спора; cp — спороциста; w — шизонт; u — цистозоит

15 мкм. Большинство грегарин размножаются половым путем, и лишь у немногих представителей происходит смена полового и бесполого поколений. Так, перед размножением грегарины, паразитирующие в кишечнике жука-чернотелки (рис. 19, 20), соединяются попарно в цепочку (сизигий), округляются и покрываются общей плотной оболочкой, образуя цисту. При этом слияния грегарин внутри цисты не происходит. Ядро каждой особи многократно делится. В каждой особи си-

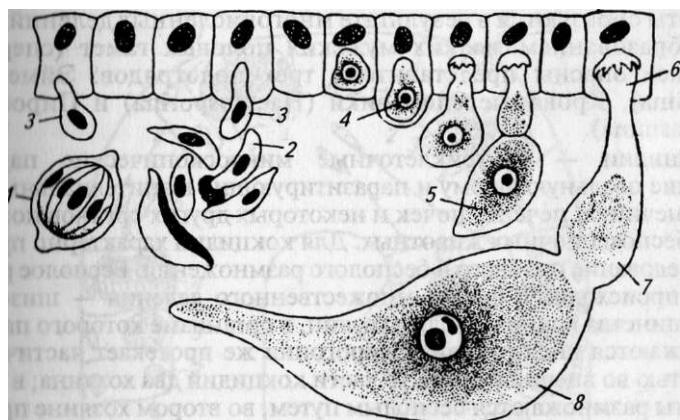


Рис. 20. Схема развития грегарини:
 1 — пора со спорозоитами; 2 — спорозоиты, вышедшие из споры; 3 — спорозоит, внедряющийся в эпителиальную клетку; 4, 5 — развитие спорозоита в грегарине; 6 — эпителий; 7 — протомерит; 8 — дейтомерит

ни ии образуется множество гамет. Микрогаметы имеют жгутики. Клетки, образовавшиеся в разных особях одного сизигия, попарно конкурируют. В результате копуляции гамет образуется зигота, которая окружает гаметы плотной оболочкой и инцистируется, превращаясь в ооцисту. Таким образом в одной цисте заключено несколько ооцист. Цисты, из которых находится множество ооцист, выводятся из тела хозяина по внешнюю среду. Это может произойти и после смерти самого хозяина. Дальнейшее развитие паразита происходит уже во внешней среде и под присутствием кислорода. Внутри каждой ооцисты протекает процесс деления, в результате которого образуется восемь спорозоитов. Эти спорозоиты становятся инвазионными, т. е. в каждой цисте под общей оболочкой находится множество инвазионных ооцист с восемью спорозоитами каждая. Ооцисты заглатываются новым хозяином с загрязненной пищей. В кишечнике хозяина спорозоиты покидают ооцисту, вне которой они в клетки кишечника, развиваются в них, а затем разрывают эти клетки и вырастают во взрослого паразита.

Таким образом, особенность рассмотренного жизненного цикла грегарини заключается в том, что внутри хозяина происходит только половое размножение, в во внешней среде — спорогония с формированием спорозоитов. Хозяин заражается, потребляя пищу, загрязненную цистами. В кишечнике нового хозяина оболочка цисты разрушается, из нее выделяются оболочки ооцист и из них выходят спорозоиты.

Отряд Кокцидии (Coccidia). В отличие от грегарин кокцидиообразные в основном внутриклеточные паразиты. У большинства происходит чередование полового и бесполого размножения. Макрофагамета (женская половая гамета, или яйцо) образуется непосредственно в процессе роста гамонта (гаметоцита) без деления. Микро-

рогаметы образуются в результате многочисленных делений гамонта с образованием мелких мужских половых гамет (спермии). Наиболее опасны представители трех подотрядов: Эймериевые (*Eimeriina*), Кровяные споровики (*Haemosporina*) и Пироплазмы (*Piroplasmida*).

Кокцидии — внутриклеточные микроскопические паразиты, имеющие овальную форму и паразитирующие в эпителиальных клетках кишечника, печени, почек и некоторых других органов позвоночных и беспозвоночных животных. Для кокцидий характерно правильное чередование полового и бесполого размножения. Бесполое размножение происходит в форме множественного деления — шизогонии. У большинства кокцидий один хозяин, в организме которого паразиты размножаются шизогонией. Спорогония же протекает частично или полностью во внешней среде. У части кокцидий два хозяина; в первом паразиты размножаются бесполым путем, во втором хозяине происходит половой процесс и спорогония.

Многие виды паразитических кокцидий из рода *Eimeria* приносят значительный ущерб животноводству. Паразитируя только у позвоночных животных, кокцидии чаще всего поражают кроликов, птиц разных видов, рогатый скот. При этом наиболее подвержен заболеванию молодняк животных. Кокцидии паразитируют в клетках кишечника и вызывают кровавый понос, приводящий к гибели большей части молодняка. Болезнь называется кокцидиозом.

Особый ущерб наносит *Eimeria magna*, поражающая кроликов, в организме которых ооцисты паразита попадают с загрязненными кормом и водой (рис. 21). В кишечнике кролика из ооцист выходят спорозоиты, которые внедряются в клетки стенок кишечника. В этих клетках кокцидии растут и размножаются бесполым путем посредством множественного деления — шизогонии. Дочерние особи носят название мерозоитов. Мерозоиты из пораженных клеток выходят в просвет кишечника, внедряются в здоровые клетки, растут и снова размножаются шизогонией. У *E. magna* развивается пять поколений мерозоитов. Последняя, пятая, генерация мерозоитов в клетках кишечника преобразуется в гамонты. Макрогамонты не делятся и дают начало макрогаметам (яйцам), микрогамонты путем деления образуют множество подвижных микрогамет с двумя жгутиками (спермии). Микрогаметы подвижны и выходят в просвет кишечника. Одна из микрогамет проникает в макрогамету и копулирует с ней. После копуляции гамет образуется зигота. Зигота одевается прочной оболочкой и превращается в ооцисту, которая выводится из кишечника кролика с испражнениями наружу.

Во внешней среде в присутствии кислорода в ооцисте проходит процесс спорогонии: сначала образуется четыре споробласта, которые покрываются собственными оболочками и превращаются в споры. В каждой споре формируется по два спорозоита. По окончании спорогонии споры становятся инвазионными. В каждой инвазионной ооцисте содержится по восемь спорозоитов. Если такая ооциста попадает в ки-

V

Рис. 21. Жизненный цикл кокцидий рода *Eimeria*:

I — мерное поколение шизогонии; **II** — второе поколение шизогонии; **III** — третье поколение шизогонии; **IV** — гамогония; **V** — спорогония; **1** — спорозоиты; **2** — одноядерный макрограмет; **3** — многоядерный шизоит; **4** — образование мерозоитов; **5** — мерозоиты; **f** — расщепление макрограмета; **7** — развитие микрограмета; **8** — ооциста; **9, 10** — образование (НОМ) пластов (видно остаточное тело); **II** — образование спор; **12** — зрелая ооциста с четырьмя спорами, в каждой из которых имеется по два спорозоита

III⁴ и I к кролику, то спорозоиты выходят из споробластов и ооцисты, **Мячики** **новый** цикл развития. Весь цикл развития этого паразита завершается за 7—8 сут, и, если не происходит повторного заражения, то организм **кролика** освобождается от кокцидий. Именно поэтому так важно принимать меры по исключению возможности повторной инвазии.

Для другого вида эймерии (*E. stiedae*) характерен такой же цикл размножения в организме кролика, но этот вид поражает клетки эпителия протоков печени, где протекает несколько циклов бесполого размножения шизогонией. Часто кролики одновременно поражаются двумя формами кокцидиоза — кишечным и печеночным, что увеличивает тяжесть заболевания животных. Каждому сельскохозяйственному животному свой вид кокцидий. Кокцидиозом болеет и человек. Не ме-

Рис. 22. Токсоплазма:
а — токсоплазмы, возникшие в результате продольного деления; б — циклы развития токсоплазм и разные способы заражения ими хозяев; 1 — кошка-хозяин, в которой проходит шизогония и стадии полового цикла; 2, 3, 4 — стадии развития ооцист, в каждой из которых в конечном счете развивается по две споры с четырьмя спорозоитами внутри; 5, 6 — мыши-хозяева, в которых протекает дополнительное бесполое размножение; 7 — внутриутробное заражение мышей

нее опасна кокцидия *E. tenella*, вызывающая опасное заболевание цыплят, приводящее к массовой гибели птицы. Болеют кокцидиозом телята (*E. ztimi*, *E. smithi*), карловые рыбы (*E. carpelli*).

Примерно такой же, как у кокцидий, жизненный цикл характерен для токсоплазмы (*Toxoplasma gondii*), вызывающей опасное заболевание у человека — токсоплазмоз. Однако жизненный цикл токсоплазмы усложнен сменой хозяев и появлением некоторых особенностей в размножении (рис. 22). Основным хозяином этого паразита являются кошки и другие виды кошачьих, в кишечнике которых токсоплазмы растут и размножаются сначала путем шизогонии, а потом и половым путем с образованием ооцист.

Ооцисты с испражнениями животного попадают во внешнюю среду, где в ооцистах происходит процесс спорогонии: в каждой образуются две споры с четырьмя спорозоитами. Инвазионные ооцисты со спорозоитами могут быть заглоchenы с загрязненными пищей и водой промежуточными хозяевами (грызуны, птицы). В кишечнике промежуточного хозяина из ооцист и спор выходят спорозоиты; они внедряются в ткани и проникают в кровяное русло. Паразиты могут оседать в любых тканях, в том числе и в мышцах, где они размножаются бесполым путем. Это особая форма деления, когда две дочерние особи образуются внутри материнского организма. Так возникают скопления паразитов, окруженные собственной общей оболочкой. Эти скопления токсоплазм в промежуточном хозяине носят названия цист.

При поедании зараженной цистами мыши в кишечнике кошки токсоплазмы выходят из цист, внедряются в эпителиальные клетки кишечника и начинают новый жизненный цикл. При общении с больной гоксоплазмозом кошкой человек может также стать промежуточным хозяином. Особенно велика вероятность заражения у детей, играющих и песочницах, где испражняются больные токсоплазмозом кошки. У человека токсоплазмоз протекает в легкой и тяжелой формах. В последнем случае возможен летальный исход.

Таким образом, особенностью токсоплазм является то, что источником инвазии служат не только ооцисты, но и ткани зараженного промежуточного хозяина, содержащие цисты токсоплазм. У млекопитающих токсоплазмы могут передаваться развивающемуся плоду через плаценту; это так называемый врожденный токсоплазмоз. В данном случае плод обычно погибает.

Подотряд Кровяные споровики (*Haemosporina*) представлен большой группой широко распространенных внутриклеточных паразитов крови, часть жизненного цикла которых протекает в эритроцитах млекопитающих и птиц. В отличие от кокцидий у кровяных споровиков спорогония никогда не протекает во внешней среде, а происходит в организме кровососущих насекомых, чаще комаров, которые одновременно являются и переносчиками паразитов. К этим паразитам относится возбудитель малярии, являющийся бичом населения многих тропических и субтропических стран.

В человеке паразитируют четыре вида плазмодия (род *Plasmodium*). Жизненный цикл малярийного плазмодия (*Plasmodium vivax*) типичен для остальных видов. Человек заражается при укусе комаром рода *Anopheles*, который в кровь человека вместе со слюной вносит спорозоитов малярийного плазмодия (рис. 23). С током крови спорозоиты достигают печени, где внедряются в паренхимные клетки печени, превращаются в шизонты и дают первое поколение путем шизогонии. Вышедшие из разрушенных клеток печени мерозоиты проникают в кровь и внедряются в эритроциты, где снова делятся шизогонией. Вышедшие из разрушенных эритроцитов мерозоиты (образуется 10—20 мерозоитов из одного шизонта) снова внедряются в здоровые эритроциты.

Продолжительность одного этапа шизогонии специфична для каждого вида плазмодия. У *PL malariae* промежутки между двумя последовательными бесполыми размножениями составляют 72 ч, поэтому заболевание получило название 4-дневной лихорадки. У наиболее широко распространенного вида *PI. vivax* этот промежуток равен 48 ч; это 3-дневная лихорадка. У *PI. falciparum* срок между двумя размножениями шизогонией составляет тоже около 48 ч, но промежутки между двумя приступами лихорадки сокращаются до 24 ч из-за периода высокой температуры тела у больного человека (тропическая лихорадка). Еще один вид плазмодия встречается лишь в тропической Африке — *PL ovale*.

Выход мерозоитов из эритроцитов сопровождается приступами лихорадки с повышением температуры, так как вместе с мерозоитами из

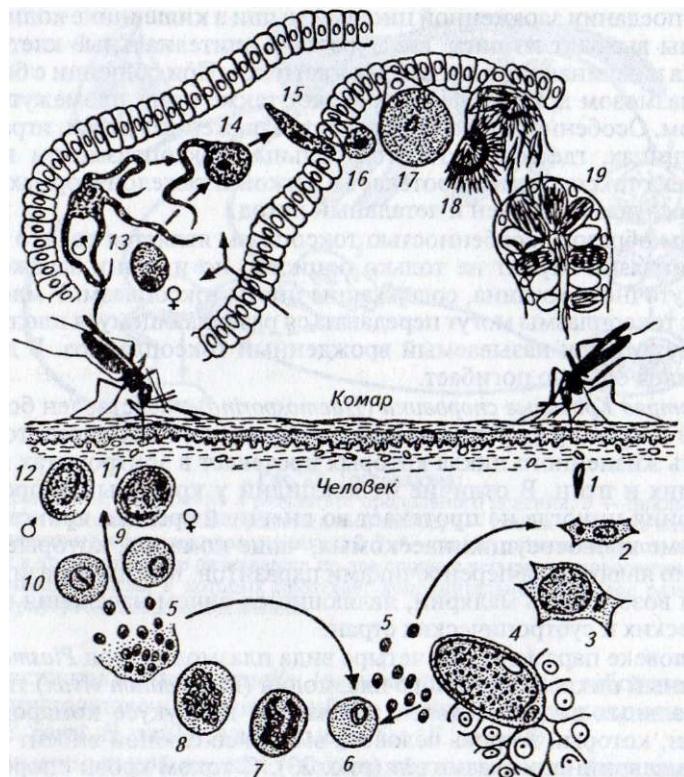


Рис. 23. Жизненный цикл малярийного плазмодия рода *Plasmodium*:
 1 — спорозоит; 2,3 — рост шизонта (агамонта); 4 — шизогония в клетках печени; 5 — мерозоиты; 6,7,8 — шизогония в эритроцитах; 9—12 — образование гамонтов (микро- и макрограмонтов); 13 — образование макрограмм и микрограмм; 14 — копуляция гамет; 15 — зигота (оокинета); 16—18 — спорогония и образование спороцисты со спорозоитами; 19 — накопление спорозоитов в слюнных железах комара

разрушенных эритроцитов в кровь поступают продукты обмена паразитов, вызывающие интоксикацию человека.

После нескольких циклов бесполого размножения шизогонией начинается подготовка к половому процессу. Внедрившиеся в эритроциты мерозоиты дают начало гамонтам, а не шизонтам, как при шизогонии. При этом образуются две группы гамонтов: макрограмм и микрограмм. Дальнейшего развития гамонтов в крови человека не происходит: человек становится носителем малярийного паразита.

У комара, напившегося крови больного малярией человека, в кишечнике из макрограмма формируется женская макрограмма, а из микрограмма образуется четыре—восемь мелких мужских микрограмм. После копуляции макро- и микрограмм образуется подвижная зи-

I o гп — оокинета. Оокинета внедряется в стенку кишечника и на ее внешней стороне в полости тела комара оокинета преобразуется в ооцисты. В ооцисте происходит процесс спорогонии с образованием множества (до 500 особей). Стенки ооцисты разрушаются и спорозоиты выходят из ооцисты и попадают в полость тела комара, откуда они проникают в слюнные железы и в ротные железы. При укусе человека комар со слюной вводит в кровь человека хоботок спорозоиты. Таким образом, в жизненном цикле малярийного плазмодия отсутствуют стадии развития, протекающие во внешней среде. Помимо этого на всем протяжении своего развития паразит не имеет стадий, когда он одевается защитными оболочками, что отличает его от кокцидий.

Малярия широко распространена на планете. У человека, больного малярией, наблюдается малокровие (анемия), интоксикация всего организма; болезнь сопровождается приступами лихорадки. От кровяных паразитов в мире погибло больше людей, чем во всех войнах. Переносит малярию комары рода *Anopheles*, в основном *A. maculipennis*. В Европе существует шесть видов-двойников малярийного комара, которых ранее объединяли в один вид.

Среди кровяных паразитов, вызывающих тяжелые заболевания многих диких млекопитающих и сельскохозяйственных животных, следует отметить пироплазмы (подотряд Пироплазмы — *Piroplasmina*). У них микроскопические паразиты эритроцитов. В эритроцитах пироплазмы размножаются бесполым путем с образованием двух клеток. При этом паразиты не разрушают гемоглобин, который у больных животных появляется в моче.

Жизнь этих паразитов протекает в двух хозяевах: млекопитающих и насекомых (паразитических) клещах. Нападая на больных пироплазмозом животных, клещи вместе с кровью получают и клетки паразита, которые размножаются в теле клещей бесполым путем и проникают в их слюнные железы. При укусе и кровососании клещи передают здоровому млекопитающему пироплазмы. Это заболевание широко распространено во многих странах. Пироплазмы из пораженной самки клеша могут проникать в ее яйца и затем в личинки, а те в свою очередь переносить паразитов млекопитающим.

ТИП МИКСОСПОРИДИИ (*Mixotrichia*)

Известно более 870 видов миксоспоридий, в основном паразитов рыб и малоштамковых червей. В конце своего жизненного цикла микроспоридии образуют споры. Но в отличие от споровиков споры микроспоридий являются многоклеточными образованиями с полярными капсулами, в каждой из которых находится спиральная полярная нить. Кроме того, жизненный цикл миксоспоридий включает развитие паразита от одноядерной фазы к многоядерной. Многоядерная фаза завершается формированием множества многоклеточных спор с двуядерным амебным зародышем. Взрослым паразитам свойствен ядерный дуализм.

Миксоспоридий делят на два класса: класс Собственно миксоспоридии (*Myxosporea*) и класс Актиноспоридии (*Actinosporea*). Актиноспоридии паразитируют в малошетинковых червях и характеризуются некоторыми особенностями строения спор.

Миксоспоридии паразитируют преимущественно в коже рыб (тканевые паразиты); в результате у рыб образуются желваки-опухоли. В этих опухолях находятся взрослые многоядерные плазмодии миксоспоридий. Размеры паразитов колеблются от микрометров до двух сантиметров.

Ядра плазмодиев делятся на вегетативные и генеративные (ядерный дуализм). Вегетативные ядра регулируют обменные процессы в организме паразита, генеративные ядра участвуют в образовании спор, формирование которых происходит внутри плазмодия. Вокруг каждого генеративного ядра образуется обособленная генеративная клетка. В этой клетке ядро неоднократно делится и образуется панспоробласть. Внутри панспоробласта формируются две споры.

Споры миксоспоридий разнообразны по форме, но всегда являются многоклеточными образованиями с полярными капсулами. Так, спора миксоспоридия *Myxobolus* (возбудителя шишечной болезни у рыбы-усача) формируется из шести клеток, что подтверждается наличием шести ядер (рис. 24). Две клетки образуют две створки споры, две другие — капсулы с полярной нитью и две оставшиеся — амебоидный зародыш.

Из больной рыбы споры попадают в воду, где заглатываются другой рыбой. В кишечнике рыбы полярные капсулы выбрасываются и нити вонзаются в стенку кишки. Створки споры раскрываются и амебоидный двуядерный зародыш проникает через эпителий кишки в кровь. По кровяному руслу зародыши попадают под кожу или в другие ткани и органы рыбы. Сначала два ядра в зародыше сливаются и зародыш ста-

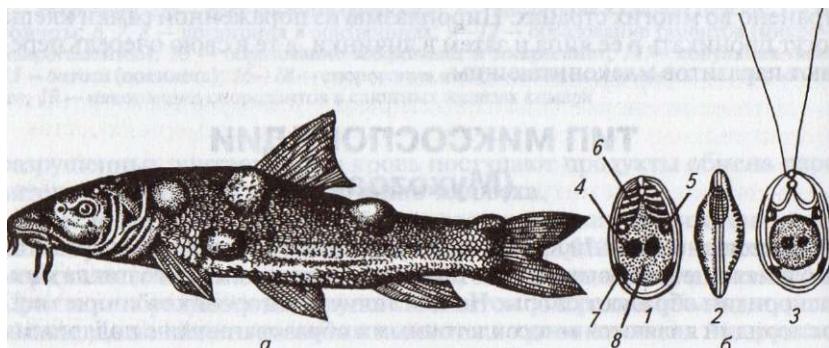
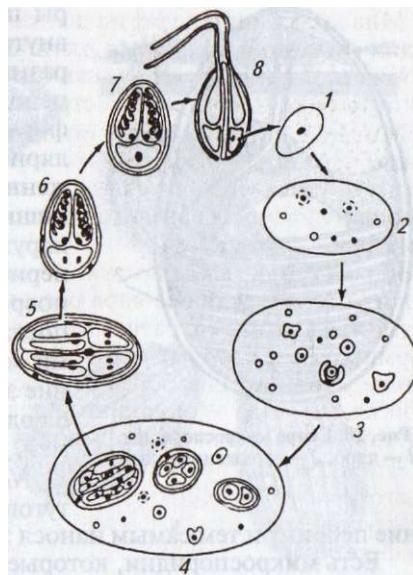


Рис. 24. Слизистые споровики:
а — рыба, пораженная споровиками; б — споры; 1 — спора с неразряженной капсулой;
2 — вид споры сбоку; 3 — спора с разряженной капсулой; 4 — ядро амебоидного зародыша;
5 — ядра клеток — образовательниц полярных капсул; 6 — стрекательная нить;
7 — ядра створок споры; 8 — амебоидный зародыш

I. < с. Лишенный цикл миксоспоридиев
Мухогод:

1 — зародыш с диплоидным ядром; 2 — образование многоядерного мицелия с вегетативными и генеративными ядрами; 3 — формирование спор; 4 — формирование поры с двуядерным амебоидным питомцем; 5 — изгнание спор; 6 — сформированная спора; 7 — образование диплоидного зародыша; 8 — выстrelивание из культических нитей из споры при прорастании.



пищци и я диплоидным. Затем ядрышко и инвагинация и зародыш превращается в многоядерный плазмодий с ядерным дуализмом. После деления в плазмодии образуются нормы (рис. 25).

Миксоспоридии вызывают миеионую гибель многих рыб, при уплотненных поселениях и прудовых хозяйствах. Этот ущерб наносит форели миксоспоридия *Myxosoma cerebralis*, циркляция которой скелет рыбы. У заболевших мальков искривляется позвоночник и нарушается координация движений.

ТИП МИКРОСПОРИДИИ (Microspore)

Существует около 800 видов микроспоридий, которые являются внутренними паразитами насекомых и других беспозвоночных животных; очень немногие виды паразитируют у позвоночных животных.

Микроспоридии из самых мелких простейших организмов, размеры их составляют 4-6 мкм.

Жизненный цикл микроспоридий также заканчивается образованием спор, но эти споры имеют иное строение, чем у споровиков и мицелий. У микроспоридий спора является одноклеточным оболоченным яйцом, в котором имеется одно-два ядра и одна ввернутая полярная нить (рис. 26). У микроспоридий отсутствует половой процесс. Они размножаются бесполым путем, образуя цепочки клеток внутри паразитной клетки хозяина.

Микроспоридия *Nosema apis* наносит серьезный ущерб пчеловодству и пчелам хозяйствам. Пчелы заражаются, заглатывая с кормом споры плазмодии. В кишечнике пчелы споры разбухают и выстреливают свои покрытые нити, которые вонзаются в стенку кишечника насекомого. Из спор

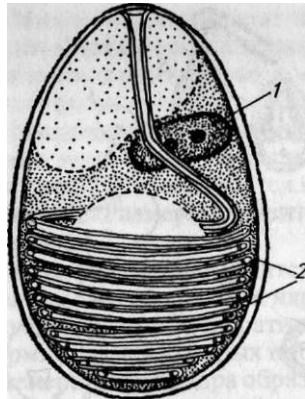


Рис. 26. Спора микроспоридии:
1 — ядро; 2 — заряженная нить

ры по каналу нити зародыш проникает внутрь кишечной клетки. В ней паразит размножается бесполым путем, образуя цепочки клеток. Из этих клеток затем формируются одноклеточные споры с полярной нитью. После разрушения пораженной клетки споры попадают в просвет кишки и с каловыми массами выводятся наружу. Больные нозематозом пчелы и период зимовки пачкают соты калом, одновременно загрязняя их спорами. При поедании пчелами загрязненного спорами меда происходит повторное заражение этих насекомых. При нозематозе наблюдают большой отход пчел и гибель семей.

Nosema bombycis патогенна для гусениц листового шелкопряда, вызывая заболевание пебрину и тем самым нанося значительный ущерб шелководству

Есть микроспоридии, которые, являясь паразитами вредных насекомых, используются в биологической борьбе с вредителями.

ТИП ИНФУЗОРИИ, ИЛИ РЕСНИЧНЫЕ (*Ciliophora*)

Инфузории отличаются наиболее сложной организацией среди одноклеточных животных. Их тело покрыто пелликулой, которая позволяет им иметь относительно постоянную форму. Под пелликулой расположена эктоплазма, в которой находятся многие органеллы, в том числе базальные тельца ресничек (рис. 27); сократительные волоконца — мионемы; защитные органеллы — трихоцисты. При раздражении инфузории выбрасывают из трихоцист множество упругих нитей, которые поражают врага, парализуя его.

Известно более 7,5 тыс. видов инфузорий, населяющих моря и пресные водоемы и входящих в состав планктона, бентоса и детрита; некоторые виды обитают в почве. Среди инфузорий много хищных и паразитических форм, в кишечнике жвачных животных обитают симбиотические формы. Органеллами движения инфузорий служат многочисленные реснички. При этом подавляющее большинство этих животных обладает ресничками в течение всей жизни. Реснички инфузорий по своему строению сходны со жгутиками. Ресничный аппарат весьма разнообразен. Особенно сложный ресничный аппарат расположен около рта.

Вторым отличительным признаком представителей этого типа является присутствие в их теле двух ядер (ядерный дуализм): крупного вегетативного ядра (макронуклеуса) и значительно более мелкого генеративного ядра (микронуклеуса).

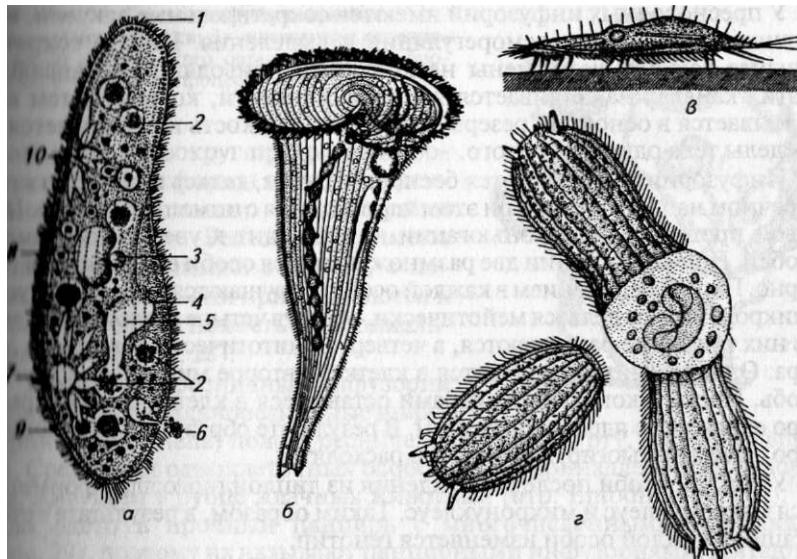


Рис. 27. Инфузории:

a — южкалька; **б** — трубач; **в** — брюхоресничная инфузория, ползающая с помощью утолщенных щетинок; **г** — хищные инфузории колепсы, напавшие на другую инфузорию; **1** — нградный конец; **2** — пищеварительные вакуоли; **3** — микронуклеус; **4** — реснички; **5** — кусочная глотка; **6** — удаленная через порошицу непереваренная пища; **7** — сократительная вакуоль; **8** — макронуклеус; **9** — трихоцисты; **10** — приводящие каналы сократительной вакуоли

Многие инфузории обладают сложной системой пищеварения. Расположенный в углублении тела (перистоме) рот, или цитостом, окружён длинными ресничками, с помощью которых пищевые частицы и погружаются в него. Часто рот ведет в глотку, погруженную в эндоплазму. В эндоплазме пищевые частицы окружены пузирьками, содержащими пищеварительные ферменты, в результате образуются пищеварительные вакуоли. В пищеварительных вакуолях создается кислая среда. На последующих этапах переваривания пищи среда становится нейтральной, что аналогично процессам пищеварения у высших животных. Непереваренные остатки пищи выбрасываются из тела инфузории в определенном месте через порошицу. Есть виды хищных инфузорий, вооруженных ротовым хоботком, с помощью которого они проникают покровы своей жертвы.

Большинство инфузорий питаются бактериями, немногие поедают одноклеточные водоросли, и среди них встречаются даже монофиты. Хищные инфузории порой охотятся на жертв, которые по размерам больше хищниц; это, например, инфузория-туфелька. Жертву они поражают хоботком и высасывают ее содержимое. Свободноживущие инфузории являются важным звеном в пищевых цепях экосистем.

У пресноводных инфузорий имеются сократительные вакуоли, выполняющие функции осморегуляции и выделения. Иногда сократительные вакуоли усложнены несколькими приводящими каналами. В этих каналах накапливается избыток жидкости, которая затем выбрасывается в основной резервуар; из него жидкость выталкивается за пределы тела одноклеточного.

Инфузории размножаются бесполым путем, делясь надвое, но в перечном направлении. При этом ядро делится с помощью митоза. Половой процесс в виде конъюгации не приводит к увеличению числа особей. При конъюгации две размножающиеся особи соединяются попарно. Перед соединением в каждой особи разрушаются макронуклеусы, а микронуклеусы делятся мейотически, образуя четыре гаплоидных ядра. Из них три также разрушаются, а четвертое митотически делится на два ядра. Одно из этих ядер остается в клетке, а второе мигрирует в другую особь. После такого обмена ядрами оставшееся в клетке стационарное ядро сливаются с ядром-мигрантом. В результате образуется диплоидное ядро. Затем конъюгирующие особи расходятся.

У каждой особи после расхождения из диплоидного ядра формируются макронуклеус и микронуклеус. Таким образом, в результате конъюгации в каждой особи изменяется генотип.

При классификации инфузорий в качестве диагностических признаков используют особенности строения ротового аппарата или структуру ресничного аппарата. Последний подход преобладает. Инфузорий делят на два класса: класс Ресничные инфузории (*Ciliata*) и класс Сосущие инфузории (*Suctoria*).

Класс *Ресничные инфузории* наиболее многочисленный. Представители этого класса покрыты ресничками на протяжении всех стадий жизненного цикла. Среди представителей подкласса Равноресничные инфузории (*Holotricha*), характеризующихся равномерным расположением на теле ресничек равной длины, много свободноживущих (например, инфузория-туфелька, *Paramecium caudatum*), хищных, питающихся своими собратьями, и паразитических форм. Среди последних следует отметить инфузорию балантидий (*Balantidium coli*), которая встречается в кишечнике свиней и человека. Эта инфузория питается в основном содержимым кишечника, но может разрушать слизистую кишечника, вызывая заболевание — балантидиоз. Заражение происходит при потреблении загрязненного цистами балантидия пищи и воды.

В природных водоемах и в прудовых хозяйствах, занимающих разведением рыб, большой вред наносят паразитические инфузории. Например, равноресничная инфузории *Ichthyophthirius* внедряется в кожу рыб и начинает питаться клетками хозяина. В результате на теле рыбы образуются многочисленные язвочки. Заболевание может привести к гибели рыб, особенно молоди карпа. На жабрах и коже часто паразитируют инфузории из рода *Trichodina*, причиняя молоди рыб существенный вред.

У большинства кругоресничных инфузорий (подкласс Кругоресничные инфузории — *Peritricha*) реснички располагаются левоспир-

Рис. 28. Сидячие инфузории сувойки:
1, J — деление сувойки; 3 — плавающая, отделившаяся от материнского организма клетка-бродильщик; 4 — половой процесс

ришно только вокруг предротовой воронки. Многие формы ведут прикрепленный образ жизни. Так, Сувойки (*Vorticella*, отр. Peritrichida) имеют винный сократимый стебелек, с помощью которого они прикрепляются к субстрату. Среди них есть и колониальные формы (рис. 28).

У спиральноресничных инфузорий (подкласс Спиральноресничные инфузории — Spirotricha) полоса ресничек, ведущих ко рту, закручена вправо. Среди этих одноклеточных особое место принадлежит инфузориям, живущим в рубце жвачных животных (отр. Entodiniomorpha). Их тело одето в прочный панцирь с многочисленными отростками (рис. 29), поэтому их называют панцирными инфузориями. Питаются они бактериями рубца и способствуют расщеплению клетчатки коры, взаимодействуя сложным образом с целлюлозорасщепляющими микробами. Эти полезные симбионты не только участвуют в переваривании пищи, но и сами служат источником питания для жвачных «и потных».

Нестейшие из класса Сосущие инфузории (*Suctoria*) лишены ресничек и в большей части жизненного цикла, и лишь на ранних этапах размножения дочерняя клетка — бродяжка — имеет реснички. У них нет рта и околосротовой воронки. Эти инфузории с шаровидным телом, на котором индивидуально расположены щупальца (рис. 30). Щупальца служат для прикрепления к субстрату и, кроме того, наполняются ловчим аппаратом. Поймав мелких инфузорий с помощью липкого секрета, суктории как бы перекачивают содержимое жертвы в свое тело.

При бесполом размножении от материальной клетки суктории отпочковываются дочерняя особь — бродяжка,

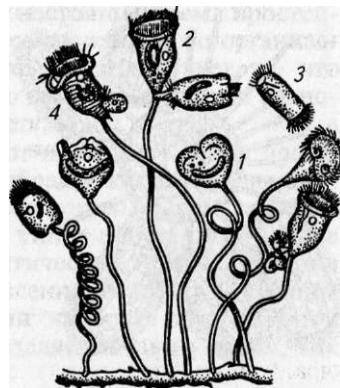


Рис. 24. Инфузория из желудка жвачного млекопитающего:
1 — перепонки, загоняющие пищу в рот; 2 — клеймо плотка; 3 — реснички; 4 — сократительные никуоли



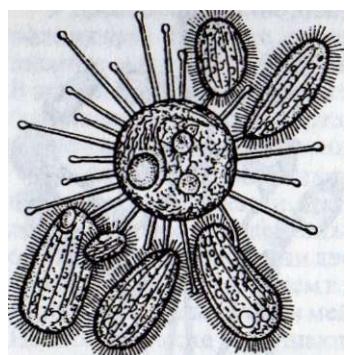


Рис. 30. Сосущая инфузория *Sphaerophrya*, высасывающая щупальцами содержимое нескольких ресничных инфузорий

покрытая ресничками. В последующем она теряет ресничный аппарат и превращается в сукторио с щупальцами. Следует отметить важную роль некоторых мелких видов инфузорий, жгутиконосцев и амеб в жизни почвы. Бактерии, в том числе азотфикссирующие, служат основной пищей для простейших. Однако они не только поедают бактерий, но и способствуют их размножению, выделяя в почву вещества, стимулирующие этот процесс. Простейшие способствуют повышению плодородия почвы, активно участвуют в процессах биологической очистки водоемов.

Имеется опыт искусственного разведения паразитических простейших для борьбы с вредными насекомыми.

ФИЛОГЕНИЯ И ЭКОЛОГИЧЕСКАЯ РАДИАЦИЯ ПРОСТЕЙШИХ

Мир живых существ делят на два надцарства: Безъядерные, или Прокариоты (Prokaryota), и Ядерные, или Эукариоты (Eucaryota).

Клетки прокариот в отличие от клеток эукариот не имеют оформленного ядра. Эукариот обычно делят на три царства: Растения (Vegetabilia, или Plantae), Грибы (Mycetalia, или Fungi) и Животные (Animalia, или Zoa). Большинство растительных организмов — автотрофы, самостоятельно синтезирующие органические вещества в процессе фотосинтеза. Грибы, хотя и относятся к гетеротрофным организмам, но питаются растворенными органическими веществами. Животные являются гетеротрофами, существующими за счет потребления других организмов или их остатков. Но различия по типам питания между этими царствами относительны, так как имеется множество переходных форм, особенно среди низших представителей.

Простейших (Protozoa) относят к примитивным одноклеточным эукариотам. Признано, что эукариоты берут начало от прокариот. Косвенным свидетельством их единства служит сходство процессов синтеза белка в клетке. Безъядерные организмы были одними из первых на нашей планете, часть из них способна существовать даже в бескислородной среде.

Среди эукариотических животных клеточный уровень организации рассматривают как более примитивный. Это позволяет полагать, что простейшие на Земле появились первыми и послужили началом более слож-

УttR форм — многоклеточным животным. В настоящее время простейшие существуют во взаимодействии с более сложными по организацией многоклеточными организмами. Этому способствует то, что Бритайшие в процессе эволюции отлично приспособились к различным условиям жизни на нашей планете,

и Полагают, что эукариоты произошли от прокариот путем постепенного возникновения органелл из мембранных клетки прокариот. В эволюции эукариот, по-видимому, большую роль сыграл симбиоз различных прокариот. Из известных семи типов простейших четыре являются исключительно паразитическими группами, которые значительно позднее, после появления их хозяев — высших проклеточных. Поэтому при выяснении наиболее примитивной группы среди простейших все внимание уделяется таким типам, как Саркомастигофоры и Инфузории.

Инфузории обычно относят к самым высокоорганизованным одноклеточным животным: есть представители с чертами многоклеточности. Таким образом, только саркомастигофоры несут первичные приемки, имеющие сходство с предками всех простейших. Но саркомастигофоры чрезвычайно неоднородны, что подтверждается особенностями (группами) подтипов Саркодовые и Жгутиконосцы.

Кольшинство исследователей придерживаются мнения Пашера (1914) о том, что жгутиковых следует относить к более древней группе. У них много общего с одноклеточными растениями, разнообразнее типов питания, а их органеллы движения имеются даже у части прокариот. Можно предположить, что разнообразные способы питания у жгутиконосцев могли стать основой для последующего их разделения на автотрофов и гетеротрофов. Следует отметить, что жгутики есть у гамет иллюзии и Metazoa. Упрощенность (отсутствие жгутиков и пелликулы) саркомастигофоры можно отнести к вторичным явлениям в связи с переходом к особому типу питания — фагоцитозу. Наличие же жгутика у гамет части саркомастигофоры может лишь свидетельствовать об их происхождении от жгутиконосцев. Кроме того, в последнее время было показано, что у некоторых амеб с ростом происходит редукция жгутиков у взрослых форм. Описано немало форм саркомастигофор, обладающих новидоподиями и жгутиками одновременно.

Таким образом, можно предположить, что предками современных групп были древние Саркомастигофоры с разными способами питания и имеющие примитивные жгутики.

Бесспорно, что Apicomplexa и Ciliophora имеют родство со жгутиконосцами. Споровики могли упроститься в связи с переходом к паразитическому образу жизни, но одновременно с этим у них усложнился жизненный цикл, который включает стадию гамет, снабженных жгутиками. Мухозоа и Microspora могли произойти от древних саркомастигофор, поскольку их развитие начинается с амебоидного иродыша, а гаметы со жгутиками отсутствуют. Возможны варианты пинтономности их эволюционного развития. На основе изложенного выше материала представляется возможной филогенетическая



Рис. 31. Филогения Protozoa

схема Protozoa, представленная на рис. 31.

На базе современных представлений можно обозначить основные пути экологической радиации одноклеточных животных (рис. 32). Центральной группой могли быть многочисленные и разнообразные представители Саркомастигофор, но с преобладанием жгутиковых форм и господством водных форм. В последующем саркодовые утратили жгутики и осуществили переход к ползающему образу жизни и питанию путем фагоцитоза и пиноцитоза. Бентосный образ жизни способствовал образованию защитных раковин разнообразной конструкции, таких как раковины корненожек, фораминифер. У части саркодовых

наружный скелет усложнился и стал более легким, в результате чего они смогли перейти к планктонному образу жизни (радиолярии, солнечники).

В связи с прогрессом клеточного строения возникают крупные паразитарные формы — инфузории. Они активно передвигаются, ведут разнообразный образ жизни. Среди них плавающие, ползающие, сидячие, скважники. Позже возникают симбиотические и затем паразитические формы инфузорий. Последние и поныне продолжают свое развитие, усложняя и совершенствуя отдельные этапы своего жизненного цикла.

Многим простейшим, в основном обитающим в пресных водоемах и паразитирующими в других организмах, свойственно образование цист и спор при наступлении неблагоприятных условий (высыхание и вымерзание водоемов). У морских представителей инцистирование является лишь исключением. В виде цист и спор простейшие могут переноситься ветром, птицами и другими животными на большие расстояния, благодаря чему пресноводные простейшие встречаются на земной поверхности повсюду, где имеются условия для их существования.

Пресноводные простейшие заселяют все виды водоемов, даже самые мелкие лужи или скопления воды, образующиеся в пазухах листьев растений. Множество их и в болотах, где в основном встречаются раковинные амебы. По составу фауны простейших относительно точно можно определить степень загрязнения водоема, так как в зависимости от этой степени водоемы заселяют разные представители простейших; их состав меняется по мере возрастания или снижения степени загрязнения воды. Простейшие живут даже в горячих источниках с темпера-

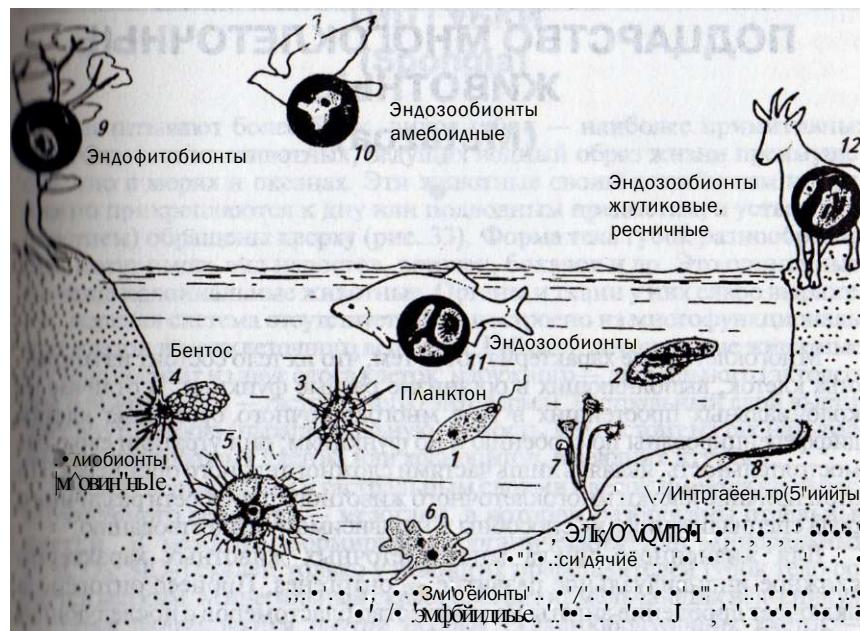


Рис. 32. Экологическая радиация простейших:

1—ушконосец; 2—инфузория; 3—радиолярия; 4—раковинная амеба; 5—фораминыф; 6—амеба; 7—суворка; 8—инфузория; 9—лептомонас; 10—кишечная амеба; II, I/—различные паразитические простейшие

И в приподнятых поди до +50 °C, находят их и в водоемах с высоким уровнем застенности (до 25 %).

Наиболее разнообразен состав простейших в морях и океанах, где они **широко распространены** в огромных количествах всю толщу воды. Вместе с одноклеточными водорослями простейшие служат кормом для других животных, в **числе которых** для рыб и китообразных. Раковинные саркодовые, отмирая, опускаются на дно, образуя мощнейшие донные отложения известняков.

ПОДЦАРСТВО МНОГОКЛЕТОЧНЫЕ ЖИВОТНЫЕ (Metazoa)

Многоклеточные характеризуются тем, что их тело состоит из множества клеток, выполняющих в организме разные функции. В отличие от колониальных простейших в теле многоклеточного организма клетки дифференцированы по строению и по функциям; они утратили свою самостоятельность, являясь лишь частями сложного единого организма. По этой причине клетки многоклеточного животного, приобретя различную роду специализацию, не способны к независимому существованию.

Для жизненного цикла многоклеточных животных характерно сложное индивидуальное развитие — онтогенез. Процесс онтогенеза включает дробление зиготы на множество бластомеров с последующей дифференцировкой их на зародышевые листки и зародышевые листки и зачатки органов, развитие, рост и образование взрослого организма. Увеличение размером тела многоклеточных по отношению к их поверхности способствовало усложнению процессов обмена веществ, что, в свою очередь, обеспечило многоклеточным животным устойчивость жизненных процессов и способствовало продлению их жизни.

Большинство ученых считают, что многоклеточные произошли от Protozoa. В пределах Protozoa прослеживается тенденция перехода к многоклеточности. В отдельных случаях у Protozoa наблюдается даже многоклеточность отдельных фаз развития (Миксоспоридии). Структурные компоненты клеток Protozoa идентичны таковым клеток Metazoa. Важное значение в решении вопроса о происхождении многоклеточных многие исследователи приписывают колониальным простейшим, например *Volvox*, у которого имеются клетки двух типов — соматические и половые.

Еще в 1874 г Э. Геккель утверждал, что предком многоклеточных была шаровидная колония какого-то простейшего, и что в процессе эволюции (филогенеза) за счет втячивания одной половины шара могла возникнуть первичная кишечная полость и первичный рот. Такой уже двухслойный организм плавал с помощью жгутиков, размножался половым путем и впоследствии стал предком многоклеточных.

Существует множество других теорий, но большая часть их сходится в одном: отдаленными предками многоклеточных животных были колониальные простейшие организмы.

ТИП ГУБКИ (*Spongia*)

Нашитмпают более 5 тыс. видов губок — наиболее примитивных Milt мок неточных животных, ведущих водный образ жизни преимущественно и морях и океанах. Эти животные своим основанием неподвижно прикрепляются к дну или подводным предметам, а устьем (отверстием*) И нем) обращены кверху (рис. 33). Форма тела губок разнообразна: они могут иметь вид наростов, веточек, бокалов и др. Это одиночные, Но чище колониальные животные. Органы и ткани у них слабо выражены* • пгтк и межклеточного вещества. Губки двухслойные животные. И*ично состоит из двух слоев клеток: наружного дермального (эктодермы) и внутреннего — гастрального (энтодерма). Гастральный слой выстилает иногреннюю (парагастральную) полость. Он состоит из так называемых ворсинок п шчковых клеток, или хоаноцитов, имеющих жгутики.

Между дермальным и гастральным слоями клеток имеется слой бесструктурного упрочняющего вещества — мезоглея, в котором разбросаны отдельные минеральные частицы. В мезогле формируется органический или минеральный ((SiO_2) скелет; только у немногих представителей губок тело освобождено от скелета. Минеральный скелет состоит из мельчайших игл-спикул, которые формируются внутри особых скелетообразующих клеток — хондроцитов.

Многочисленные, расположенные в мезогле. Роговой (спонгиновый) скелет имеет химическому составу близок к шелку.

Внешняя поверхность тела губок пронизана множеством пор, через которые вода поступает в систему каналов и камер. Движение воды обеспечивается жгутиками хоаноцитов. Из каналов и камер вода попадает в центральную или гастральную полость, откуда выводится наружу через устье.

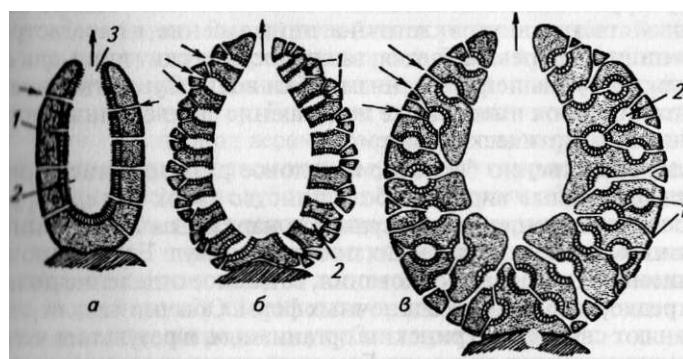


Рис. II Типы строения губок с различной сложностью системы каналов и расположением жгутиковых камер:

Л — шишкообразная губка; б — сикон; в — лейкон; 1 — поры; 2 — жгутиковые камеры; 3 — устье (стрелки указывают направление тока воды в теле губки)

Специальные клетки — пороциты — способны к сокращению и могут открывать и закрывать поры.

Среди клеток, находящихся в мезоглее, выделяются подвижные клетки — амебоциты. По своим функциям они универсальны: выполняют транспортную функцию, выделяют вещество мезоглея, из них образуются половые клетки и клетки всех других типов, они участвуют в бесполом размножении, дают начало клеткам, образующим скелет жи вотного. Они переносят пищевые частицы от ханоцитов к другим клеткам, удаляют экскреты, а в период размножения переносят спермин в мезоглее к яйцеклеткам. Такие разнообразные функции амебоцитов характеризуют губок как животных, находящихся на эволюционной лестнице ниже прочих многоклеточных.

Наиболее простой тип строения губок называют аскон (рис. 33), но он в основном характерен для одиночных форм и для молодых колониальных особей. Усложнение в период индивидуального развития приводит к возникновению более сложной формы — типу сикон. Дальнейшее усложнение строения тела губок (утолщается мезоглея, образуются карманы и камеры, покрытые слоем воротничковых клеток — ханоцитов) ведет к самому сложному типу — лейкон. Таким образом, у губок типа лейкон и частично сикон парагастральная полость (в отличие от типа аскон) оказывается выстиланной клетками эктодермы. Ими же выстиланы приводящие и отводящие (у лейкон) каналы, которые являются впаяньями эктодермы. Энтодермой выстиланы лишь жгутиковые камеры, число которых у губок лейконового типа огромно — до нескольких миллионов.

Движение воды по каналам тела губок обеспечивает их организм кислородом и способствует удалению из тела продуктов обмена. С водой в тело губок попадают пищевые частицы (мелкие водные животные и растительные организмы, гниющие остатки), которые захватываются псевдоподиями ханоцитов и перевариваются в их цитоплазме. Часть захваченной пищи передается амебоцитам, и те ее переваривают и транспортируют питательные вещества в мезоглею. Таким образом, губкам свойственно внутриклеточное пищеварение, в парагастральной полости пища не переваривается; эта полость служит лишь для сбора и эвакуации поступившей в организм губки воды. Существенное значение в питании губок имеет также поглощение растворенных органических веществ осмотическим путем.

Губкам свойственно бесполое и половое размножение. Среди них есть раздельнополые виды, но большинство губок — гермафродиты. Бесполое размножение осуществляется наружным почкованием или образованием особых внутренних почек — геммул. Если бесполое размножение носит характер почкования, то полное отделение почки происходит редко, в основном у одиночных форм. Обычно же дочерние особи сохраняют связь с материнским организмом, в результате чего образуются и разрастаются колонии. Границы между отдельными особями могут исчезать, и тогда вся колония сливаются в общую массу (рис. 34).

Геммулы в виде групп клеток, окруженных оболочками и содержащих запас питательных веществ, образуются в мезоглее. Эти образова-

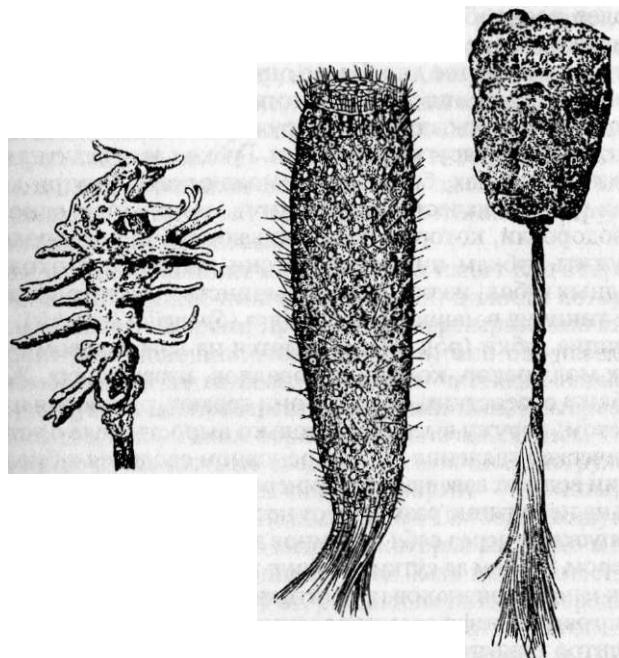


Рис. 34. Различные виды губок:
а — бадяга; б — стеклянные губки

ним представляют собой покоящиеся зимние почки. Так, например, и **р (и овод ная** губка бадяга летом размножается обычным почкованием и **половым** путем. Но к осени в мезоглее бадяги амебоциты образуют **Шаровидные** геммулы. Зимой тело бадяги умирает и распадается. Геммулы **остаются** на дне и перезимовывают. Весной содержащаяся внутри геммул клеточная масса выползает, прикрепляется к субстрату и **ршииняется** в новую губку. Геммулы выполняют также функцию **распрггин**, так как в период весеннего разлива они переносятся течением. **Мри** пересыхании водоемов геммулы могут переноситься ветром.

Ноловое размножение губок происходит путем образования в мезогле амебоцитов яйцеклеток и спермиев. Спермин выносятся в по-
ш п. канальцев и с водой парагастральной полости — во внешнюю ир#лу через устье (оскулюм). С током воды спермин попадают в тело **Уок и, и меющей** зрелые яйцеклетки, проникают в мезоглею и сливаются с ними, т.е. оплодотворение у губок перекрестное.

В материнском организме из зиготы развивается личинка, покрытая ресничками; личинка выходит наружу, активно плавает, перемещаясь течением воды на значительные расстояния, затем опускается на субстрат и прикрепляется к нему и превращается в губку.

Наиболее разнообразны и многочисленны губки тропических и субтропических морей. Встречаются губки на небольших глубинах, предпочитая каменистое дно. Часто они сожительствуют с другими организмами, вступая с ними в симбиотические отношения различного типа. В колониях губок можно обнаружить кольчатых червей, ракообразных, иглокожих и других животных. Губки часто селятся на подвижных животных (крабах, брюхоногих моллюсках). Внутри клеток пресноводных губок в качестве симбионтов часто живут одноклеточные зеленые водоросли, которые обеспечивают губку кислородом и могут также служить губкам пищей. В России встречается около 20 видов пресноводных губок, из которых большинство обитает в озере Байкал. Наиболее типична в наших реках бадяга (*Spongilla lacustris*).

Сверлящие губки (род *Cliona*) селятся на известковом субстрате — раковинах моллюсков, колониях кораллов, известняках. Живут сверлящие губки в отверстиях, которые они делают, растворяя известок особым секретом; наружу выступают только выросты тела с устьями.

Практическое значение губок в основном сводится к биологической фильтрации воды от взвешенных минеральных и органических веществ. Несмотря на небольшие размеры (от нескольких миллиметров до 1,5 м), губки пропускают через себя огромное количество воды: одна губка бадяга размером 5—7 см за сутки профильтровывает около 3 л воды.

У губок много признаков примитивности организации: у них отсутствуют настоящие дифференцированные ткани и органы, для клеточных элементов характерна высокая пластичность и т. п. Губки способны к регенерации: при удалении отдельных участков тела происходит их восстановление. Если измельченную губку просеять через сито, то образовавшаяся масса из отдельных клеток и их групп способна к восстановлению целого организма. Клетки кашицы активно двигаются и собираются вместе, в последующем из этого скопления клеток формируется маленькая губка. Такой процесс формирования организма из скопления клеток называют *соматическим эмбриогенезом*.

Губки — древние организмы. Отделение губок от ствола многоклеточных произошло очень давно. Существует мнение, что губки могли произойти от колониальных воротничковых жгутиконосцев независимо от прочих многоклеточных. Не менее обоснована гипотеза, что многоклеточные произошли общим стволом, от которого одними из первых отделились губки. Вторая гипотеза представляется более обоснованной, ибо личинки губок сходны с личинками планулами кишечнополостных.

ТИП КИШЕЧНОПОЛОСТНЫЕ (*Coelenterata*)

Общая характеристика. Тип объединяет более 10 тыс. видов примитивных многоклеточных животных, ведущих исключительно водный образ жизни и обитающих в основном в морях. Часть из них ведут свободноплавающий образ жизни, другие — сидячий и прикрепленный к дну.

Кишечнополостным свойственна радиальная симметрия, что связано с их образом жизни. У сидячих форм один полюс тела обычно служит для прикрепления к субстрату, на другом имеется рот. Многие органы получают одинаковое развитие, что приводит к радиальной симметрии. Кишечнополостные — двухслойные животные: у них формируются только два зародышевых листка — эктодерма и энтодерма. Между этими листками находится первичная полость тела, заполненная мезоглеей, которая у одних представителей имеет вид пластинки, а у других — это большая масса студенистого вещества.

В простом случае тело кишечнополостных имеет вид открытого на одном конце мешка, в кишечной (гастральной) полости которого, выстланной клетками энтодермы, происходит переваривание пищи. Отверстие служит для кишечнополостных ртом, оно окружено венцом щупалец, помогающих захватывать пищевые частицы. Анальное отверстие отсутствует, а непереваренные остатки пищи выбрасываются через ротовое отверстие. Таким образом, можно заключить, что просто устроенные кишечнополостные сводятся к типичной гаструле. К этой схеме строения наиболее близки сидячие формы — полипы, широко распространенные среди кишечнополостных. Свободноживущие формы имеют уплощенное тело; это медузы, которые активно и пассивно с: течениями передвигаются в водной среде. Тело медуз имеет вид прозрачного студенистого зонтика. Рот, расположенный посередине нижней стороны купола и окруженный предротовыми лопастями, ведет в кишечную полость, от которой отходят радиальные каналы. Океанические медузы достигают двух метров в диаметре.

Деление кишечнополостных на полипы и медузы чисто морфологическое, поскольку иногда один и тот же вид кишечнополостных на разных стадиях жизненного цикла может иметь строение то медузы, то полипа. Медузы — обычно одиночные свободноживущие животные, а полипы в своем большинстве — колониальные формы. Начиная жизнь как одиночный организм, полип путем неполного почкования образует колонии, насчитывающие тысячи особей.

Для кишечнополостных характерно наличие стрекательных клеток, служащих для добывания пищи и защиты.

Размножаются кишечнополостные бесполым (почкованием) и половым путем. У многих форм при этом наблюдается чередование поколений: бесполое поколение полипов сменяется половым поколением медуз.

Строение и жизненные отправления. Покровы кишечнополостных образованы однослойным эпителием эктодермального происхождения. В эпителии располагаются узкоспециализированные клеточные элементы. Это эпителиально-мышечные клетки, содержащие миофибриллы, которые обеспечивают укорочение тела полипа. По всей поверхности тела и особенно густо на щупальцах и вокруг рта разбросаны чувствительные клетки, выполняющие функции рецепторов, мое принимающих сигналы из внешней среды. Характерны в покровах кишечнополостных стрекательные клетки, в основном расположены

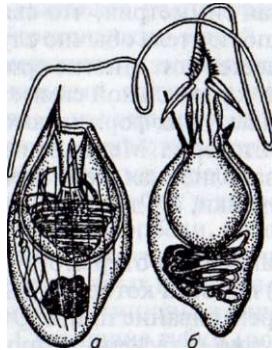


Рис. 35. Стрекательные клетки гидры *Hydra olidactis*:
а — в покоящемся состоянии;
б — с выброшенной нитью

ные на щупальцах (рис. 35). Внутри каждой такой клетки имеется капсула со спирально закрученной полой нитью. Если прикоснуться к чувствительному волоску клетки, стрекательная нить выворачивается и выбрасывается наружу. Вооруженная шипиками нить вонзается в тело жертвы и удерживается в ранке, вводя при этом в нее ядовитый секрет, парализующий мелкую добычу. У крупных животных этот секрет вызывает ожоги. Стрекательные клетки — одноразовое оружие. На месте сработавших клеток образуются новые, так как в покровах кишечнополостных имеются особые клетки, которые могут превращаться в стрекательные, половые, чувствительные и другие.

Нервная система у полипов представлена нервным сплетением диффузного

типа, образованным звездчатыми нервными клетками, соединенными своими отростками. Нервное сплетение лежит под покровным эпителием. У свободноживущих медуз нервная система сложнее: это нервное кольцо, располагающееся по краю купола и скопления нервных клеток вокруг глазков и статоцистов.

Органы чувств примитивны и лучше развиты у медуз (статоцисты и глазки). Чувствительные клетки имеются в покровах тела, особенно на щупальцах и вокруг ротового отверстия.

Мускулатура. У полипов форма тела меняется в результате действия эпителиально-мышечных клеток, имеющих миофибриллы. У медуз движение обеспечивается специальными мышечными волокнами, залагающими в мезоглее по краям купола. У коралловых полипов продольные и поперечные мышечные волокна находятся в перегородках кишечной полости.

Органы пищеварения. У гидр и близких к ним форм ротовое отверстие открывается непосредственно в кишечную (гастральную) полость. У большинства же видов рот ведет в эктодермальную глотку и затем в кишечник. У коралловых полипов для увеличения всасывающей поверхности в кишечную полость вдаются радиально расположенные продольные перегородки. У медуз от кишечной полости внутрь купола отходят радиальные каналы, впадающие в кольцевой канал. Кишечная полость у медуз продолжается и в полости щупалец.

Кишечная полость у кишечнополостных выстилана однослойным энтодермальным эпителием, клетки которого имеют жгутики, служащие для перемещения пищевых частиц. Есть особые железистые клетки. Некоторые клетки эпителия образуют псевдоподии, захватывающие частицы пищи. Одновременно с внутриклеточным пищеварением у кишечнополостных частично происходит и полостное переваривание

в кишечной полости с помощью пищеварительных ферментов, вырабатываемых железистыми клетками кишечного эпителия. У гидроидных полипов существуют две фазы переваривания пищи. Сначала они заглатывают крупный комок пищи или целое животное, которое начинает перевариваться в гастральной полости. Затем мелкие частицы полупереваренной пищи попадают внутрь эпителиально-мускульных пищеварительных клеток, где происходит внутриклеточное пищеварение. Непереваренные остатки выбрасываются через рот наружу.

Органы дыхания у кишечнополостных отсутствуют, а газообмен осуществляется через покровы тела.

Выделительная система. Продукты обмена (вода, диоксид углерода, мочевина, мочевая кислота, аммиак и др.) выделяются через эпителиальный слой эктодермы и энтодермы.

Размножение. Большинство кишечнополостных — раздельнополые животные, но есть и гермафродиты. У гидроидных половые продукты образуются в эктодерме, у остальных представителей их образование происходит в энтодерме. Оплодотворение у одних видов наружное (в воде), у других — внутреннее, в теле женских особей, куда проникают сперматии. Обычно развитие происходит со стадией личинки планулы, покрытой ресничками, позволяющими плануле плавать. У пресноводных гидр развитие прямое.

Тип Кишечнополостные делят на три класса: Гидроидные (*Hydrozoa*), Сцифоидные медузы (*Scyphozoa*) и Коралловые полипы (*Anthozoa*).

КЛАСС ГИДРОИДНЫЕ (*Hydrozoa*)

Низший класс кишечнополостных, состоящий примерно из 4 тыс. видов. Гидроидные представлены разнообразными одиночными и колониальными формами, населяющими преимущественно моря и океаны. Имеются и пресноводные представители. В отличие от сцифоидных медуз и коралловых полипов полипы и медузы, которые принадлежат к классу *Hydrozoa*, называются гидроидными. У гидроидных отсутствует глотка, стенки кишечной полости не имеют продольных перегородок. Половые продукты образуются в эктодерме.

Наиболее типичными для пресных вод являются различные виды гидр (*Hydra*), ведущие одиночный образ жизни полипа (рис. 36). Это небольшие животные высотой 1—2 см с расширенным основанием, на котором они удерживаются на субстрате. Ротовое отверстие окружено венчиком из 6—12 щупалец, а более широкое тело переходит в стебель. Мезоглэя имеет вид тонкой опорной пластинки, в которой разбросаны нервные, эпителиально-мускульные и промежуточные клетки. Из последних при необходимости формируются половые, стрекательные и другие клетки. Нервная система гидры имеет диффузный характер, хотя вокруг рта и на подошве находятся небольшие скопления нервных клеток. Эпителиально-мускульные клетки могут образовывать псевдоподии и поэтому способны к фагоцитозу.

Рис. 36. Пресноводная гидра *Hydra olidactis*:
1 — общий вид; 2 — продольный разрез; 3 — тело; 4 — подошва; 5 — щупальца; 6 — кишечная полость; 7 — энтоцерма; 8 — эктодерма; 9 — опорная пластинка — мезоглея; 10 — семенники; 11 — образование яйца

Обитают гидры в пресных водоемах со стоячей или малоподвижной водой. Гидры могут медленно передвигаться за счет скольжения по-дошвы по субстрату или «кувырканием» через головной конец. Питаются мелкими ракообразными, инфузориями, коловратками и другими планктонными животными, улавливая добычу щупальцами, вооруженными стрекательными клетками.

Размножаются гидроидные почкованием и половым путем. При мерно на середине тела гидры имеется пояс почкования. Дочерние организмы отпочковываются и начинают самостоятельную жизнь в течение всего лета. Осенью гидры размножаются половым путем. На поверхности тела появляются особые выпуклости: несколько семенников или один-два яичника, в каждом из которых образуется только одна яйцеклетка. Гидры раздельнополы, но есть и гермафродиты. В последнем случае семенники на теле гидры образуются выше яичников. Спермин выходит в воду и проникают в яйцеклетку другой особи. Перекрестное оплодотворение у гермафродитных форм достигается разным временем созревания спермиев и яйцеклеток. Сначала развитие зиготы проходит в яичнике, затем зародыш покрывается оболочками, падает на дно и зимует. В таком состоянии зародыш может переносить промерзание и высыхание водоема. Весной из перезимовавшего зародыша вырастает гидра. Таким образом, у пресноводных гидр развитие прямое.

Гидры способны к регенерации, даже из части тела восстанавливается весь организм.

Среди обитателей морских вод подавляющее большинство гидроидных являются колониальными формами со сложным жизненным циклом (рис. 37). Колонии образуются путем многократного неполного почкования. В результате получается комплекс особей, сидящих на общем стволе и его побочных ветвях. Поэтому колония обычно напоминает бурые нарости мха или кустик, на ветвях которого сидят отдельные особи колонии — гидранты, похожие по строению на гидру. Кишечные полости всех гидрантов сообщаются между собой, т. е. пища и колонии может распределяться по всей колонии, что обеспечивает ее выживание. Для устойчивости и прочности за счет выделений эктодермального эпителия полипы образуют органическую оболочку — теку, одевающую не только общий ствол, но и отдельных гидрантов.

Размножение гидроидных полипов включает чередование бесполого поколения, ведущего прикрепленный образ жизни, и полового поколения — свободноплавающих гидроидных медуз (гидромедуз). В самих гидрантах колонии половые железы не образуются. Периодически на веточках колонии гидроидных полипов образуются особые почки,

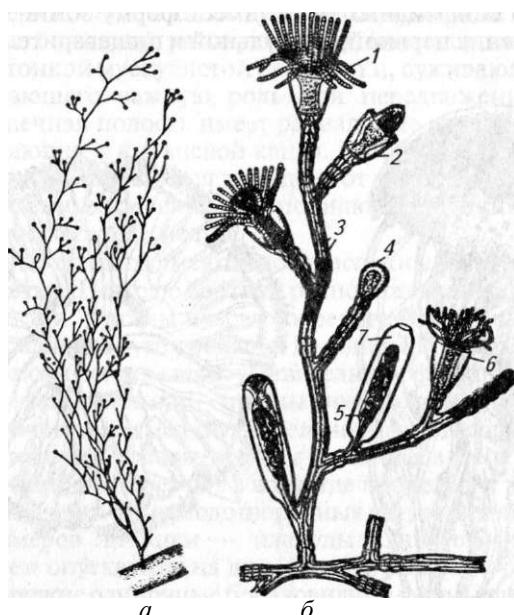


Рис. 37. Гидроид *Obelia*:
а — колония (слегка увеличено); б — отдельная веточка колонии (несколько схематизировано, часть особой колонии изображена в разрезе); 1 — гидрант в расправленном состоянии; 2 — сократившийся гидрант; 3 — тека; 4 — почка; 5 — бластостиль с развивающимися медузами; 6 — гидротека; 7 — гонотека (участок теки, одевающий бластостиль)

дающие начало половым особям — мелким гидроидным медузам. Эти медузы отрываются от материнской колонии и свободно плавают. Гидроидные медузы растут, и в них развиваются половые клетки. Медузы раздельнополы. Гидроидные медузы устроены значительно сложнее, чем гидроидные полипы; у медуз имеется нервное кольцо, статоцисты, глазки и т. п. Медузы ведут хищный образ жизни, захватывая и убивая щупальцами мелких животных, проглатывая и переваривая их в желудке. После созревания половые клетки выходят в воду и копулируют.

После копуляции гамет образуются личинки планулы, которые свободно плавают в воде с помощью многочисленных ресничек. Через некоторое время планулы опускаются на дно, прикрепляются к субстрату и превращаются в неподвижных полипов, которые дают начало новым колониям.

КЛАСС СЦИФОИДНЫЕ МЕДУЗЫ (Scyphozoa)

Класс, насчитывающий около 200 видов, представлен крупными и мелкими морскими медузами. Большая часть их жизненного цикла проходит в форме плавающих медуз (немногие формы ведут прикрепленный образ жизни); фаза полипа кратковременна или может отсутствовать. Тело сцифоидных медуз имеет форму зонта, купола и т. п. (рис. 38). Строение нервной, мускульной и пищеварительной систем у

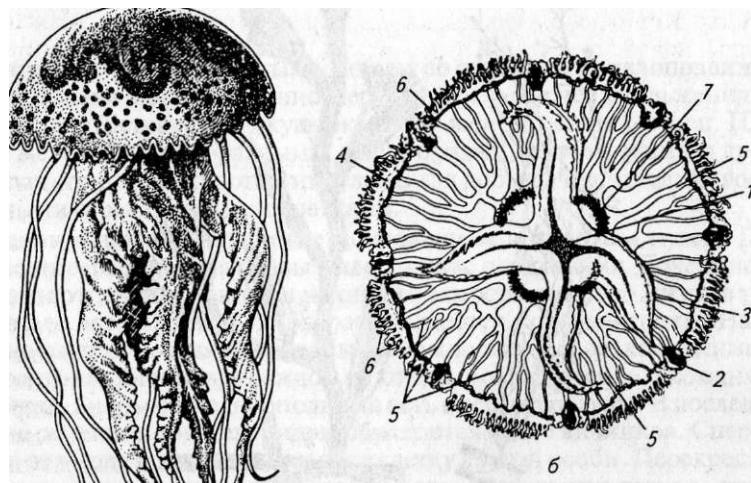


Рис. 38. Сцифоидные медузы:
а — медуза корнерот; б — схема строения аурелии; 7 — рот; 2 — ропалий; 3 — ротовые лопасти; 4 — кольцевой канал; 5 — радиальные каналы; 6 — щупальца; 7 — половые железы

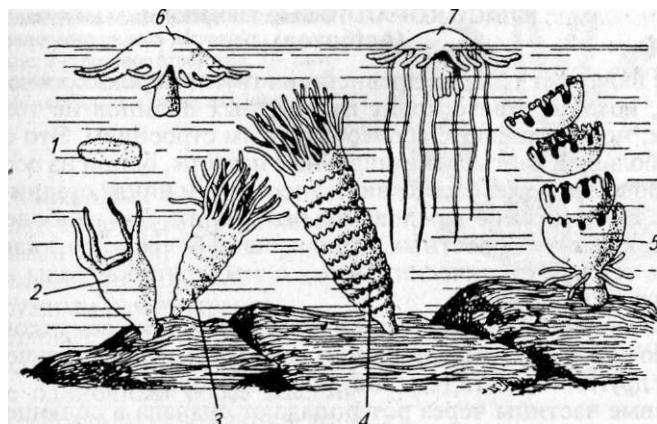


Рис. 39. Схема развития сцифоидной медузы аурелии (*Aurelia aurita*):
1 — личинка планула; 2 — полип сцифистома; 3,4 — стадии почкования сцифистомы; 5 — отделение от сцифистомы личинок эфир; 6 — молодая медуза-эфира; 7 — взрослая медуза

этих медуз более сложное. В мезоглее купола имеются мышечные волокна, обеспечивающие сжатие купола. Сцифоидные медузы отличаются не только большими размерами тела, но и отсутствием специального паруса (тонкой мускулистой перепонки, суживающей кромки колокола), играющего важную роль при передвижении гидроидных медузок. Кишечная полость имеет радиальные складки и радиальные каналы, впадающие в кольцевой канал. Центральной частью пищеварительного аппарата является желудок, от которого отходит большое число разветвленных канальцев, выполняющих функции переноса питательных веществ в теле медуз.

Предротовые лопасти имеют многочисленные осязательные и стрекательные клетки. По краю зонтика расположены скопления нервных клеток — ганглии. Органы чувств сосредоточены в укороченных щупальцах — ропалиях. Внутри ропалии находится статоцист, а по бокам — два глазка, выполняющих светочувствительные функции. На щупальцах имеются обонятельные ямки — органы химического чувства.

В большинстве своем медузы раздельнополы. Половые продукты образуются в энтодерме: половые железы находятся в стенках желудка. Половые клетки выходят через рот в воду, где происходит копуляция мужских и женских гамет. Из оплодотворенных яиц развиваются микроскопических размеров личинки — планулы. Они плавают с помощью ресничек, затем опускаются на дно, прикрепляются к субстрату и превращаются в мелкие одиночные бокаловидной формы полипы — сцифистомы. По мере роста сцифистомы на ее теле появляются поперечные перетяжки, деля полип на ряд дисков — медуз (эфиры). Каждая эфира отделяется от сцифистомы, растет и превращается в свободноплавающую взрослую медузу. Таким образом, развитие сцифоидных медуз не прямое, а происходит через стадии планулы и сцифистомы (рис. 39).

КЛАСС КОРАЛЛОВЫЕ ПОЛИПЫ (Anthozoa)

Класс включает одну из древнейших групп морских животных — полипов, которые превосходят гидроидных полипов не только по размерам, но и отличаются более сложным строением. Это одиночные или большей частью колониальные полипы, одной из особенностей которых является отсутствие в жизненном цикле стадии медузы (рис. 40), т. е. у них нет чередования поколений. Это наиболее крупный класс кишечнополостных, включающий более 6 тыс. видов, обитающих в теплых тропических морях с температурой воды не ниже 20 °C на глубинах до 50 м.

Ротовое отверстие коралловых полипов окружено венчиком щупалец, число которых у одних полипов равно восьми (восьмилучевые кораллы), у других — шести (шестилучевые кораллы).

Пищевые частицы через рот попадают сначала в сплющенную с боков эктодермальную глотку, а оттуда — в хорошо развитую с перегородками (септами) кишечную полость. Число перегородок может быть либо восемь, либо шесть, или кратно шести — по числу щупалец. В глотке есть клетки с длинными ресничками, которые непрерывно гонят воду внутрь гастральной полости полипа, откуда вода выводится наружу. Так обеспечивается постоянная смена воды. Септы образованы мезоглеем, выстланной энтодермой (рис. 41). В нижней части полипа септы прикреплены только к стенке тела, в результате чего центральная часть гастральной полости (желудок) остается неразделенной.

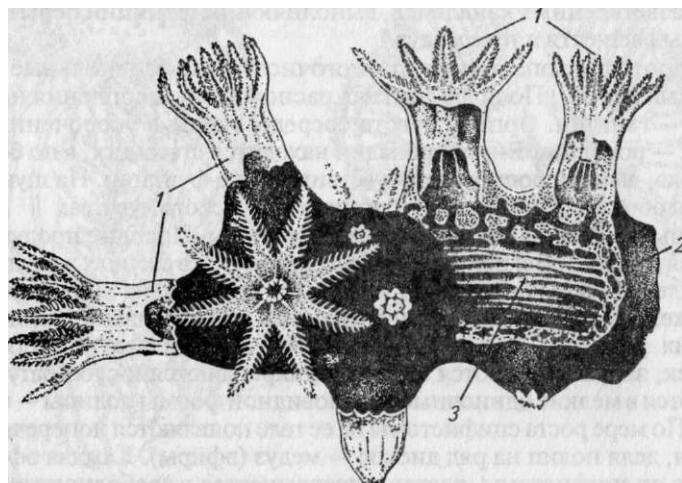


Рис. 40. Ветка колонии красного коралла:
1 — полипы; 2 — кора ветки; 3 — осевой скелет

Рис. 41. Схематическое изображение строения кораллового полипа *Alcyonium*:
1 — щупальце; 2 — ротовое отверстие; 3 — глотка; 4 — перегородки кишечной полости — септы; 5 — мезентериальные нити; 6 — яйца

У колониальных полипов мощный скелет чаще всего представлен углекислыми солями, реже — рогоподобным веществом. Скелет может быть наружным или внутренним.

Коралловые полипы размножаются бесполым и половым путем. Ведущие одиночный образ жизни актинии иногда размножаются делением, у колониальных видов наблюдается почкование. Половые железы формируются в перегородках между энтодермой и мезоглеем. Спермин выходят через ротовое отверстие наружу и через рот же проникают в гастральную полость женской особи, где и происходит оплодотворение. У некоторых форм оплодотворение наружное. Развитие происходит с метаморфозом: из зиготы развивается плавающая личинка — планула, которая прикрепляется к субстрату и дает начало новому полипу.

Актинии — одиночные шестилучевые яркоокрашенные полипы, лишенные скелета (рис. 42). Они могут медленно передвигаться с помощью мускулистой подошвы. Актинии очень чувствительны к раздражениям, сильно сокращаются, превращаясь в небольшой комок. Это хищники, питающиеся ракообразными, моллюсками и другими крупными животными, которых они захватывают щупальцами, парализуя стрекательными нитями. Некоторые актинии живут в симбиозе с раками-отшельниками, поселяясь на их раковинах. Рак служит для актиний средством передвижения, а актинии пассивно защищают рака от хищников.

В тропиках распространены рифообразующие мадрепоровые шес-

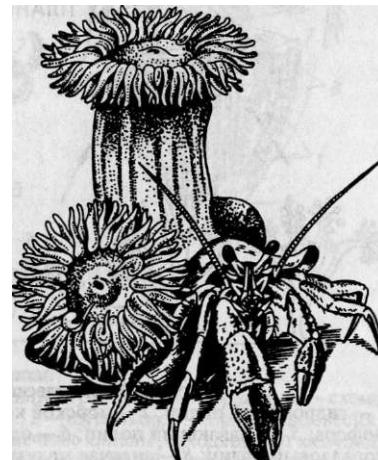
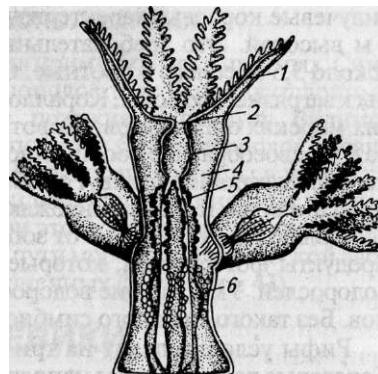


Рис. 42. Актиния на раковине, занятой раком-отшельником

тилучевые кораллы, характеризующиеся крупными размерами — более 4 м высотой. Это требовательные к температуре воды и ее солености (около 3,5 % солей) животные. Очень чувствительны коралловые полипы к загрязнению воды. Коралловые рифы служат местом обитания многих морских организмов. Животные и растения в коралловых рифах образуют своеобразное сообщество (биоценоз) рифа. В клетках энтодермы коралловых полипов живут симбиотические одноклеточные водоросли — зооксантеллы. Кораллы снабжают водоросли диоксидом углерода и предоставляют им укрытие, а от зооксантелл кораллы получают кислород и продукты фотосинтеза, которые поступают непосредственно из клеток водорослей. Умирающие водоросли перевариваются в цитоплазме полипов. Без такого сложного симбиоза коралловые полипы погибают.

Рифы условно делят на три типа: береговые, барьерные и атоллы. Береговые расположены непосредственно по берегам островов или материков, барьерные рифы располагаются параллельно береговой линии на некотором расстоянии. Атоллы — это кольцеобразные возвышающиеся над океаном коралловые острова с озерцом внутри.

Значение кишечнополостных в Мировом океане трудно переоценить: с их помощью осуществляется круговорот кальция в биосфере, они очищают морскую воду от органической взвеси, являются звеньями в пищевых цепях и т. п. Кишечнополостные служат объектом промысла: медузы (Япония, Китай), кораллы для украшений, коллекций и ювелирных изделий, медицинских препаратов.

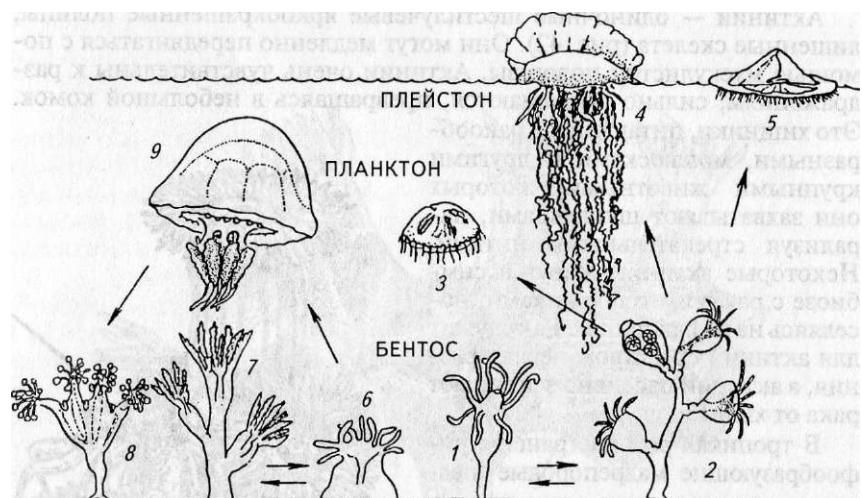


Рис. 43. Экологическая радиация кишечнополостных:
1 — гидроидный полип; 2 — морской колониальный полип; 3 — гидромедуза; 4 — сифонофора; 5 — плавающий полип; 6 — одиночный коралловый полип; 7 — колониальный коралловый полип; 8 — сидячая медуза; 9 — сцифоидная медуза

ФИЛОГЕНИЯ КИШЕЧНОПОЛОСТНЫХ

Кишечнополостные — древняя группа примитивных животных. Считается, что далекими предками кишечнополостных были двухслойные плавающие многоклеточные животные, похожие на планулу. Видимо, первыми были одиночные полипы. От полипов без перегородок развивались различные группы гидроидных. Коралловые полипы в процессе эволюции дали широкий спектр полипоидных форм: одиночных и колониальных, со скелетом и без него, но при этом сохранили древний признак развития без метагенеза. Основные пути морфо-экологической эволюции отражены на схеме радиации жизненных форм (рис. 43).

ТИП ГРЕБНЕВИКИ (*Ctenophora*)

Гребневики — это морские животные, характеризующиеся радиальной симметрией; они ведут одиночный свободноплавающий образ жизни, реже встречаются ползающие или сидячие формы. Известно около 120 видов гребневиков, заселяющих все моря. Питаются эти животные обычно планктоном.

Форма прозрачного и нежного тела мешковидная или грушевидная (рис. 44). Вдоль тела тянутся восемь рядов тонких и прозрачных гребных пластинок, образованных сросшимися ресничками. Пластинки

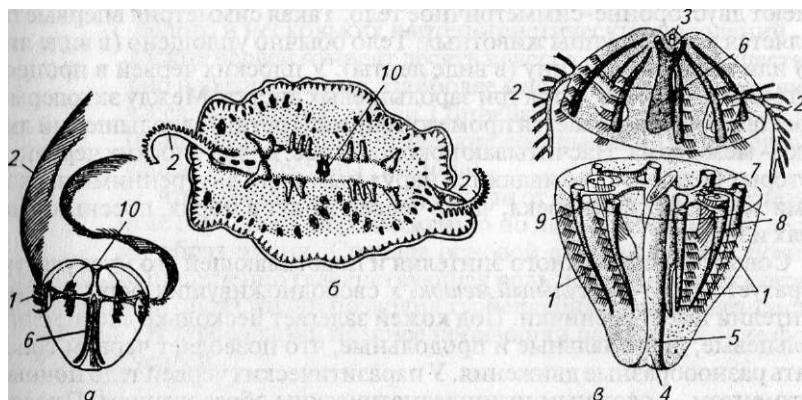


Рис. 44. Гребневики:
a — плавающий гребневик *Bolinopsis*; *б* — ползающий гребневик *Coeloplana*; *в* — схема строения гребневика; 1 — ряды гребных пластинок; 2 — щупальца; 3 — орган равновесия; 4 — рот; 5 — глотка; 6 — кишечная полость; 7 — кишечные каналы, отходящие от желудка; 8 — продольные кишечные каналы; 9 — влагалище щупальца; 10 — органы чувств

расщеплены наподобие гребешка и бьют по воде в одном направлении, что позволяет гребневикам двигаться только в одном направлении — ротовым концом тела вперед. Многие гребневики имеют пару щупалец с расположенными на них особыми клетками, вырабатывающими липкий секрет. С помощью этих щупалец гребневики захватывают разных животных, которыми питаются.

Рот, расположенный на оральном конце тела, ведет в глотку, которая переходит в короткий пищевод и затем в кишечную полость. Кишечная полость имеет разветвленные каналы и слепые отростки. Это двухслойные животные: снаружи тело покрыто эктодермой, а кишечная полость и ее каналы выстланы энтодермальным эпителием. Между экто- и энтодермой лежит студенистая прозрачная мезоглея. Наличие в мезогле многочисленных клеточных элементов и отсутствие стрекательных клеток отличает гребневиков от кишечнополостных.

Гребневики — гермафродиты; половые железы расположены в стенах пищеварительных каналов. Половые клетки выходят в просвет кишечных каналов и оттуда через рот наружу. Оплодотворение наружное. В развивающемся эмбрионе происходит образование зачаточного третьего зародышевого листка — мезодермы. Развитие без метаморфоза.

ТИП ПЛОСКИЕ ЧЕРВИ (*Plathelminthes*)

Общая характеристика. Большинство представителей этого типа имеют двусторонне-симметричное тело. Такая симметрия впервые появляется у этой группы животных. Тело обычно уплощено (в виде листа) или вытянуто в длину (в виде ленты). У плоских червей в процессе онтогенеза формируются три зародышевых листка. Между эктодермой и энтодермой развивается промежуточный (третий) зародышевый листок — мезодерма. Насчитывают около 15 тыс. видов плоских червей, из которых большинство являются наружными или внутренними паразитами животных и человека, часть червей живут в морях, пресных водоемах и почве.

Совокупность кожного эпителия и подстилающей его мускулатуры образует *кожно-мускульный мешок*. У свободноживущих форм кожный эпителий имеет реснички. Под кожей залегает несколько слоев мышц: кольцевые, диагональные и продольные, что позволяет червям совершать разнообразные движения. У паразитических червей тело покрыто тегументом — сложным цитоплазматическим образованием. Плоские черви не имеют полости тела (бесполостные), поскольку все пространство между внутренними органами и стенкой тела заполнено рыхло расположеннымными клетками мезодермального происхождения — паренхимой (паренхиматозные черви); в промежутках между клетками паренхимы циркулирует межтканевая жидкость. Паренхима выполняет опорные функции, в ней накапливаются резервные питательные вещества, она участвует в процессах обмена веществ.

Пищеварительный канал примитивен, обычно разветвлен и представлен двумя отделами: эктодермальной глоткой (передняя кишечник) и энтодермальной средней кишкой, которая заканчивается слепо. Задней кишки и анального отверстия нет. У части паразитических форм кишечник отсутствует.

Нервная система представлена парным мозговым ганглием и отходящими от него несколькими парами нервных стволов, идущих назад и соединенных между собой кольцевыми перемычками — комиссурами. Таким образом, у плоских червей формируется центральный аппарат 11 нервной системы. Органы чувств наиболее развиты у свободноживущих видов: имеются глазки, органы равновесия — статоцисты и многочисленные сенсиллы (осознательные клетки и органы химического чувства).

Кровеносная и дыхательная системы отсутствуют. Свободноживущие плоские черви дышат через кожу; для эндопаразитических форм характерно анаэробное дыхание.

У плоских червей появляются *органы выделения*, построенные по типу протонефридиев в виде системы разветвленных канальцев, оканчивающихся в паренхиме звездчатыми клетками с пучком ресничек внутри. Реснички способствуют откачке конечных продуктов обмена из паренхимы в один или два магистральных канала и затем через специальные выделительные отверстия (экскреторные поры) выводу этих продуктов наружу.

Плоские черви в большинстве своем гермафродиты. Половая система устроена сложно и обеспечивает внутреннее оплодотворение и высокую плодовитость. Развитие может быть прямым или с метаморфозом. Эндопаразитам присущи сложные жизненные циклы с чередованием обоеполого и нескольких партеногенетических поколений.

К типу плоских червей относят десять классов, из которых шесть — исключительно паразитические. Наиболее многочисленными являются четыре класса: Ресничные черви (*Turbellaria*), Дигенетические сосальщики (*Trematoda*), Моногенетические сосальщики (*Monogenea*) и Ленточные черви, или Цестоды (*Cestoda*).

Считается, что плоские черви произошли от древних кишечнополостных, которые перешли к передвижению по дну, где они могли вести хищнический образ жизни. Сначала основой для движения служили I-спички, но постепенно главенствующая роль перешла к мускулатуре гена. Активный образ жизни позволил турбелляриям усложнить систему органов. От турбеллярий позднее произошли паразитические формы плоских червей.

КЛАСС РЕСНИЧНЫЕ ЧЕРВИ (*Turbellaria*)

К классу ресничных червей относится большая группа (около 10 000 видов) свободноживущих в воде или в почве плоских червей, гибкость которых не расчленено и покрыто мерцательным (ресничным) мштелием. Все турбеллярии — хищники. На переднем конце тела ресничных червей имеется несколько примитивных глазков. У большин-

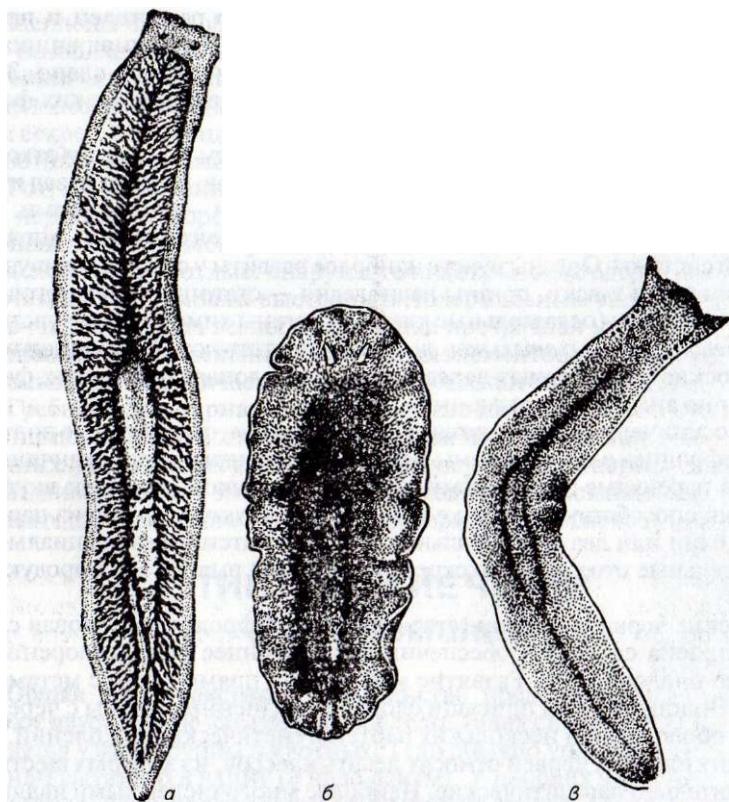


Рис. 45. Виды ресничных червей:
а — молочная планария *Dendrocoelum lacteum*; б — морская турбеллярия *Leptoplana*;
в — планария многоглазка *Polycoelis cornuta*

ства представителей рот расположен посередине тела на его брюшной стороне.

Строение и жизненные отравления. Длина тела может колебаться от долей миллиметра до 35 см. Форма тела уплощена и чрезвычайно разнообразна. Морские турбеллярии ярко окрашены (рис. 45).

Покровы представлены ресничным однослойным эпителием. Реснички способствуют передвижению мелких червей в воде, а более крупные представители ползают, вытягивая, сокращая и изгибая тело. В кожном эпителии располагаются особые палочковидные образования — рабдиты, выполняющие защитные функции: выбрасываясь наружу, они окутывают врага рыхлой клейкой оболочкой. В покровах ресничных червей много железистых клеток, одни из которых выделяют слизь, а другие — ядовитые вещества. Так, молочная планария на-

крыывает жертву своим телом и убивает ее ядом, который вырабатывают ядовитые железы, находящиеся на брюшной стороне тела червя.

Нервная система у разных представителей различна по своей сложности. У примитивных форм она диффузного типа. Есть виды, у которых вдоль тела идет несколько нервных тяжей. У более сложно организованных имеются ганглии с продольными нервными тяжами. Органы чувств представлены примитивными глазками, статоцистами и осознательными клетками.

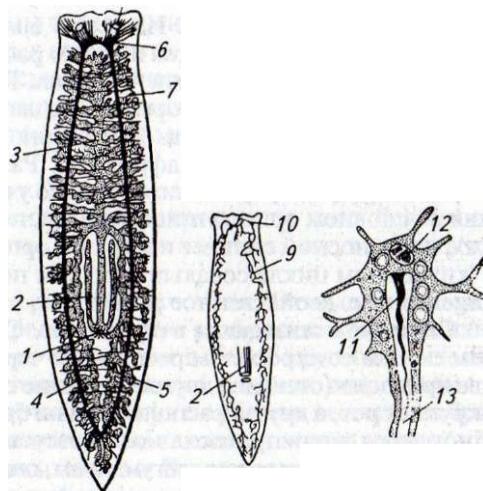
Органы пищеварения. У большинства видов на брюшной стороне в средней ее части расположен рот, ведущий в глотку, которая может выпячиваться наружу, присасываться к жертве и высасывать ее содержимое. Кишечник чаще всего ветвится на две и более ветвей, заканчивающихся слепо. Пища переваривается в полости кишечника и с помощью фагоцитоза (внутриклеточно) в клетках эпителия кишечной полости. Непереваренные остатки пищи выбрасываются наружу через рот. У некоторых ресничных червей кишечник отсутствует, а пища переваривается в пищеварительных вакуолях клеток, расположенных в специально обособленной пищеварительной паренхиме.

Газообмен осуществляется путем диффузии кислорода из воды через покровы внутрь тела, а диоксида углерода — наружу.

Выделительная система протонефридиального типа (рис. 46) как отдельная система органов впервые появляется у ресничных червей. У многих морских червей органов выделения нет: продукты обмена удаляются из тела через покровы и стенки кишечника.

Органы размножения устроены сложно. Большая часть ресничных червей гермафродиты. Мужская половая система представлена множе-

Рис. 46. Нервная система, органы пищеварения и выделения планарии:
а — пищеварительная и нервная системы; б — расположение главных канальцев выделительной системы; в — одна из концевых клеток протонефридиальной системы; / — рот; 2 — глотка; 3 — передняя ветвь кишечника; 4,5 — задние ветви кишечника; 6 — головной нервный узел; 7 — боковой нервный ствол; 8 — глазок; 9 — канальцы выделительной системы; К — выделительная пора; 11 — «мерцательное пламя»; 12 — ядро клетки; 13 — внутриклеточный каналец



ством мелких семенников, разбросанных в паренхиме. От семенников отходят семявыносящие каналы, которые, сливаясь, образуют два семяпроводы. Семяпроводы формируют непарный семязвергательный канал, пронизывающий совокупительный орган, расположенный в половой клоаке. Сюда же впадают и женские половые протоки.

Женская половая система представлена одним или множеством яичников. От яичников отходят два яйцевода, принимающие протоки желточников и сливающиеся в один канал — влагалище. Влагалище открывается в половую клоаку. Оплодотворенные яйцеклетки окружаются желточными клетками и вместе с ними покрываются общей скорлупой.

Благодаря разным срокам созревания половых продуктов самооплодотворения у этих червей не происходит. Оплодотворение внутреннее. Оплодотворенные яйца выводятся наружу либо через разрывы стенок тела, либо через рот, либо через специальные выводные протоки. У пресноводных форм развитие прямое. У морских видов развитие с превращением: из яйца развивается планктонная личинка, плавающая с помощью ресничек в толще воды.

Интересна способность ресничных червей к регенерации: при расчленении одного червя на сотни частей из каждой части может восстановиться новая особь.

В морях и океанах обитают мелкие ресничные черви из отрядов Бескишечные, Макростомиды, Многоветвистокишечные и др. Среди пресноводных представителей отечественной фауны можно отметить молочно-белую планарию, многочисленные трехветвистокишечные турбеллярии населяют озеро Байкал. Многие виды ресничных червей служат кормом для рыб.

КЛАСС СОСАЛЬЩИКИ (*Trematoda*)

Общая характеристика. Известно более 4 тыс. видов сосальщиков, многие из которых являются широко распространенными и опасными эндопаразитами животных и человека. Тело сосальщиков не расчленено и имеет листовидную форму. У большинства есть присоски для прикрепления к телу хозяина. Кишечник двутветственный, заканчивается слепо. Большинство гермафродиты. Развитие происходит со сменой хозяев. Взрослые формы живут только у позвоночных животных, поражая в основном органы пищеварения, но есть виды, обитающие в легких, кровеносной системе и в других органах. В связи со сменой хозяев в жизненном цикле сосальщиков они получили еще одно название — *Digenea*, т. е. двойственное развитие.

Строение и жизненные отправления. Строение сосальщиков во многом сходно со строением ресничных червей. На теле обычно имеются две присоски (отсюда и другое название сосальщиков — двуустки): одна окружает рот, а другая расположена на брюшной стороне тела (рис. 47). Иногда брюшная присоска может отсутствовать.

Покровы представлены тегументом, снаружи у некоторых форм усеянным шипиками, способствующими фиксации паразита в теле хозяина.

Рис. 47. Строение половой (л) и пищеварительной (б) систем печеночного сосальщика фасциолы:

1 — ротовая присоска; 2 — брюшная присоска; 3 — разветвленный кишечник (левая и правая ветви); 4 — копулятивный орган; 5 — яичник; 6 — желточники; 7 — желточные протоки; 8 — семенники; 9 — семяпроводы; 10 — матка

Мускулатура представлена слоями мышечных волокон, которые вместе с тегументом образуют кожно-мускульный мешок. Двигаются трематоды медленно.

Нервная система слагается из парного головного ганглия и отходящих от него парных нервных тяжей. От ганглия и тяжей идут ответвления ко всем органам. У взрослых червей органы зрения отсутствуют. В покровах размещены осязательные и другие нервные окончания.

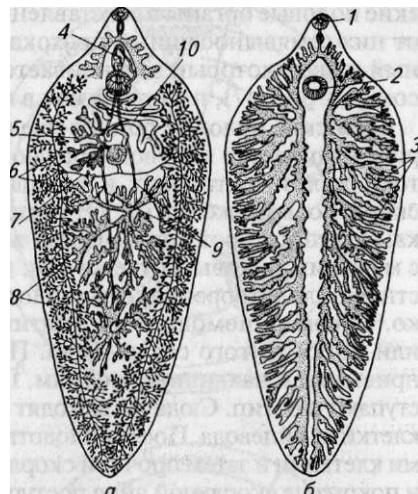
Органы пищеварения начинаются ротовым отверстием, ведущим в глотку, которая может совершать сосательные движения. За глоткой лежит пищевод, который ветвится на две ветви кишечника. Иногда ветвистый кишечник имеет дополнительные боковые отростки, облегчающие распределение продуктов пищеварения в паренхиме червя. Анальное отверстие отсутствует

Пищеварение в основном происходит в полости кишечника (внеклеточное). Наряду с кишечным пищеварением наблюдается всасывание растворенных органических веществ через покровы тела. Непереваренные остатки пищи выбрасываются через ротовое отверстие. Продукты пищеварения транспортируются в теле сосальщиков с помощью межтканевой жидкости и клеток паренхимы за счет сокращения мускулатуры тела.

Органы дыхания отсутствуют. У эндопаразитических форм дыхание анаэробное, процессы диссимиляции происходят по типу брожения. Но есть и исключения: паразиты, живущие в легких, могут дышать через покровы своего тела.

Органы выделения протонефридиального типа. От выделительных звездчатых клеток, покрытых ресничками, и разбросанных в паренхиме, отходят канальцы, которые сливаются в более крупные выделительные каналы. Вся выделительная система каналов открывается в мочевой пузырь, а из него конечные продукты обмена веществ выбрасываются через выделительное отверстие на заднем конце тела.

Органы размножения устроены сложно. Все трематоды гермафродиты, лишь немногие виды кровяных сосальщиков раздельнополы. Муж-



ские половые органы представлены двумя семенниками и отходящими от них семявыносящими протоками, образующими семязвергательный канал, который заканчивается копулятивным органом — циррусом (см. рис. 47), находящимся в половой клоаке.

Женские половые органы представлены одним яичником, от которого начинается яйцевод, впадающий в оотип. Сюда же впадают протоки желез: желточных и тельца Мелиса. Оотип окружен мелкими скорлуповыми железами. От оотипа начинается длинная извитая матка, конец которой открывается женским половым отверстием рядом с мужским половым отверстием. Сосальщикам свойственно перекрестное оплодотворение, самооплодотворение возможно, но очень редко. Сперма с помощью копулятивного органа вводится в свою матку или матку другого сосальщика. По матке спермии поступают в семяприемник и накапливаются там. По мере необходимости спермии поступают в оотип. Сюда же выходят продукты придаточных желез и яйцеклетки из яйцевода. После оплодотворения яйца покрываются желточными клетками и затем прочной скорлуповой оболочкой. Оплодотворенные и покрытые скорлупой яйца поступают в матку и выводятся наружу. Яйцо имеет крышечку, открывающуюся при выходе из него личинки.

Развитие большинства сосальщиков протекает со сложными превращениями и со сменой хозяев. Первые промежуточные хозяева — всегда брюхоногие моллюски, пресноводные или наземные. Вторые промежуточные (дополнительные) хозяева (если они есть) — разные беспозвоночные и позвоночные животные. Есть виды сосальщиков, которые имеют трех промежуточных хозяев (рис. 48).

Наиболее опасны как паразиты сельскохозяйственных животных и человека следующие дигенетические сосальщики.

Печеночный сосальщик (*Fasciola hepatica*), или фасциола печеночная, имеет листовидное тело до 5 см длиной (см. рис. 47). На переднем конце тела расположено ротовое отверстие, окруженное ротовой присоской. Тегумент с шипиками. На брюшной стороне тела имеется брюшная присоска. Кишечник двутрубистый с множеством отростков. Два ветвистых семенника расположены в середине тела ниже компактного ветвистого яичника. По бокам тела находятся желточники.

Печеночный сосальщик паразитирует в желчных протоках печени растительноядных и всеядных животных, может поражать и человека, вызывая заболевание фасциолез. Питается сосальщик желчью. Сильнее всего поражаются овцы и молочный скот, которых пасут в поймах рек, на заливных лугах и т. п. Нередко болезнь может иметь летальный исход из-за закупоривания двустками желчных протоков и невозможности оттока желчи из печени.

В желчных протоках паразиты копулируют, но возможно и самооплодотворение. С желчью оплодотворенные яйца, покрытые скорлуповыми оболочками, через кишечный тракт с калом хозяина попадают во внешнюю среду. За сутки один паразит может отложить сотни тысяч яиц (рис. 49).

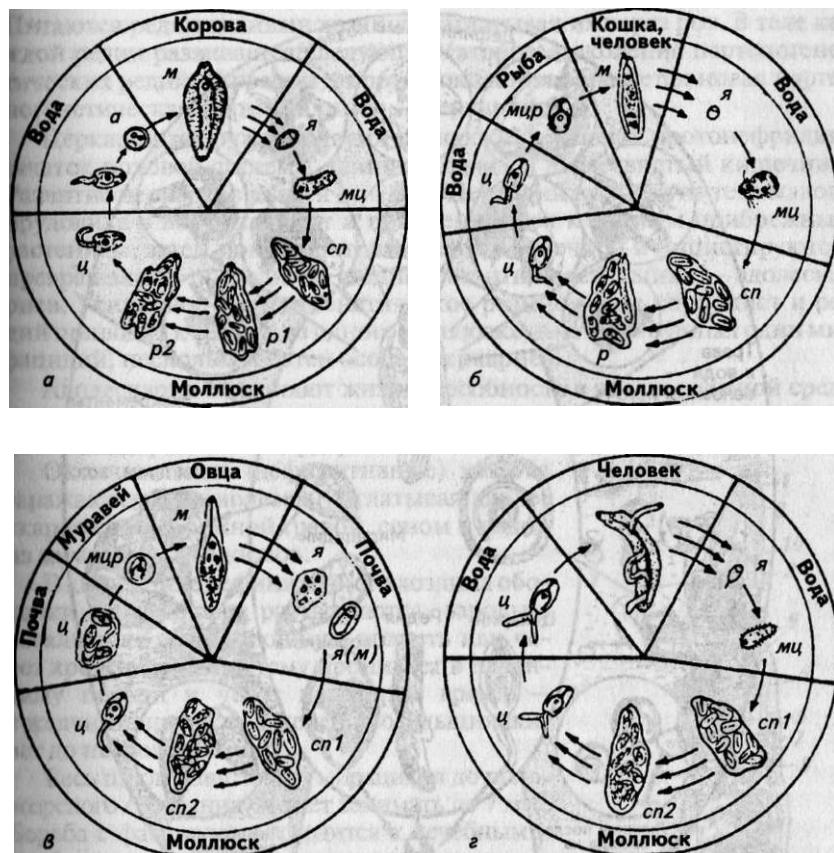


Рис. 48. Схемы жизненных циклов третратод:
 а — печеночный сосальщик; б — кошачья двуустка; в — ланцетовидный сосальщик;
 г — кровяной сосальщик; м — марита; я — яйцо; миц — мирадицид; сп — спороциста;
 р — редия; и — церкарий; мир — метацеркарий

Для дальнейшего развития яйцо должно попасть в воду, где через 3—6 нед крышечка открывается и из яйца выходит микроскопическая личинка, покрытая ресничками, — мирадицид. На переднем конце мирадиции имеются глазки и рот, ведущий в кишечник. Свободно плавать мирадиции могут не более 2 сут. Они не питаются. Если мирадиции не попадут в тело промежуточного хозяина, то они погибают. В задней части тела мирадиции лежат партеногенетические яйца. Найдя промежуточного хозяина (пресноводный брюхоногий моллюск малый прудовик, *Limnaea truncatula*), мирадицид с помощью специальных железок, расположенных около рта, проникает в моллюска и превращается в следующую стадию развития — спороцисту. Это половозрелая ста-

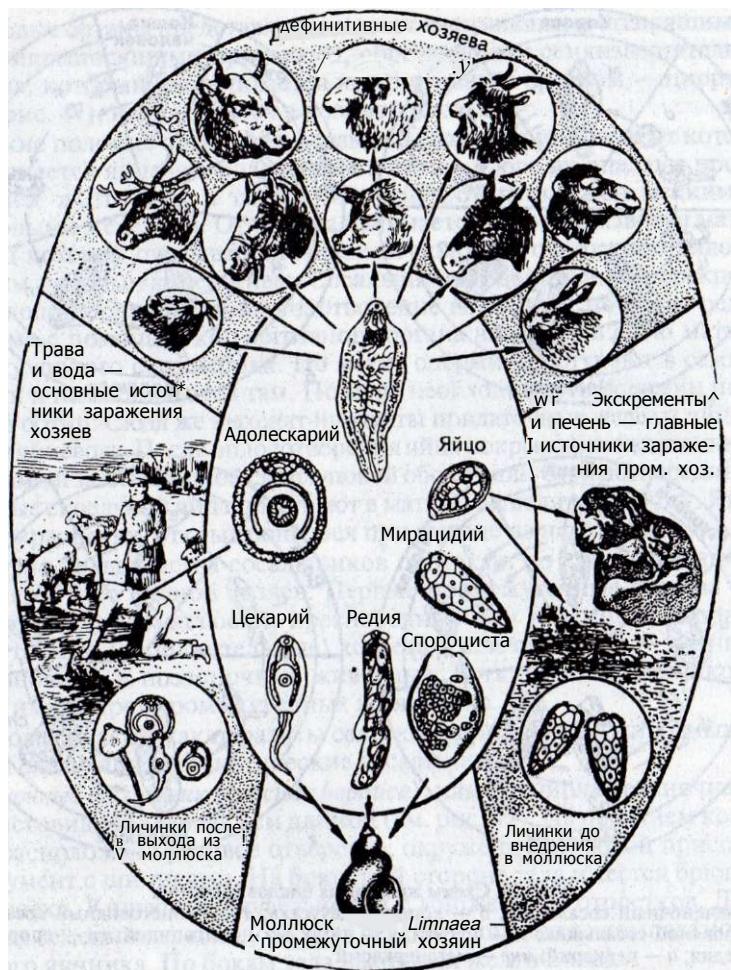


Рис. 49. Цикл развития печеночного сосальщика фасциолы (*Fasciola hepatica*)

дия, которая способна к размножению. Иными словами, мириацидий является как бы личинкой спороцисты.

Спороциста лишена ресничек, она выделяет в ткани хозяина пищеварительные ферменты и питается продуктами гидролиза тканей хозяина, поглощая эти продукты всей поверхностью тела. Внутри спороцисты путем дробления и дифференцировки в течение 3—4 нед из партеногенетических яиц развивается новое поколение личинок — редии. Редии выходят из погибшей спороцисты через разрывы ее стенок. Редии имеют вытянутое тело с развитым пищеварительным аппаратом.

11итаются редии тканями хозяина, заглатывая их через рот. В теле каждой редии развивается следующее (второе) поколение партеногенетических редий. Из редий второго поколения образуется новая партеногенетическая форма личинок — церкарии.

Церкарии вооружены двумя присосками, имеют протонефридии, зачаток половой системы, длинный хвост и двуветвистый кишечник. Развитие церкариев длится до 6 нед. Церкарии выходят из тела малого прудовика в воду, плавают и прикрепляются к водным прибрежным растениям, затем покрываются плотной оболочкой и инцистируются, превращаясь в последнюю стадию развития сосальщика —adolескариев. Усиленное партеногенетическое размножение спороцист и редий приводит к выходу из одного моллюска, в которого попал один мириацидий, нескольких сотен особей церкариев.

Адолескарии сохраняют жизнеспособность в воде и влажной среде многие месяцы, долго живут они в сене, заготовленном из растений, на которых закрепились церкарии.

Окончательные (дефинитивные) хозяева заражаются фасциолезом, заглатывая адолоскариев с прибрежной травой, сеном и водой из зараженных водоемов.

В кишечнике дефинитивного хозяина оболочки адолоскариев растворяются, зародыш сосальщика через брюшную полость или через кровеносную систему проникает в паренхиму печени и через некоторое время — и желчные протоки. В печени сосальщик живет до нескольких лет.

Весь цикл развития (от мириацидия до половозрелого состояния) может занимать до 7 мес. Борьба с фасциолезом сводится к лечебным и профилактическим мерам с целью предупредить попадание адолоскариев в организм сельскохозяйственных животных (мелиорация пастбищ, поение скота чистой водой из специальных сооружений, смена пастбищ и т. д.).

Ланцетовидный сосальщик (*Dicrocoelium lanceatum*), или ланцетовидная двуустка, — небольшой червь (около 1 см) ланцетовидной формы, распространен в засушливых регионах страны. Имеет две присоски, двуветвистый кишечник без боковых отростков, яичник и семенники неветвящиеся (рис. 50). Паразитирует в печени мелкого и крупного рогатого скота и других травоядных млекопитающих.

Первым промежуточным хозяином лан-
I (стовидного сосальщика служат разные виды сухопутных брюхоногих моллюсков. С кало-

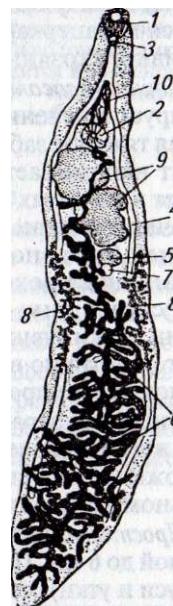


Рис. 50. Ланцетовидный сосальщик *Dicrocoelium lanceatum*:

1 — ротовая присоска; 2 — брюшная присоска; 3 — глотка; 4 — ветви кишечника; 5 — яичник; 6 — матка; 7 — семяприемник; 8 — желточники; 9 — семенники; 10 — семязвергательный канал

выми массами дефинитивного хозяина выходят яйца сосальщика, окруженные толстыми оболочками, которые позволяют сохранять жизнеспособность в течение нескольких месяцев. В яйцах находятся сформированные мирадии. Моллюски заражаются, поедая эти яйца. В кишечнике моллюсков мирадии освобождаются от яйцевых оболочек и проникают в печень промежуточного хозяина. Там мирадии превращаются в спороцисты, в которых партеногенетически развиваются дочерние спороцисты. Последние дают начало партеногенетическим церкариям. Сотни церкариев проникают в легкие моллюска, обволакиваются слизью в комочки (пакеты церкарий) и в таком виде выбрасываются через дыхательное отверстие наружу.

Вторым промежуточным хозяином являются разные виды муравьев. Муравьи могут поедать слизистые комочки с церкариями. Через стенку кишечника муравьев церкарии проникают в полость тела насекомых, где превращаются в метацеркариев (без хвоста, окруженные оболочками).

Скот, заглатывая траву с пораженными муравьями, заражается сосальщиком. В кишечнике скота муравьи перевариваются, а освободившиеся метацеркарии через желчный проток попадают в печень дефинитивного хозяина.

Кошачий сосальщик (*Opistorchisfelineus*), или кошачья двуустка, паразитирует в печени хищных млекопитающих, а иногда и человека, вызывая тяжелое заболевание — описторхоз. По строению и размерам паразит напоминает ланцетовидного сосальщика. Кошачий сосальщик живет в желчных протоках печени, в желчном пузыре и даже в поджелудочной железе. Первым промежуточным хозяином служит пресноводный брюхоногий моллюск битиния (*Bithynia leachi*), в котором мирадий проходит те же стадии цикла развития, что и печеночный сосальщик: мирадий—спороциста—редии—церкарии. Церкарии, покинувшие первого промежуточного хозяина, выходят в воду и затем активно внедряются в тело рыб, в основном различных видов карповых, где превращаются в метацеркариев.

Хищники заражаются, съедая рыбу, пораженную метацеркариями. Заражение человека и животных происходит при поедании вяленой, мороженой или сырой рыбы, инфицированной метацеркариями. При сильном заражении болезнь может закончиться смертью человека.

Простогонимусы (виды рода *Prosthogonimus*) — небольшие сосальщики, длиной до 6 мм. Паразитируют эти черви в яйцеводах птиц (куриные, реже гуси и утки), что приводит к образованию бесскорлупных яиц и к последующему прекращению яйцекладки. Промежуточными хозяевами являются различные виды пресноводных моллюсков, а дополнительными хозяевами — личинки стрекоз. Яйца развиваются в воде. Вышедшие из них мирадии проникают в тело моллюсков, в печени которых превращаются в спороцисты. Последние дают начало партеногенетическим церкариям. Выходя из тела моллюсков, церкарии с водой попадают в кишечник личинок стрекоз. Там они лишаются хвоста, проникают в разные части тела личинок и превращаются в метацеркариев. Поедая личинок и взрослых стрекоз, птицы заражаются простогонимусами.

Кровяные двуустки (несколько видов из рода *Schistosoma*) существенно отличаются от других групп трематод. Длина тела до 2 см. Они раздельнополы. Самка размещается в специальном желобе на брюхе самца (рис. 51). Паразитируют они в венах пищеварительной и выделительной систем птиц, млекопитающих и человека. Оплодотворенные яйца с помощью особого шипа проникают в заднюю часть кишечника и с калом (у птиц с пометом) выносятся наружу. В воде из яиц выходят мирандии и внедряются в пресноводных брюхоногих моллюсков. В моллюсках паразит проходит все стадии спороцисты, дочерней спороцисты и перкария. Из тела моллюска церкарии выходят в воду и через кожу проникают в дефинитивного хозяина, вызывая тяжелое заболевание — шистосомоз. В тропических странах шистосомозом поражено около 900 млн человек.

Для нормального прохождения всего ЦИК-ла развития сосальщикам требуются весьма благоприятные условия: наличие воды, промежуточных хозяев в ней, постоянное посещение водоемов дефинитивными хозяевами и т. д. Однако есть виды, отлично приспособившиеся к паразитированию. Например, сосальщик *Leucochloridium paradoxum* живет в кишечнике насекомоядных певчих птиц. В теле промежуточного хозяина — наземной улитки, съевшей яйца паразита, мирадии превращаются в спороцисты. Спороцисты разрастаются, образуя большое число отростков, распространяющихся по телу улитки. Некоторые отростки попадают в щупальца улитки, сильно раздуваются, приобретают яркую окраску, просвечивающую сквозь тонкую кожу улитки, и периодически сокращаются. Птицы склевывают эти щупальца, похожие на гусениц насекомых, заражаясь таким образом сосальщиком.

КЛАСС МОНОГЕНЕИ (*Monogenea*)

За редким исключением моногенетические сосальщики являются эктопаразитами позвоночных (рыб, амфибий) и беспозвоночных (моллюсков) животных. Известно более 2,5 тыс. видов моногеней. Паразитируют они на жабрах и коже рыб, некоторые поражают мочевой пузырь амфибий и рептилий. По своей организации моногеней близки к трематодам. Они обладают мощными органами для прикрепления к хозяину. Это присоски и крючья или только крючья, которые расположены на обособленном заднем отделе тела в виде диска, а также мелкие присоски около рта, выделяющие липкий секрет. Ротовая и брюш-

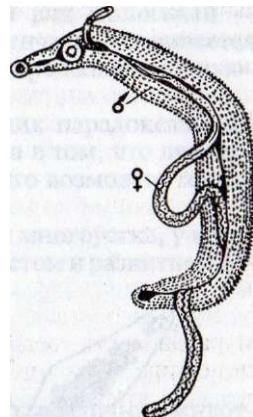


рис. 51. Кровяная двуустка
Schistosoma haematobium:
Г — лобке более широкого
самца (σ^{\wedge}) Т —

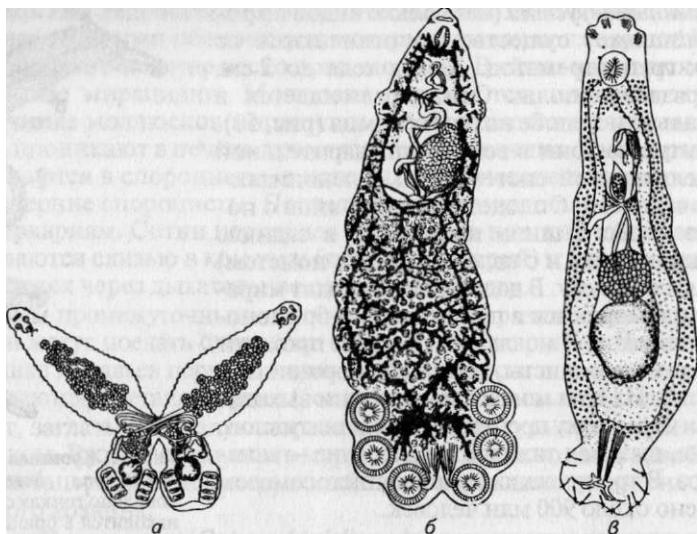


Рис. 52. Моногенетические сосальщики:
а — спайник парадоксальный *Diplozoon paradoxum*; б — лягушачья многоустка *Polystoma inlegerrimum*; в — дактилогирус *Dactylogyrus vastator*

ная присоски у моногеней отсутствуют (рис. 52). Из органов чувств можно отметить наличие на покровах многочисленных чувствующих клеток и на переднем конце — глазков.

Тело моногеней заключено в кожно-мускульный мешок. Кишечник двуветвистый, иногда с боковыми отростками или мешковидный. Половые железы непарные. Парные выделительные каналы протонефридиальной системы открываются на переднем конце тела парными отверстиями.

В половой системе имеется влагалище, по которому сперма вводится в оотип и семяприемник. Матка открывается самостоятельным отверстием в половую клоаку и служит только для выведения оплодотворенных яиц.

Размножаются моногеней исключительно половым путем, некоторым видам свойствен партеногенез. Жизненный цикл без смены хозяина, все развитие паразита проходит в одном хозяине.

Из оплодотворенного яйца выходит свободноплавающая личинка с ресничками, глазками и органами прикрепления на заднем конце тела. Личинка напоминает планарию.

Некоторые моногенетические сосальщики приносят вред рыбному хозяйству, поражая рыб и вызывая их истощение. Например, мелкие черви *Dactylogyrus vastator* длиной 1—3 мм живут на жабрах и коже карповых и других рыб, питаясь кровью. Из яиц, отложенных дактилогирусом, вылупляются личинки, которым затем надо прикрепиться к жабрам рыб, где они превращаются во взрослых паразитов.

На карпах часто паразитирует живородящий вид моногеней — *(Iyrodactylus elegans)*. В этом паразите партеногенетически развивается **только один** зародыш, в котором имеется еще три зародыша последующих поколений.

На жабрах карповых рыб паразитирует спайник парадоксальный (*Diplozoon paradoxum*). Особенность этого паразита в том, что две гермафродитные особи срастаются таким образом, что возможно только перекрестное оплодотворение.

Есть паразиты лягушек, в частности лягушачья многоустка, у которой жизненный цикл усложнен и тесно связан с ростом и развитием хозяев — головастиков и лягушек.

КЛАСС ЛЕНТОЧНЫЕ ЧЕРВИ (Cestoda)

Общая характеристика. Цестоды — эндопаразиты различных животных, преимущественно позвоночных, и человека. Взрослые черви паразитируют в тонком отделе кишечника дефинитивного хозяина; личинки паразитов развиваются в различных органах и полостях тела промежуточного хозяина — беспозвоночных и позвоночных животных. Известно более 3 тыс. видов цестод, среди которых много паразитов животных и человека.

У большинства представителей ленточных червей тело имеет вид Itosкой ленты, часто расчлененной на множество члеников. Тело имеет головку — сколекс, которая продолжается в шейку; за шейкой следует тело червя — стробила. Стробила состоит из множества (от нескольких тысяч до сотен, но бывает и два—четыре) члеников — пролоттид. Реже встречаются цестоды с нерасчлененным телом. Головка цестод имеет специальные органы прикрепления: присоски, крючья, ботрии (щелевидные углубления).

В связи с паразитическим образом жизни у ленточных червей слабо развиты нервная система и органы чувств, редуцирована пищеварительная система. Однако половая система достигает высокого уровня развития, обеспечивая огромную плодовитость, а следовательно, и возможность выживания паразитов.

Строение и жизненные отправления. Длина тела колеблется от нескольких миллиметров до 15 м. Головка (сколекс) имеет разное строение у различных цестод (рис. 53). У бычьего цепня сколекс

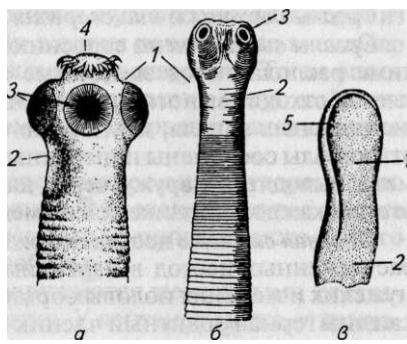


Рис. 53. Головки цепней и лентецов:
а — цепня вооруженного; б — цепня не вооруженного; в — лентеца широкого;
1 — головка; 2 — шейка; 3 — присоски;
4 — хоботок с крючьями; 5 — присасывающие ямки — ботрии

леке имеет только четыре присоски (невооруженный цепень), головка свиного цепня помимо четырех присосок на вершине вооружена дополнительно венчиком из хитиновых крючьев (вооруженный цепень), головка широкого лентеца имеет два щелевидных углубления — ботрии, которыми они зажимают складку стенки кишечника хозяина. Еще более сложны органы прикрепления у гвоздичника и других цестод.

Шейка цестод является зоной роста черва, так как в ней происходит отшнуровывание новых членников, из которых состоит стробила. Только у немногих представителей (ремнец, гвоздичник) тело не подразделяется на членники. На заднем конце тела цестод находятся зрелые членники, наполненные яйцами червя. Они отрываются по мере созревания и увлекаются с калом хозяина во внешнюю среду. Таким образом, у цестод происходит постоянный прирост молодых членников и отрыв старых (зрелых) членников. Число членников у разных цестод может варьироваться в широких пределах: от двух—четырех до нескольких тысяч. В передней части стробилы расположены незрелые членники, у которых еще не развиты половые органы; за незрелыми следуют гермафродитные членники с развитой гермафродитной половой системой. Конец стробилы представлен зрелыми членниками с маткой, набитой яйцами.

Покровы представляют собой тегумент, который подстилают кольцевые и продольные слои мускулатуры. По сравнению с trematodами тегумент у цестод выполняет более многообразные функции: защита от действия пищеварительных ферментов кишечника хозяина путем их нейтрализации, всасывание питательных веществ из кишечного содержимого хозяина, выработка и выделение собственных ферментов и т. п. Тегумент имеет волоски и ворсинки, увеличивающие поверхность всасывания пищи.

Мускулатура представлена наружным кольцевым и внутренним продольным слоями. Может быть и третий — диагональный слой. В кишечнике хозяина ленточные черви совершают медленные движения. Такие же движения совершают и вышедшие наружу с калом зрелые членники.

Нервная система состоит из скопления в сколексе червя нервных клеток и продольных парных тяжей, идущих до конца тела. Органы чувств выражены слабо.

Органы дыхания и пищеварения у цестод отсутствуют.

Органы выделения по строению однотипны с trematodами. В паренхиме располагаются звездчатые клетки, несущие реснички; от этих клеток отходят выносящие канальцы, сливающиеся в два крупных выделительных канала, идущих по бокам стробилы. В каждом членнике эти каналы соединены поперечным протоком. Конечные продукты обмена выводятся наружу через каналы последнего членника. Помимо этого в каждом членнике тоже имеются отверстия протонефридиев.

Половая система цестод похожа наловую систему trematod. У нерасчлененных цестод в паренхиме расположен лишь один комплект мужских и женских половых органов (рис. 54). У расчлененных цестод каждый гермафродитный членник имеет по одному комплекту женских

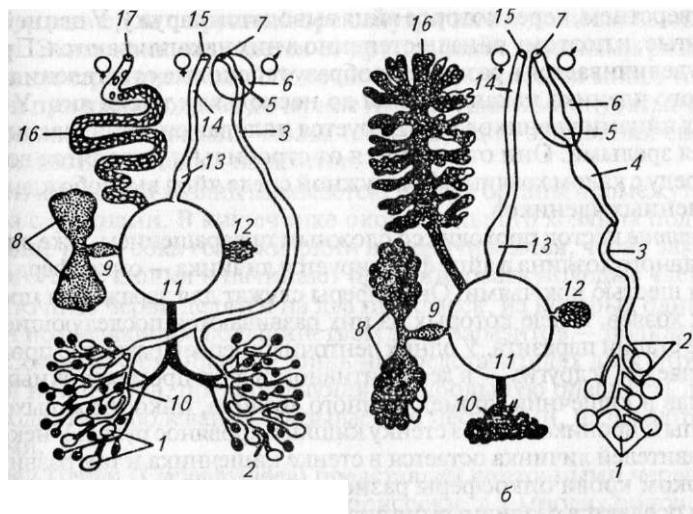


Рис. 54. Схема строения половой системы ленточных червей:
 ч — лентец (с открытой маткой); б — цепень (с закрытой маткой); 1 — семивыносящие каналы; 2 — семяпровод; 3, 4 — семяпровод; 5 — совокупительный орган; 6 — его сумка; 7 — мужское половое отверстие; 8 — яичники; 9 — яйцевод; 10 — желточник; II — оотип; 12 — тельце Мелиса; 13 — семяприемник; 14 — влагалище; 15 — женское половое отверстие; 16 — матка; 17 — отверстие матки

п одному комплекту мужских половых органов, а у некоторых представителей — по два комплекта. Мужская половая система представлена множеством мелких семенников, разбросанных в паренхиме. Отходящие от семенников семивыносящие каналы сливаются в семяпровод, который заканчивается семязвергательным каналом и копулятивным органом.

Женская половая система состоит из яйцевода, оотипа, матки, влагалища, семяприемника, желточников и тельца Мелиса. Влагалище и семязвергательный каналы открываются в половую клоаку. У одних ленточных червей матка открывается наружу специальным отверстием — открыта матка. В этом случае оплодотворенные яйца постоянно выходят из тела червя и вместе с каловыми массами хозяина попадают во внешнюю чаду. У других червей матка не имеет выводного отверстия. Тогда оплодотворенные яйца или развивающиеся в них личинки могут выйти из яйцеклетки дефинитивного хозяина только вместе с оторвавшимися от стробилы зрелыми членниками.

У ленточных червей происходит перекрестное оплодотворение, но может происходить и самооплодотворение. В последнем случае сперма может вводиться во влагалище собственного членика, а также любого другого гермафродитного членика стробилы.

Оплодотворенные в оотипе яйца окружаются желтовыми клетками и скорлупой, затем выводятся в матку. У лентецов матка открывается

ется отверстием, через которое яйца выводятся наружу. У цепней матки замкнутые, и поэтому яйца постепенно в них накапливаются. При этом матка увеличивается в размерах и образует боковые ответвления. В матке одного членика накапливается до нескольких тысяч яиц. У наполненных яйцами члеников редуцируется половая система, членики становятся зрелыми. Они отрываются от стробилы и выводятся во внешнюю среду с калом хозяина. В наружной среде яйца высвобождаются из разрушенных члеников.

Развитие цестод проходит со сложным превращением. Уже в теле дефинитивного хозяина в яйце формируется личинка — онкосфера, вооруженная шестью крючьями. Онкосфера служат для заражения промежуточных хозяев, в теле которых из них развиваются последующие личиночные стадии паразита. У одних ленточных червей развитие проходит в двух хозяевах, у других — в дефинитивном и двух промежуточных.

Попав в кишечник промежуточного хозяина, онкосфера выходит из скорлупы и проникает через стенку кишки в кровяное русло. У некоторых представителей личинка остается в стенке кишечника и там развивается.

С током крови онкосфера разносятся по телу промежуточного хозяина и оседают в различных органах и тканях, где образуют новую стадию личинки — финну. Строение финн у цепней неодинаково. Различают три типа финн: цистицерк, ценур и эхинококк. Цистицерк в виде небольшого округлого пузырька имеет одну впаянную внутрь головку будущего паразита (рис. 55). У лентецов встречается примитивная финна — плероцеркоид, имеющий лентовидную форму и одну ввернутую головку с ботриями.

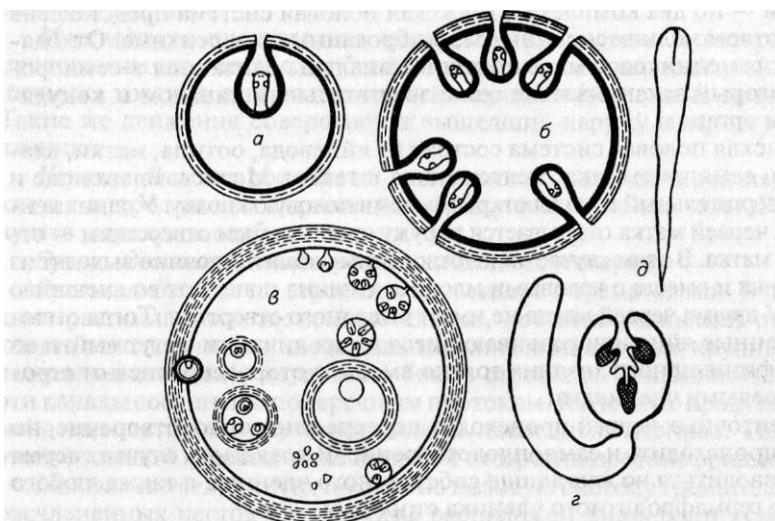


Рис. 55. Различные формы финн ленточных червей:
а — цистицерк; б — ценур; в — эхинококк; г — цистицеркоид; д — плероцеркоид

Ценур размером с крупный орех внутри имеет до нескольких сотен головок. Эхинококк отличается от других финн крупными размерами и сложным строением: под оболочкой пузыря имеется много головок, внутри пузыря находятся дочерние пузыри, дающие внучатые пузыри с несколькими головками. Таким образом, один эхинококк способен дать множество молодых паразитов.

Окончательный хозяин заражается, поедая органы промежуточного хозяина с финнами. В кишечнике окончательного хозяина под действием кишечного сока головки финн выворачиваются, черви закрепляются на стенке кишки и начинают продуцировать молодые членики.

Ленточные черви делятся на два подкласса, из которых один представлен подавляющим большинством этих паразитов — подкласс Цестоды (Cestoda). Остановимся лишь на представителях двух отрядов — I (епней и Лентецов, представляющих наибольшую опасность для животноводства. Заболевания, вызываемые цестодами, называются цестодозами.

Отряд Цепни (Cyclophyllidea) представлен ленточными червями, на скелете которых имеется четыре присоски, а у многих видов дополнительно есть венчики крючьев. Матки закрытого типа. Онкосфераe развиваются не во внешней среде (лентецы), а в матках зрелых члеников. Половая клоака находится сбоку членика.

Невооруженный (бычий) цепень (*Taeniarhynchus saginatus*) достигает в длину 8—12 м. На головке расположены только четыре присоски, а хоботка с крючьями нет. Паразитирует только в кишечнике человека. К матке зрелого членика может быть до 100 тыс. яиц; матка сильно разветвлена — до 35 ответвлений с каждой стороны (рис. 56).

Вышедшие с калом человека зрелые членики могут передвигаться. Промежуточный хозяин — крупный рогатый скот, заражается, проглатывая яйца с кормом и водой. В кишечнике скота из яиц выходят онкосфераe, которые вбуравливаются в стенку кишечника и проникают в кровь. Онкосфераe оседают в мышцах внутренних органов, где образуются финны типа цистицерк. Человек может заразиться, потребляя плохо проваренное или недожаренное мясо пораженного скота.

Заболевший человек худеет, происходит интоксикация организма продуктами выделения цепня. Бычий цепень живет до 18 лет, производя за этот период до 11 млрд яиц.

Вооруженный (свиной) цепень (*Taenia solium*), или свиной солитер, немного уступает бычьему по своим размерам — 2—4 м. Дефинитивным хозяином является человек. На головке помимо четырех присосок имеется хоботок с венчиком острых хитиновых крючьев. Матка зрелого членика имеет всего 8—12 ответвлений с каждой стороны, что определяет меньшее число зрелых яиц в ней — не более 50 тыс. Цикл развития > I ого цепня сходен с циклом развития бычьего цепня. Однако свиной солитер для человека более опасен, так как его труднее изгнать из кишечника (он прочно прикреплен к стенке кишки), а главное, человек может быть и промежуточным хозяином. Финны солитера развиваются

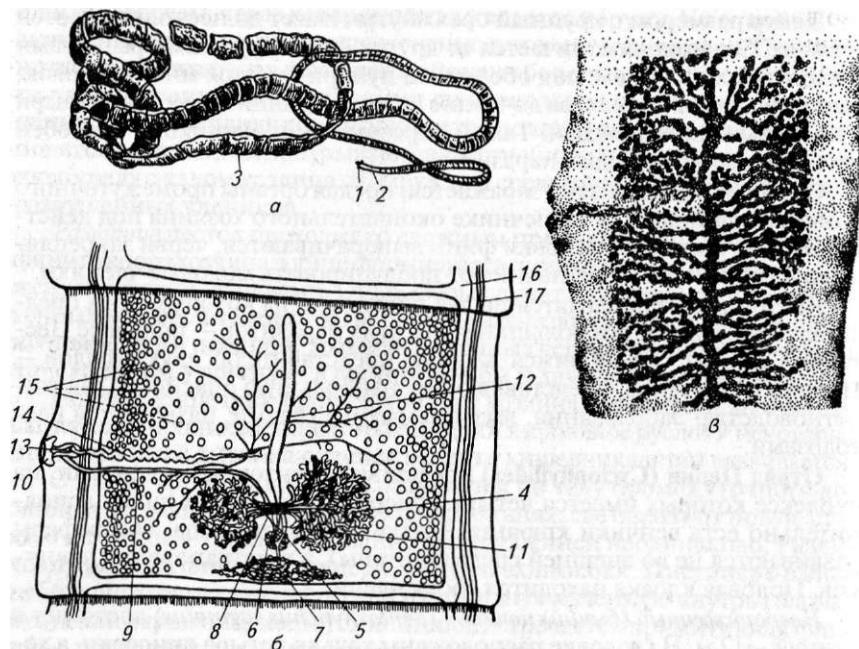


Рис. 56. Цепень невооруженный:
 а — внешний вид; б — гермафродитный членник; в — зрелый членник; 1 — головка;
 2 — шейка; 3 — стробила; 4 — двураздельный яичник; 5 — яйцевод; 6 — тельце Мелиса;
 7 — желточник; 8 — семяприемник; 9 — влагалище; 10 — половая клоака; 11 — устье матки;
 12 — матка; 13 — копулятивный орган; 14 — семяпровод; 15 — семенники; 16 — канал выделительной системы; 17 — нервный тяж

ся в различных внутренних органах, в том числе в печени, сердце, мозге, что может привести к смерти.

Зрелые членники из тела человека могут выходить целыми обрывками стробилы. Членники передвигаться по субстрату не могут. Промежуточным хозяином может быть свинья, кабан, собаки, кошки, кролики, зайцы, медведь, верблюд, иногда и человек. Финны типа цистицерк концентрируются в основном в мышцах, но могут оседать в сердце, печени, мозге и глазах, длительное время (до 6 лет) сохраняя жизнеспособность. Заражение происходит при поедании непрожаренного и непроваренного мяса, чаще всего свиного. Человек может быть промежуточным хозяином, если в его кишечник попадут зрелые яйца. В этом случае финны образуются в мышцах человека, вызывая тяжелое заболевание. Взрослые черви живут в кишечнике человека несколько лет.

Овечий мозговик (Multiceps multiceps) — червь небольших размеров; длина его доходит до 80 см. Головка кроме четырех присосок вооружена хоботком с двумя рядами крючьев. Дефинитивным хозяином является собака и ее дикие родичи. Зрелые членники с калом собак попадают

мо внешнюю среду. Если яйца будут проглочены промежуточным хо-
| ином (овца, коза, а также крупный рогатый скот, реже свиньи, верб-
люды и другие животные, очень редко человек), то из яиц выходят он-
косфера, которые внедряются в стенки кишечника и с током крови
разносятся по организму овцы. В головном мозге животного онкосфе-
ра превращается в центр, достигающий размеров куриного яйца. По-
раженная финной овца совершает круговые движения, так как обычно
Юражается одна половина мозга, что и определило название болезни —
нертнячка овец. Среди больных овец наблюдается массовая гибель. Соба-
ки и заражаются, поедая мозг погибших от вертнячки овец. Взрослые черви
живут до 6—8 мес.

Мониезии (различные виды рода *Moniezia*) достигают 5 м в длину и
более. Головка червя имеет только четыре присоски. Особенностью
мониезии является то, что в каждом гермафродитном членике находит-
ся двойной комплект половых органов, а половые отверстия расположены
по обеим сторонам членника. Дефинитивным хозяином паразита
является мелкий и крупный рогатый скот. Особенно тяжело переносят
заболевание молодые животные.

Промежуточными хозяевами служат некровососущие микроскопиче-
ские малые панцирные клещи, населяющие почву. Клещи поедают
онкосферы, выпавшие из разрушившихся членников паразита. Онко-
сфера через стенки кишечника клещей проникают в полость тела и там
превращаются в мелкие финны типа цистицеркоид (мельчайшая личинка
с одной головкой). Млекопитающие заражаются, поедая с травой
пораженных клещей. В борьбе с мониезией важен режим чередова-
ния выпаса скота на пастбищах.

Карликовый цепень (*Nyumentolepis nana*) соответствует своему названию:
его длина около 1 см. Развитие червя происходит в одном хозяине — че-
ловеке. В кишечнике человека может паразитировать до тысячи парази-
тов одновременно. Из онкосфер, попавших в кишечник человека, в
ворсинках развиваются мелкие финны — цистицеркоиды, которые за-
тем выпадают в просвет кишечника, закрепляются с помощью присо-
сок и крючьев на стенках кишки и превращаются в половозрелых чер-
вей. У человека часто наблюдается самозаражение. Весь цикл развития
цепня занимает всего 20 сут.

Эхинококк (*Echinococcus granulosus*) достигает в длину около 5 мм
(рис. 57). Головка этого паразита имеет четыре присоски и хоботок с
двумя рядами крючьев. Для эхинококка характерно наличие всего
трех-четырех членников: незрелый, гермафродитный и зрелый. Зрелый
членник, содержащий до 800 яиц, отрывается от тела паразита и выносится
с каловыми массами дефинитивного хозяина (собаки, волка, шакала,
лисицы, а также других хищных животных) во внешнюю среду. Зрелые
членники во внешней среде некоторое время могут передвигаться, в том
числе и в шерсти хозяина. Место оторвавшегося зрелого членника после
оплодотворения занимает гермафродитный членник, становясь зрелым.

Промежуточным хозяином могут стать мелкий и крупный рогатый
скот, свиньи, лошади, кролики, грызуны и другие млекопитающие,

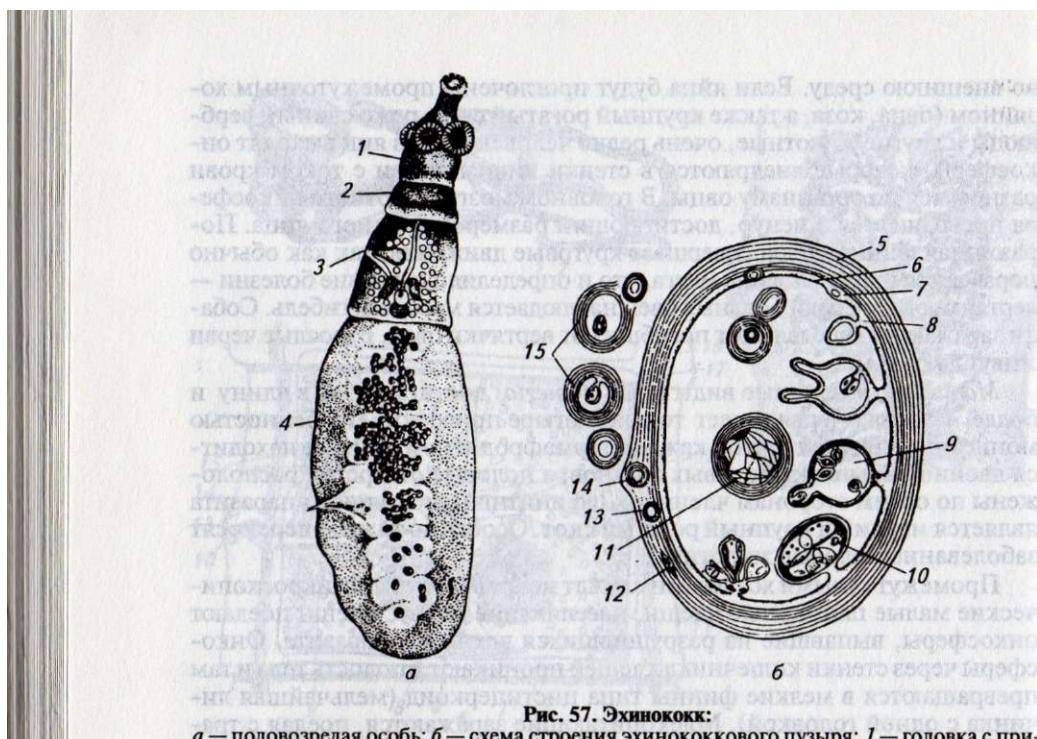


Рис. 57. Эхинококк:
а — половозрелая особь; б — схема строения эхинококкового пузыря; 1 — головка с присосками и хоботком, вооруженным крючьями; 2 — незрелый членик; 3 — гермафродитный членик; 4 — зрелый членик; 5 — кутикула; 6 — производящая оболочка; 7, 8, 9, 10, // — внутренние дочерние пузыри на разных стадиях развития; 12, 13, 14, 15 — наружные дочерние пузыри на разных стадиях развития

кроме семейства собачьих. Может им стать и человек. Заболевание называется эхинококкозом, часты летальные исходы.

В кишечнике промежуточного хозяина из яйца выходит онкосфера. Попадая через стенки кишечника в кровяное русло, онкосфера мигрирует по всему организму, оседая и образуя финны типа эхинококк чаще всего в печени, реже в легких, мышцах и мозге. У крупного рогатого скота масса финны может достигать более 50 кг. Внутри финны много дочерних пузырей, что можно рассматривать как бесполое размножение паразита на ранних стадиях развития.

Источником заражения человека могут стать собаки, особенно пастушки или при свободном содержании в поселках и городах. Выползающие из анального отверстия зрелые членики вызывают у собак зуд, животные чешутся и разносят яйца по шерсти. Взрослые черви живут в кишечнике собак до 6 мес. Эхинококк распространен в местностях с развитым животноводством.

В последние годы участились случаи заражения человека альвеококком (*Alveococcus multilocularis*) в Европейских странах. Дефинитивным хозяином этого мелкого цепня, похожего на эхинококка, обычно являются

лисицы, собаки и кошки. Промежуточным хозяином могут быть мышевидные грызуны, которые заражаются яйцами альвеококка, подбирая остатки у нор лис и жилищ человека, где могут встречаться зараженные кошки и собаки. Человек также может стать промежуточным хозяином, заражаясь яйцами цепня от собак и кошек. Чаще всего финны образуются у человека в дыхательных путях, что может вызывать удушье.

Отряд лентецы (Pseudophyllidea). У представителей этого отряда головка не имеет присосок и крючьев. Органами прикрепления служат ботрии — щелевидные ямки, с помощью которых паразиты защемляют стенку кишечника. Матка у лентецов открывается наружу на брюшной стороне членика отверстием, через которое зрелые яйца могут выходить в просвет кишечника дефинитивного хозяина. Отряд включает много паразитических видов, из которых наиболее опасен широкий лентец (*Diphyllobothrium latum*). Червь живет в кишечнике хищных животных, достигая длины 8—10 м. Дефинитивным хозяином может быть человек, заражаются также дельфины и тюлени. Вышедшие с калом во внешнюю среду зрелые яйца с крышечкой должны попасть в пресную воду. Из яиц в воде выходит личинка, покрытая ресничками, — корацидий (рис. 58). Личинку могут съесть веслоногие ракообразные —

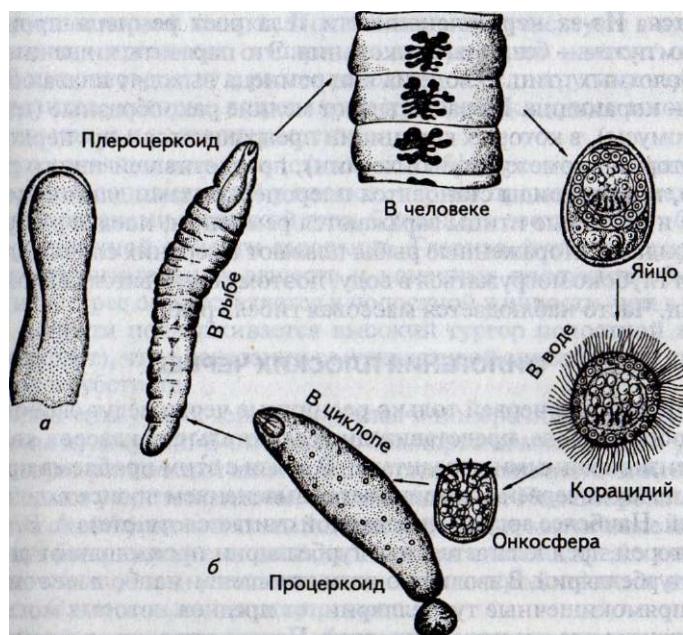


Рис. 58. Лентец широкий *Diphyllobothrium latum*:
а — головка; б — цикл развития

цикlopы и диаптомусы, в кишечнике которых из корацидия выходит сформировавшаяся личинка онкосфера с шестью крючьями. Онкосфера внедряется в полость тела первого промежуточного хозяина и там превращается в покоящуюся фазу — процеркоид, имеющий удлиненную форму с диском на заднем конце тела, несущем крючья.

Если пораженного ракообразного проглотит рыба (щука и другие), то в теле рыбы процеркоид внедряется во внутренние органы и мышцы, где превращается в финнозную стадию — плероцеркоид. Плероцеркоиды имеют червеобразное тело длиной 1—2 см с одной ввернутой головкой на переднем конце тела. Головка вооружена двумя ботриями.

Дефинитивный хозяин заражается, поедая сырую, замороженную или слабо просоленную рыбу. Человек заболевает дифиллоботриозом чаще всего там, где потребляют в больших количествах сырую рыбу, в основном в виде строганины из замороженной рыбы, в которой остаются жизнеспособные плероцеркоиды. В цикле развития лентеца важны резервуарные хозяева (хищные рыбы). При поедании щуками пораженных плероцеркоидами рыб в кишечнике щук паразиты не перевариваются, а проникают в их ткани и накапливаются в различных органах.

К лентециам относится червь, имеющий нерасчлененное тело, — *ремнец* (сем. Ligulidae). Длина паразита колеблется от 7 до 200 см. Тело в виде ленты, в которой комплекты половых органов многократно повторяются. Из-за нерасчлененности тела рост ремнца происходит обычным путем — без зоны почкования. Это паразиты кишечника водных и болотных птиц. В воде из яиц ремнца выходят плавающие личинки — корацидии. Их заглатывают мелкие ракообразные (цикlopы и диаптомусы), в которых корацидии превращаются в процеркоиды. В рыбе (второй промежуточный хозяин), проглатившей такого ракообразного, процеркоиды становятся плероцеркоидами длиной до 60 см. Водные и болотные птицы заражаются ремнцем, поедая рыбу с плероцеркоидами. Пораженные рыбы плавают в верхних слоях воды; они не могут глубоко погружаться в воду, поэтому становятся легкой добычей птиц. Часто наблюдается массовая гибель рыб.

ФИЛОГЕНИЯ ПЛОСКИХ ЧЕРВЕЙ

Среди плоских червей только ресничные черви ведут свободноживущий образ жизни, представители всех остальных классов являются специализированными паразитами. В связи с этим проблема происхождения плоских червей ограничивается выяснением происхождения турбеллярий. Наиболее аргументированной считается гипотеза А. В. Иванова, в которой предполагается, что турбеллярии произошли от ацелоподобных турбеллярий. В эволюционном отношении наибольшее значение имеют прямокишечные турбеллярии, от предков которых могли произойти другие классы плоских червей. Переход плоских червей к паразитизму мог осуществляться через симбиоз, тем более что такие тенденции проявляются и у современных турбеллярий.

Моногеней могли произойти от турбелляриеподобных предков через квартирантство на жабрах и плавниках рыб. Затем постепенно они перешли к эктопаразитизму. Среди современных моногеней наблюдается переход к эндопаразитизму, например у лягушачьей многоустки. Родственные связи моногеней и цестод были доказаны русским ученым И. Е. Быховским. Эволюция же трематод могла идти независимо от моногеней и цестод. Можно предположить, что на первом этапе эволюции трематоды вели свободный образ жизни, а к паразитизму перешли их личинки, вступившие в симбиотические отношения с моллюсками.

ТИП КРУГЛЫЕ, ИЛИ ПЕРВИЧНО- ПОЛОСТНЫЕ ЧЕРВИ (*Nemathelminthes*)

Круглых червей часто называют первично-полостными червями, так как они имеют несегментированное тело с первичной полостью, заполненной полостной жидкостью. Кишечный канал не разветвлен и заканчивается анальным отверстием. Известно более 100 тыс. видов круглых червей, среди которых много свободноживущих форм, встречающихся в морях, пресных водоемах и в почве. Практическое значение круглых червей велико. Почвенные виды участвуют в почвообразовательных процессах, а паразитические черви наносят огромный ущерб животноводству и растениеводству. Среди круглых червей много паразитов, которые встречаются практически у всех многоклеточных животных и у многих растений.

Первичная полость тела (схицоцель) образуется за счет разрушения паренхимы, заполняющей у плоских червей промежутки между внутренними органами и стенкой тела. Схицоцель соответствует бластоцелю — первичной полости зародыша. Главная функция схицоцеля — транспорт питательных веществ и конечных продуктов обмена, что легче и быстрее осуществляется в полостной жидкости, чем в паренхиме. В полости поддерживается высокий тургор полостной жидкости (гидроскелет), что в сочетании с мускулатурой способствует движению нематод в субстрате.

Форма тела у этих червей округлая в поперечнике. Покровы представлены кутикулой. В отличие от брюхоресничных червей у нематод отсутствует ресничный эпителий. Остатки ресничного эпителия имеются только у представителей низших классов. Под покровами располагается слой продольных мышц или отдельные мышечные пучки, которые могут быть кольцевые мышцы. Нервная система представлена окончательным узлом и отходящими от него продольными нервыми тяжами. Органы дыхания отсутствуют.

Кишечник состоит из трех отделов: переднего, среднего и заднего. Переднее отверстие находится на брюшной поверхности переднего конца тела. Имеется анальное отверстие.

Выделительная система построена по протонефридиальному типу, но без мерцательных клеток, или в виде особых кожных (гиподермальных) желез.

Большая часть круглых червей раздельнополые, но встречаются и гермафродитные формы. Часто выражен половой диморфизм. Размножаются толькоовым путем. Развитие прямое, реже — с метаморфозом. Круглые черви не способны к регенерации.

Тип Круглые черви включает несколько классов, из которых наибольший интерес представляют три класса: класс Собственно круглые черви, или Нематоды (*Nematoda*), класс Скребни (*Acanthocephala*) и класс Коловратки (*Rotatoria*). Для сельского хозяйства наибольшее значение имеют представители класса нематод.

КЛАСС СОБСТВЕННО КРУГЛЫЕ ЧЕРВИ, ИЛИ НЕМАТОДЫ (*Nematoda*)

Общая характеристика. Округлое тело длиной от нескольких миллиметров до 1 м (паразитическая нематода кашалота достигает в длину 8 м) сохраняет постоянную ширину и покрыто плотной кутикулой. Свободноживущие виды населяют соленые и пресные воды, живут в почве. Много паразитов растений, животных и человека. Только паразитов растений описано более 1 тыс. видов. Среди паразитических форм большой интерес представляют те виды, которые паразитируют на вредных насекомых и сорных растениях.

Строение и жизненные отправления. Несмотря на большое экологическое разнообразие, нематоды однообразны морфологически. Форма тела обычно веретенообразная, нитевидная и реже колбасовидная. Свободноживущие черви живут в илистом грунте водоемов или гумусовом горизонте почвы, питаясь органикой. Паразитические формы, обитаю в тканях растений и в теле животных, поглощают органические вещества из организма хозяина. Есть виды с округлой формой тела. На переднем конце расположены рот, органы осязания и химического чувства, светочувствительные органы встречаются редко. Туловище заканчивается хвостовым отделом, следующим за анальным отверстием. Тело у самцов может оканчиваться бурсой — органом прикрепления к телу самки при спаривании. Половой диморфизм часто хорошо выражен.

Покровы нематод образованы гиподермой, покрытой кутикулой. Кутикула может состоять из четырех—десяти слоев. Ее поверхность кольчатая или гладкая. Химический состав кутикулы представлен сложном комплексом белков, липопротеинов и других веществ. Кутикула благодаря своему составу находится в биологически активном состоянии и устойчива к действию пищеварительных ферментов хозяина, хотя у погибших червей кутикула легко переваривается в кишечнике животных.

Вещества, образующие кутикулу, выделяются клетками гиподермы. У свободноживущих червей гиподерма представлена однослойным эпителием, а у паразитических взрослых червей — протоплазматической массой, содержащей многочисленные ядра. На спинной и брюш-

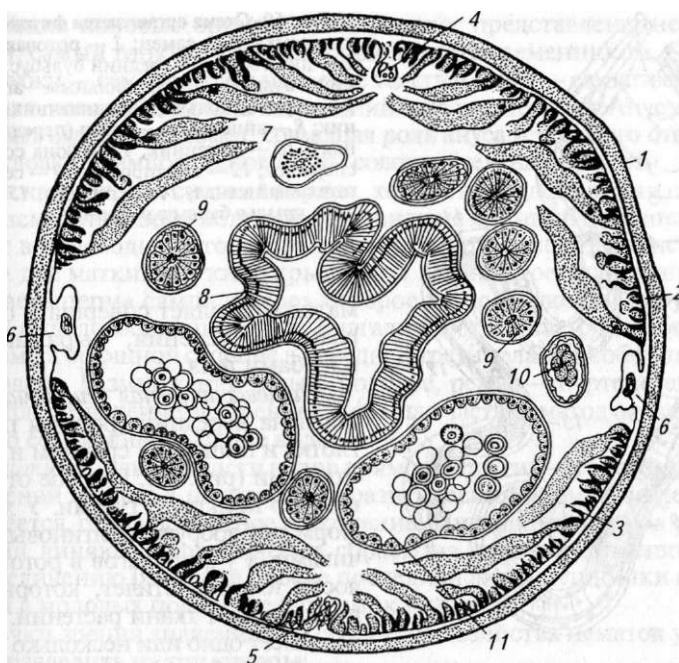


Рис. 59. Поперечный разрез тела самки аскариды *Ascaris lumbricoides*:
1 — кутикула; 2 — гиподерма; 3 — мышцы; 4 — спинной валик гиподермы с нервным ганглием; 5 — брюшной нервный тяж; 6 — боковые валики гиподермы с каналами выделительной системы; 7 — первичная полость тела; 8 — кишечник; 9 — яичники; 10 — яйцевод; 11 — матка

ной сторонах, а также по бокам гиподерма образует валикообразные утолщения, тянувшиеся вдоль тела.

Первичная полость тела заполнена полостной жидкостью и заключена в кожно-мускульный мешок. В полости тела расположены внутренние органы,

Нервная система в виде окологлоточного нервного кольца, которое опоясывает пищевод, и отходящих от него двух продольных нервных тяжей: спинного и брюшного. Образована нервная система небольшим числом нервных клеток. Большее развитие получили брюшные и спинные нервные тяжи, соединенные комиссурами и дающие ответвления к различным органам (рис. 59). Органы чувств развиты слабо и представлены осязательными и обонятельными клетками.

Мускулатура образована обычно полосами продольных мышечных волокон, которые разделены по бокам тела выростами (валиками) гиподермы, а на спинной и брюшной сторонах — нервными стволами, нежащими в валиках гиподермы. Мышечные полосы образованы слоем удлиненных клеток, содержащих миофibrиллы. Мускулатура не-

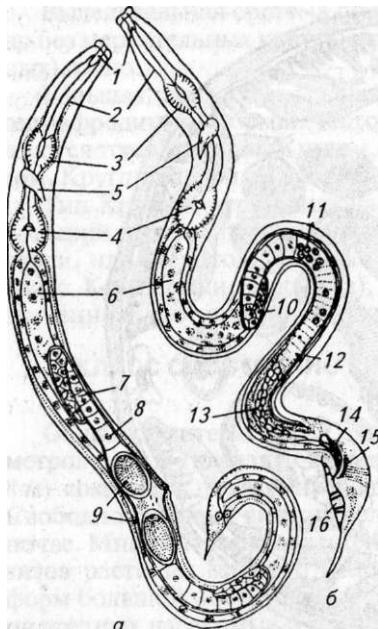


Рис. 60. Схема организации фитонематод:
а — самка; б — самец; 1 — ротовая полость;
2 — пищевод; 3 — средний бульбус; 4 — зад-
ний бульбус с дробильным аппаратом;
5 — нервное кольцо; 6 — кишечник; 7 — яич-
ник; 8 — яйцевод; 9 — матки (передняя и зад-
няя); 10 — семенник; 11 — зона созревания
спермиев; 12 — семяпровод; 13 — семязвер-
гательный канал; 14 — спикулы; 15 — рулек;
16 — крылья бурсы с ребрами

матод позволяет совершать однообразные движения, выражющиеся изгибами тела.

Пищеварительная система представлена передним (ротовая полость, глотка и пищевод), средним и задним отделами (рис. 60). Ротовое отверстие обычно прикрыто губами. У хищных форм рот вооружен хитиновыми зубчиками, а у фитофагов в ротовой полости имеется стилет, которым они прокалывают ткани растений. Пищевод имеет одно или несколько расширений (бульбусов) с мощной мускулатурой. Передний отдел кишечника выстлан эпителием.

Процессы переваривания пищи происходят в средней части кишки, выстланной однослойным эпителием эндоцермального происхождения. Задняя кишка выстлана кутикулой и заканчивается анальным отверстием.

Органы дыхания. У свободноживущих нематод и фитопаразитов газообмен происходит через покровы тела; у большинства паразитов животных и человека анаэробное дыхание.

Органы выделения. В головном отделе нематод расположены одна-две гигантские клетки гиподермы, называемые шейными железами, которые выполняют функции осморегуляции и выделения. Железа имеет отростки и канал внутри, который открывается наружу. У многих нематод шейная железа представляет собой большую клетку с двумя отростками, которые тянутся вдоль тела в боковых валиках гиподермы и имеют внутри каналы. Эти каналы в шейной железе сливаются и открываются наружу порой на брюшной стороне в передней части нематоды. Функцию почек накопления выполняют одна-две пары фагоцитарных клеток, лежащих около выделительных каналов.

Органы размножения. Нематоды раздельнополы. У них обычно развит половой диморфизм. У паразитических нематод самки крупнее самцов, задний конец тела которых закручен. У части фитонематод самки при созревании яиц сильно раздуваются, приобретая округлую форму. Половые органы имеют трубчатое строение.

Мужские половые органы, как правило, представлены непарной трубкой, самый тонкий конец которой является семенником. Средняя часть трубы — семяпровод, а наиболее толстый отдел — семязвергательный канал, который открывается в задний отдел кишечника. Анус у самцов выполняет функцию клоаки, совмещающую роль ануса и полового отверстия. (около клоаки у самцов расположены совокупительные спикулы.

Женская половая система парная, хотя есть представители с непарной женской половой системой. Нитевидные яичники постепенно переходят в яйцеводы, которые расширяются и переходят в толстые каналы — две матки. Матки открываются в непарное влагалище, куда поступает сперма самца и через которое в последующем наружу выводятся оплодотворенные яйца. Влагалище открывается половым отверстием на брюшной стороне в передней трети тела на особом перехвате — пояске. Размножение только половое, редко — партеногенетическое. Оплодотворение внутреннее. Большая часть нематод откладывают яйца, но есть и живородящие виды.

Развитие у большей части видов прямое, у других — с метаморфозом и со сменой хозяев. У некоторых паразитических форм в цикле развития имеется гермафродитное поколение. Личинки в период роста и развития линяют, периодически сбрасывая кутикулу, препятствующую увеличению размеров. После последней линьки личинки превращаются в молодых половозрелых самок и самцов.

С точки зрения значения для сельского хозяйства нематод условно можно разделить на три группы:

- свободноживущие в водоемах и почвах сапрофитные и хищные формы, питающиеся органическими остатками и представителями мелкой почвенной фауны;
- паразиты растений, обитающие в тканях различных растений в течение всей жизни или на определенной стадии своего развития;
- ® паразиты животных и человека.

Паразитических нематод условно можно разделить на две экологические группы: геогельминты, часть жизненного цикла которых проходит во внешней среде, и биогельминты, развитие которых проходит только в одном или нескольких хозяевах без выхода во внешнюю среду.

Свободноживущие круглые черви. В соленных и пресных водах, а также в почве обитает множество свободноживущих мелких круглых червей. Водные нематоды являются важным звеном в цепях питания юных животных. Для сельского хозяйства большой интерес представляют нематоды, населяющие почву. Они предпочитают хорошо увлажненные и богатые органикой почвы. В благоприятных условиях обитания в 1 м² верхнего почвенного слоя можно насчитать десятки миллионов этих червей. Основная их часть — постоянные обитатели почв, но значительное их число находится в почве только на определенной стадии своего развития (жизненного цикла).

Почвенные нематоды питаются гниющими остатками растений и других органических веществ. Есть и хищные нематоды, поедающие других мелких почвенных животных. Все почвенные нематоды явля-

ются участниками почвообразовательного процесса. Перерабатывая органику, нематоды способствуют гумификации почв. Зная биологические особенности почвенных нематод и изучив закономерности их жизненных циклов, полеводы могут создавать благоприятные условия для их жизнедеятельности, используя соответствующую технику, систему обработки почв, их мелиорации, внесения органических и минеральных удобрений и т. п. Зная особенности жизнедеятельности хищных нематод, можно попытаться использовать их для борьбы с вредными животными, в частности с фитонематодами.

Круглые черви — паразиты растений (фитонематоды). Для круглых червей, паразитирующих на растениях (фитонематоды), характерны небольшие размеры тела (длина 0,1—12 мм, ширина около 15—20 мкм), имеющего цилиндрическую, нитевидную или веретенообразную форму. Часто самки уплощены и по форме напоминают мешок или лимон. Тело состоит из трех отделов: головного, собственно тела и хвостового. В ротовой полости нематод имеется колющий орган — стилет (копье), с его помощью нематода прокалывает ткани растений. В стилете находится канал, через который нематода впрыскивает секрет желез пищевода в растение. Общим для большинства фитонематод является частично внекишечное пищеварение. В секрете желез пищевода содержатся пищеварительные ферменты, которые вызывают гидролиз высокомолекулярных соединений. Приготовленная полупереваренная питательная смесь засасывается с помощью стилета в кишечник фитонематоды, где заканчивается процесс переваривания и всасывания пищи. В средней части пищевода имеется расширение — бульбус, с его помощью паразит всасывает соки растений (рис. 61).

При движении фитонематоды совершают медленные волнообразные движения. Есть и неподвижные формы; это — самки, тип питания которых не требует перемещения. У таких нематод мускулатура полностью редуцирована, подвижна лишь передняя часть тела. Нематоды могут передвигаться на небольшие расстояния — в пределах 30 см, но



0
δ

Рис. 61. Формы стилета и копья фитонематод:
 $a - 2$ — стилеты; $\delta - 3$ — копья

существуют формы и более подвижные, передвигающиеся на расстояния до 100 см.

Фитонематоды раздельнополы, для них характерен половой диморфизм. Особенно сильно половой диморфизм выражен у видов, имеющих раздутых сидячих самок, характеризующихся мешкообразной формой тела. Самцы же имеют типичную для нематод нитевидную форму и способны передвигаться.

Размножаются фитонематоды половым путем, очень редко встречается гермафроптизм и партеногенез. Размножение всегда происходит яйцами, есть виды, у которых наблюдается живорождение — личинки выходят из яиц еще в яичнике самки. Оплодотворение внутреннее. У самца имеются специальные выросты дна клоаки — спикулы. Их назначение — расширение вульвы при совокуплении. Спермин фитонематод совершают амебоидное движение, так как они лишены хвоста.

Плодовитость фитонематод высока; одна самка может отложить несколько сотен яиц, но у отдельных видов это число доходит до нескольких тысяч. Следует различать два показателя плодовитости: число яиц, отложенных самкой за всю ее жизнь, и число яиц, развивающихся в матке самки одновременно.

Нематоды откладывают яйца в соответствии со своим образом жизни: в ткань растений, в яйцевой мешок на субстрате, внутрь галла (вздутия в месте повреждения паразитом), яйца могут оставаться в теле самки; в этом последнем случае тело ее представляет собой нечто вроде цисты.

Механизм выхода личинки из яйца весьма сложен и до конца не ясен. Есть виды, у которых выход личинок стимулируется выделениями корней растений-хозяев. У других представителей выход личинок зависит от влажности, температуры, содержания кислорода и т. п. Перед выходом личинка в яйце совершает быстрые движения, разрывающие оболочку яйца. Развитие личинок включает четыре личиночных возраста и взрослую форму. Все эти стадии четко разграничены линьками.

Личинки первого возраста у ряда видов проходят линьку еще в яйце. Такой тип первой линьки называют закрытым (денударным), и он характерен для фитонематод. Сапробиотические нематоды первую линьку проходят уже вне яйца — открытый тип (конвелярный).

Личинки первого возраста отличаются от взрослых форм (имаго) в основном значительно меньшими размерами, а также недоразвитыми половой и пищеварительной системами. Зачаточная половая система появляется лишь у личинок второго возраста, и затем по мере роста и развития личинок она достигает своего окончательного развития.

Перед началом линьки личинки перестают питаться. В процессе линьки вся кутикула сбрасывается и одновременно формируется новая. В период развития у нематод может меняться соотношение полов: при неблагоприятных условиях внешней среды (нетипичное растение-хозяин, засуха, нематодоустойчивые сорта) развивается больше самцов. Число генераций у нематод, имеющих несколько поколений, также зависит от условий среды, и прежде всего от температуры и влажности.

Онтогенез фитонематод обусловлен особенностями их связи с растением-хозяином. Одна группа фитонематод-эктопаразитов питается, проникая стилетом в растение, но весь жизненный цикл их проходит в почве, где они мигрируют (лонгидоры, триходоры и др.).

У группы нематод—полуэндопаразитов корней (спиральные нематоды) часть жизненного цикла проходит в почве, где они мигрируют, а часть — в корнях растений, куда они внедряются передним концом тела.

Эндопаразитические формы фитонематод развиваются в корнях растений, но могут свободно мигрировать из корней в почву и обратно, т. е. почва для этих нематод является средой переживания (пратиленхи).

Онтогенез нематод—эндопаразитов корней (тиленхиды, гетеродериды и др.) проходит только в тканях растений без выхода паразитов в почву. Растения для этих фитонематод являются постоянной средой обитания. Нематод, поражающих надземные части растений, по особенностям онтогенеза также можно условно разделить на три группы. Есть эктоэндопаразиты, которые мигрируют на растении, почва для этих нематод не нужна. У второй группы эндопаразитических нематод (стеблевые нематоды), которые мигрируют только в тканях растений, а также группы галлообразователей (ангвины), онтогенез проходит в одном локальном месте растения. Почва для этих двух последних групп служит средой переживания.

Систематика вредоносных фитонематод базируется на различиях в их морфологии, биологии и на особенностях их экологии:

Класс Круглые черви (Nematoda)

Подкласс Сецерненты, или Фазмидиевые (Secernentea)

Отряд Тиленхиды, или Настоящие шишкоиглые нематоды (Tylenchida)

Отряд Афеленхиды (Aphelenchida)

Подкласс Аденофореи (Adenophorea)

Отряд Дорилаймиды (Dorylaimida)

Небольшое число фитонематод относятся к отряду Афеленхиды. Эти нематоды повреждают надземные части растений (завязи, почки, листья) и могут быть как экто-, так и эндопаразитами. *Рисовый афеленхойд* (*Aphelenchoides besseyi*) является полифагом, но наибольший вред приносит рисовым и земляничным посадкам. Нематоды разного возраста сохраняются под пленкой рисовой зерновки и в послеуборочных остатках. Весной нематоды выходят в почву и отыскивают всходы риса. *Земляничная нематода* в симбиозе с бактерией *Corynebacterium fascians* вызывает заболевание земляники «цветная капуста».

Хризантемная нематода (*Aphelenchoides ritzemabosi*) является эктопаразитом хризантем, земляники, крыжовника, смородины, томата, ма-лины и других растений. Паразиты зимуют на стеблях или в почве.

В отряде Дорилаймиды подкласса Аденофореи имеется несколько видов нематод, являющихся паразитами растений. Крупные нематоды из семейства Лонгидориды повреждают многие культуры: овощные, свеклу, табак, зерновые, бобовые травы, землянику, древесные и т. д.

Основная масса червей, паразитирующих в растениях (фитонематод) относится к отряду 111ишкоиглые нематоды подкласса Сецирненты. В этот подкласс входят также нематоды почвенные, сапробиотические, хищные, паразиты насекомых.

Нематод из семейства Гетеродериды часто называют разнокожими нематодами, так как самки гетеродерид имеют шаровидное тело с плотными, часто окрашенными покровами, а самцы — вытянутое гонкое тело с прозрачными покровами. У нематод сильно развит стилет. У самок парные яичники, у самцов один или два семенника.

Подсемейство Гетеродерины относится к группе цистообразующих нематод, у которых в жизненном цикле обязательно наличие особых образований из отмершего тела самки — цист. В цистах яйца и инвазионные личинки в течение нескольких лет сохраняют жизнеспособность даже при неблагоприятных условиях.

Биологический цикл у всех видов цистообразующих нематод одинаков (рис. 62). Из перезимовавших в почве цист весной выходят личинки второго возраста. Они поражают корни растений, которые часто находят посредством хемотаксиса на выделения корней растения-хозяина.

В корнях личинки становятся неподвижными, усиленно питаются, линяют и превращаются в бутылковидных личинок третьего возраста. После следующей линьки они превращаются в сильно утолщенных личинок четвертого возраста, принимающих шаро-, лимоно-, груше- или мешкообразную форму. У них подвижен только головной отдел. Из личинок четвертого возраста развиваются самки и самцы в соотношении 1:1. При неблагоприятных условиях развития доля самцов возрастает.

Самцы в оболочке личинки находятся в свернутом состоянии. После разрыва шкурки самцы выходят через разрыв корня растения в почву. Тело самцов тонкое и прозрачное, что не позволяет обнаружить их невооруженным глазом.

Самки также разрывают ткани коры корня растения, и в этом разрыве находится задний конец их тела. Головной же отдел погружается в ткань корня. Таким образом, самки как бы прикреплены к корням, и их крупное (до 1 x 0,5 см) белое тело легко обнаружить невооруженным глазом. Самцы находят самок, оплодотворяют их и после этого погибают, хотя у некоторых видов один самец может оплодотворить нескольких самок. Самки некоторых видов вырабатывают половые аттрактанты, играющие важную роль в привлечении самцов.

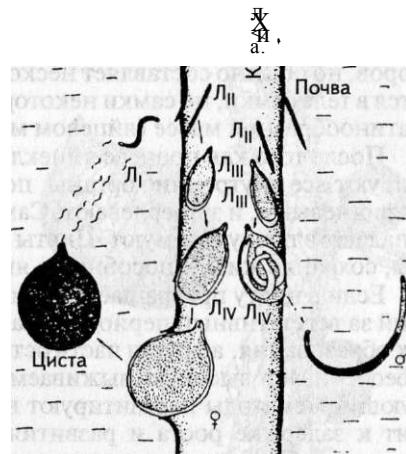


Рис. 62. Цикл развития цистообразующих нематод: L₁—L₄ — личиночные стадии

Оплодотворенные самки питаются и начинают откладывать яйца, число которых зависит от наличия пищи и других абиотических факторов, но обычно составляет несколько сотен. Большая часть яиц остается в теле самки, но самки некоторых видов откладывают наружу в желатинообразной массе (яйцевом мешке) по нескольку яиц.

После того как процесс яйцекладки завершен, в теле самки регенерируют все внутренние органы, покровы тела меняют цвет, становясь коричневыми, и затвердевают. Самки превращаются в цисты, которые опадают в почву и зимуют. Цисты могут находиться в почве долгие годы, сохраняя жизнеспособность яиц и личинок.

Если в цисту превращается самка вида, дающего несколько генераций за вегетативный период, то часть личинок выходит сразу же после их образования, а другая часть остается в цистах на несколько лет, что обеспечивает высокую выживаемость фитонематод. Все цистообразующие нематоды паразитируют внутри корней растений, что приводит к задержке роста и развития растений-хозяев. Степень ущерба культуре обусловлена численностью паразита.

Из цистообразующих фитонематод значительный вред приносят овсяная нематода (*Heterodera avenae*), люцерновая нематода (*H. medicaginis*), соевая нематода (*H. glycines*), свекловичная нематода (*H. schachtii*), картофельная нематода (*H. rostochiensis*) и некоторые другие.

Овсяная нематода (*H. avenae*) является представителем семейства разнокожих нематод (Heteroderidae). Это вредитель злаковых культур, прежде всего овса, а также пшеницы и ячменя. Известны случаи, когда растениями-хозяевами служили пырей, овсяница и дикий овес. Растения, пораженные овсянной нематодой, отстают в росте, их листья рано желтеют. Ущерб особенно велик в засушливые годы, так как гельминты задерживают поступление воды и питательных веществ в растение.

Как все представители цистообразующих нематод, овсяная нематода обладает четким половым диморфизмом. Лимоновидные самки покрыты беловатой слизистой оболочкой. Постепенно кутикула взрослой самки темнеет, и самка превращается в бурую цисту. Самцы имеют нитевидное прозрачное тело длиной 1,2—1,4 мм.

Для цикла развития овсянной нематоды характерно наличие только одной генерации в год. Весной из цист нематод, перезимовавших в почве, начинается выход личинок второго возраста. Они внедряются в корни проростков овса и начинают питаться. После линьки самцы приобретают нитевидную форму, а самки становятся лимоновидными.

Оплодотворенная самка начинает продуцировать яйца. Формирование цист происходит в почве на глубине 10—40 см. Цисты должны перезимовать, так как активация личинок в яйцах цисты становится возможной после прохождения периода низких температур.

Бороться с овсянной нематодой можно путем использования севооборотов с возвратом посевов зерновых злаков не ранее чем через 4 года, а также выведением устойчивых к нематодам сортов зерновых злаков.

Картофельная нематода (*H. rostochiensis*) паразитирует на корнях и клубнях картофеля. Является одним из самых опасных вредителей кар-

гофеля; объект карантинных мероприятий. Из европейской части РФ паразит постепенно расселяется на восток. Поражает растения только из семейства пасленовых — картофель, томат, баклажан. Растения сильно угнетены, листья рано желтеют и завядают, урожай может снижаться на 80 %.

У самца тонкое прозрачное тело длиной не более 1,2 мм. Самки шаровидной формы диаметром до 1 мм. По мере старения покровы самки окрашиваются в коричневый цвет, иногда становясь почти черными. Самки живут на корнях картофеля и выглядят мелкими желтовато-коричневыми шариками (рис. 63), у самок яйцевых мешков не образуется.

В одной цисте может быть до 1200 яиц. Личинки нематод выходят из цист и проникают в корни всходов картофеля. На корнях образуются вздутия, а через месяц растущие личинки разрывают кору корня, линяют последний раз и становятся взрослыми червями. Самцы выходят в почву для поиска самок и спаривания с ними. У оплодотворенных самок тело становится шаровидным, его полость заполняется яйцами.

Для выхода личинок из цист необходимы не только оптимальные температура и влажность, но и наличие растения-хозяина. Выделения корней растения-хозяина стимулируют выход личинок. Новая генерация личинок появляется через 40—75 сут, т. е. в течение лета возможно развитие одной, реже двух генераций фитогельминта.

Для борьбы с картофельной нематодой и профилактики заражения рекомендуется использовать противонематодный севооборот с возвращением восприимчивых культур не ранее чем через 3—4 года. Хорошие результаты дает выращивание нематодоустойчивых сортов картофеля.

Свекловичная нематода (H. schachtii) распространена повсюду, но особый вред приносит в южных регионах, где выращивают сахарную свеклу. Лимоновидное тело самки окрашено в темно-желтый или бурый цвет (рис. 64). Размеры тела самок не превышают 0,7—1 мм в длину и 0,4—0,5 мм ширину. Самец с нитевидным прозрачным телом диаметром 0,02—0,03 мм достигает длины 1,5 мм. Стилет мощный, семенник один.

Свекловичная нематода паразитирует на растениях семейств маревых и крестоцветных, но особый ущерб причиняет посевам сахарной

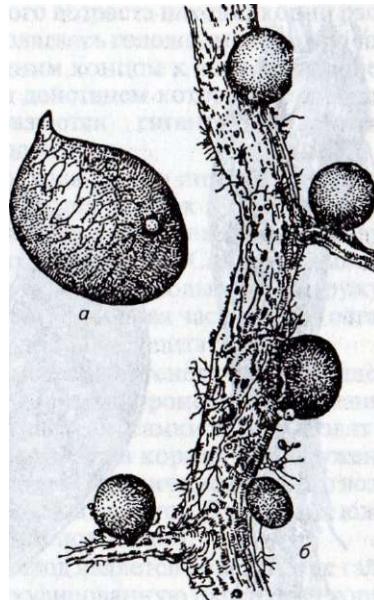


Рис. 63. Самки картофельной нематоды *Heterodera rostochiensis*:
а — циста; б — самки на корнях картофеля

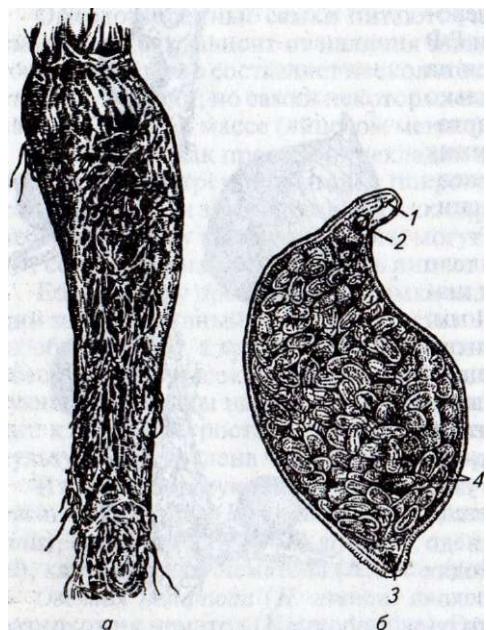


Рис. 64. Свекловичная нематода *Heterodera schachtii*:
а — корневая система свеклы, поврежденная нематодой; б — самка нематоды; 1 — стилет; 2 — средний бульбус; 3 — вульва; 4 — яйца

свеклы. Потери урожая могут достигать 60 % при одновременном снижении (до 15 %) сахаристости. Из-за недостаточного питания пораженные нематодой растения угнетены, их листья желтеют и засыхают. Посевы поражаются пятнами. На участках с большим количеством цист в почве может наблюдаться полная гибель растений.

Самки свекловичной нематоды в отличие от овсяной и картофельной могут откладывать часть яиц в яйцевые мешки и при благоприятных

условиях давать несколько генераций в течение вегетации свеклы. Для развития одного поколения требуется около 4—5 нед. Развитие одной генерации свекловичной нематоды при 18 °C занимает 57 сут, а при 28 °C — 24 сут. По этой причине в Московской области две генерации, а в Винницкой — три-четыре. За свою жизнь самка формирует 100—150 яиц, но иногда и больше. Личинки, которые находятся в яйцах, заключенных в цисты, могут сохранять жизнеспособность до 9 лет.

Выход личинок из цист начинается при температуре 10 °C и выше, но оптимумом считается 18—28 °C. Радиус расселения личинок из цисты не превышает 40 см. Для выхода личинок необходимы не только соответствующие температура и влажность почвы, но и наличие растения-хозяина, секрет корней которого активирует нематод.

Для профилактики рекомендуют бороться с сорными растениями из семейств Маревые и Крестоцветные, соблюдать севооборот с возвращением повреждаемых культур не ранее чем через 5 лет, использовать нематодоустойчивые сорта свеклы, применять в севооборотах растения—антагонисты фитонематоды (вика, клевер и другие бобовые, цикорий, злаковые и др.).

Представители рода Мелойдогини из семейства Мелойдогиниды (*Meloidogynidae*) относятся к галловым нематодам. Особенность галловых нематод в том, что самки у них не превращаются в цисту. Биологический цикл у всех галловых нематод одинаков.

Находящиеся в почве личинки второго возраста находят корни растений-хозяев и внедряются в них, располагаясь головным концом к сосудам проводящих пучков корня, а задним концом к коре корня. Нематода выделяет особые вещества, под действием которых вокруг головной части фитогельминта образуются гигантские клетки. Содержимым этих клеток и питаются паразиты.

Личинки развиваются и линяют, проходя стадии личинок третьего и четвертого возрастов, а затем превращаются в самок и самцов. У нематод средняя часть тела сильно утолщается. Самцы выходят в почву, а самки после последней линьки сильно раздуваются. Самка разрывает кору корня растения и задняя ее часть с вульвой высывается наружу, становясь доступной для оплодотворения. Головная часть самки остается погруженной в корень, где самка продолжает питаться.

Размножаются галловые нематоды и партеногенетически. Общее число яиц достигает сотни и более. Откладка яиц происходит в течение всего вегетационного периода растения. Из тела самки яйца выходят в виде яйцевого мешка. Мешки хорошо видны на корнях невооруженным глазом. Развитие яиц происходит в мешке до личинок второго возраста, которые уходят в почву. Далее цикл повторяется. В теплицах южная галловая нематода может дать до семи поколений.

Важной особенностью галловых нематод является образование галлов. Галлы представляют собой гипертрофированную паренхиму коры корня, которая образуется вокруг внедрившейся личинки в виде вздутия. Размеры галлов могут варьировать от 1 мм до огромных разрастаний. Некоторые растения галлов не образуют. Другая особенность галловых нематод — их широкая пищевая специализация, они полифаги. Известно более 70 видов галловых нематод. В нашей стране практическое значение имеют южная галловая нематода (*Meloidogyne incognita*) и северная галловая нематода (*M. hapla*). Только северная галловая нематода в открытом грунте распространяется далеко на север. Остальные виды встречаются в южных регионах и в тепличных хозяйствах (рис. 65).

Среди представителей семейства Тиленхиды (Tylenchidae), относящихся к настоящим шишкоиглым нематодам, есть широко распространенные вредители сельскохозяйственных культур. Самки, самцы и личинки тиленхид имеют червеобразное тело с заостренными головным и хвостовым концами. Стилет развит слабо. У самок в большинстве случаев один яичник.

Пшеничная нематода (Anguina tritici) поражает все сорта пшеницы. Распространена в южных регионах страны. Из фитогельминтов пшеничная нематода является самым крупным видом: длина половозрелых самок достигает 5 мм при толщине 0,1—0,2 мм. Самцы мельче самок.

Пшеничная нематода живет в тканях надземных частей пшеницы. Типичным признаком повреждения является образование галлов вместо нормального зерна (рис. 66). Зрелые галлы напоминают по форме и размерам зерна пшеницы, но отличаются от них коричневой окраской и шероховатостью. Внутри галла находится белая масса, состоящая в основном из личинок нематод, находящихся в анабиозе.

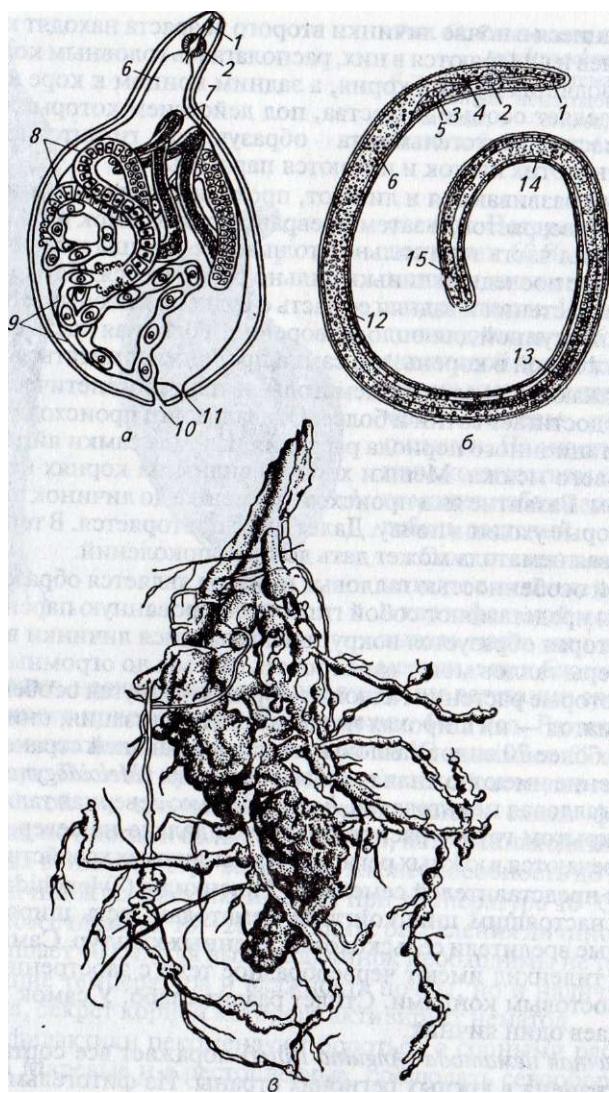


Рис. 65. Галловая нематода:

6 - самец в галле на корнях огурца; 1 - стилет; 2 - бульбус пишева
 7 - яичник Леза; 3 - Невное кольцо; 3' - "Делительное отверстие"; 6 - Гшка
 Г" тм' * - яйцевод; 9 - матка с яйцами; 10 - половое отверстие; 11 - анус
 ~ семяприемник; 13 - семяпровод; 14 — семязвергательный канал; 15 - спикулы



Рис. 66. Пшеничная нематода:
а — самка; б — молодое растение пшеницы, зараженное нематодой; в - • галлы; г — здо-
ровый колос пшеницы; д — колос, пораженный нематодой

Жизненный цикл пшеничной нематоды приспособлен к циклу развития растения-хозяина. Весной вместе с галлами фитогельминты попадают в почву с высеваемым зерном, в почве могут находиться и галлы, которые осипались с поврежденных колосьев осенью. Внутри галлов содержатся личинки второго возраста. При разбухании галлов под действием почвенной влаги личинки выходят и двигаются по направлению к проросткам пшеницы, но не далее 15—20 см от материнского галла. Значительная часть личинок погибает. Достигшие проростков пшеницы личинки заползают в пазухи листьев и становятся эктопаразитами. Яровые посевы поражаются весной, а озимые могут повреждаться осенью.

В период формирования колоса фитогельминты перебираются в зачатки цветков злаков, из которых формируются галлы. Для окончательного формирования галла необходимо присутствие нескольких самцов и самок нематоды. В галле они достигают половой зрелости и копулируют. Самка откладывает до 2,5 тыс. яиц. Из яиц появляются личинки, которые питаются ослизненной тканью галла. В одном галле может обитать до 17 тыс. личинок. Галлы завершают свое развитие несколько позже зерен пшеницы. В это время личинки заканчивают питаться и впадают в анабиоз. Взрослые особи погибают. Часть зрелых галлов выпадает из колосьев еще до уборки урожая, но большая их часть высвобождается из колосьев во время обмолота пшеницы, засоряя зерно. В состоянии анабиоза в галле личинки могут сохранять жизнеспособность в течение 25 лет. Пора-

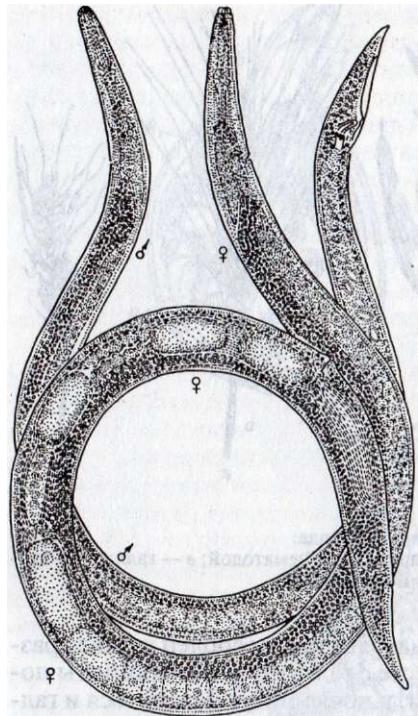


Рис. 67. Стеблевая нематода картофеля

на, затем кожура сморщивается и растрескивается, открывая доступ гнилостным микроорганизмам. В итоге происходит разрушение клубней.

Весь цикл развития стеблевой нематоды картофеля происходит внутри тканей растения-хозяина. Основной источник расселения фитогельминта — поврежденный посадочный материал, что определяет меры профилактики и борьбы с этим фитогельминтом.

Нематоды из большого маточного клубня по мере роста растения переходят по столонам в молодые клубни. Реже заселение молодых клубней происходит из почвы. В стеблях и клубнях паразиты размножаются, давая за лето несколько поколений. Осенью основная масса нематод остается в клубнях, а часть мигрирует в почву. Особое внимание следует уделять уборке послеуборочных остатков и их утилизации.

Представителем семейства настоящих шишкоиглых нематод является *стеблевая нематода на луке и чесноке (Ditylenchus dipsaci)*. Это многоядный фитогельминт, у которого насчитывают до 20 экологических рас, различающихся по тому, на каких растениях эти расы паразитируют. Луковая

жленные растения пшеницы отстают в росте, стебель искривляется.

Главная мера борьбы с пшеничной нематодой — предпосевная очистка зерна пшеницы, использование в севооборотах для очистки полей от нематод чистых паров и т. п.

Стеблевая нематода картофеля (Ditylenchus destructor) является представителем семейства настоящих шишкоиглых нематод (*Tylenchidae*) и встречается на всей территории РФ. Нематода имеет небольшие размеры: длина тела у самок 0,7—1,4 мм, у самцов — 0,7—1,3 мм. Стилет небольшой и тонкий. Стеблевая нематода поражает в основном картофель и вызывает разрушение клубней по типу сухой гнили (рис. 67). Однако могут повреждаться также морковь, горох, томат и некоторые сорные растения. Признаки поражения в виде небольших беловатых пятен видны после срезания кожуры клубня картофеля. В период хранения на пораженных клубнях появляются серые пят-

раса широко распространена в нашей стране. По строению стеблевые нематоды на луке и чесноке сходны со стеблевой нематодой картофеля.

У поврежденных луковиц репчатого лука рыхлые чешуи, их ткань набухает. У лука-севка на боковой стороне поврежденной луковицы заметно ярко-белое пятно — очаг размножения фитогельминта. Заселенные растения отстают в росте, а сильное поражение вызывает гибель посевов. После гибели растения нематоды уходят в почву. Стеблевая нематода может впадать в анабиоз и сохранять жизнеспособность в течение 2—3 лет.

В благоприятных условиях нематоды размножаются в тканях растений непрерывно, давая поколение за поколением. Нижний порог активности фитогельминта лежит в пределах 7—9 °С. После уборки урожая стеблевая нематода сохраняется в послеуборочных остатках поврежденных растений, реже в почве. Если лук хранят при высоких температурах, то пораженные луковицы усыхают, а если при низких температурах, то загнивают. Основными источниками распространения луковой расы стеблевой нематоды являются лук-севок и почва, что и определяет профилактику и борьбу с этим фитогельминтом.

В нашей стране широко распространены и другие расы стеблевой нематоды, в том числе на землянике (*D. dipsaci*), которая по своему строению идентична луковой расе. У пораженных растений видны вздутия на стеблях, черешках и листьях. Эти утолщения, где происходит размножение фитогельминта, становятся мягкими и трухлявыми. Размножаются и развиваются эти нематоды только в наземных активно растущих частях без выхода во внешнюю среду. Зимует нематода в сердечках земляники, реже в листьях. За сезон она дает до четырех-пяти поколений. Главной причиной распространения нематоды является использование заселенной ею рассады земляники.

Особенности экологии фитонематод. Жизненный цикл фитонематод связан с растениями и почвой. Взаимоотношения нематод с окружающей внешней средой, и прежде всего с растениями, определяют целый ряд важных особенностей, которыми обладают эти животные, свидетельствующих о путях развития круглых червей от типичных почвенных форм до узкоспециализированных вредителей растений. На основе взаимоотношения с растением построена экологическая классификация нематод.

Типичные сапробионты — нематоды, живущие в гниющей среде. Часть из них обитают в мертвой древесине, другие — в разлагающихся корнях, третий — в погибших зеленых растениях. У типичных сапробионтов кутикула имеет повышенный коэффициент полупроницаемости, что препятствует проникновению вредных веществ в тело нематод из гнилостной среды. Следует учитывать, что сапробиотические процессы сопровождаются постоянной сменой бактериальной флоры, что естественно ведет к соответствующим изменениям и в видовом составе нематод.

Нетипичные сапробионты. Для этих нематод также характерно обитание в гнилостной среде, но они могут заселять и живую ткань растений, проникая в нее самостоятельно или вслед за патогенными организмами. У таких нематод повысилась проницаемость кутикулы, изменя-

нился характер движения и замедлились процессы развития и темпы размножения. Нематоды в значительной степени зависят от бактерий и их деятельности, так как сами черви не способны воздействовать химическим путем на растительную ткань. Они способны лишь к повреждению растений, измельчению тканей и проглатыванию ее мелких частиц.

Прикорневые нематоды обитают в ризосфере и связаны с растением. Эти нематоды либо имеют стилет, либо являются хищниками. Первые, прокалывая стилетом ткани растения, питаются главным образом его соками, вторые пытаются животными организмами, обитающими в прикорневой части растений.

Фитогельминты относятся к настоящим вредителям растений. Для них характерны постоянное пребывание в органах растений и развитие в пищеводе желез, секрет которых способен разрушать ткани растения-хозяина. При этом изменения происходят не только в поврежденном органе, но и в растении в целом.

Все фитогельминты вооружены стилетом, а их кутикула обладает наивысшими барьерными свойствами. Фитогельминты не совместимы с сапробиотической средой: при загнивании пораженного органа они уступают место сапробиотическим организмам, лишь некоторые фитогельминты могут существовать в гниющих остатках, питаясь, по-видимому, гифами грибов. Таким образом, некроз ткани приводит к переселению фитогельминта в здоровые части растения или в почву.

Воздействие нематод на растения. Многие фитогельминты приспособились к жизни в ограниченном круге растений-хозяев. Например, стеблевая нематода картофеля живет только в тканях растений, которые богаты углеводами. Фитогельминт вырабатывает большое количество амилолитических ферментов, гидролизующих крахмал клубней до Сахаров. Отсюда и разрушение клубня по типу сухой гнили вследствие оттягивания воды из поврежденной части клубня в здоровую.

Стеблевая нематода на луке и чесноке вызывает сильное набухание поврежденного участка луковицы из-за роста активности пектиназы, разрушающей чешую.

Галловые нематоды вызывают изменения в процессах обмена веществ растения. Фитогельминты ведут неподвижный образ жизни на корнях растений, поэтому поврежденные корни не разрушаются, а только сильно видоизменяются. Вокруг переднего конца гельминта формируется группа гигантских многоядерных клеток, а в паренхиме корня происходит усиленное разрастание ткани, приводящее к образованию галла. Нематода питается за счет гигантских клеток, в которые паразит вводит стилет и выделяет пищеварительные ферменты. Клетка, выделяя вещества-ингибиторы, быстро тормозит действие гидролитических ферментов нематоды. В результате нематода получает часть пищи и ткань растения остается живой.

Фитогельминты хорошо приспособились к жизни во всех органах растений-хозяев. Патогенные фитогельминты иногда могут быть полезными, если поражают сорные растения. В перспективе возможно будет использовать некоторые виды нематод в борьбе с сорняками.

Не только фитогельминты воздействуют на растения, но и растения могут оказывать существенное влияние на вредителей. Известно, например, что при выращивании растений под пленочными укрытиями, а не в теплицах из стекла значительно снижается ущерб от галловых нематод. Оказывается, что пленочное укрытие пропускает больше ультрафиолетовых лучей, которые способствуют усиленному синтезу в растении ингибиторов, подавляющих ферменты фитонематод.

Через почву и растения-хозяев на фитогельминтов опосредованно воздействует комплекс абиотических и биотических факторов. Из абиотических факторов наибольшее влияние оказывают температура, влажность, механический состав почвы, ее кислотность, обеспеченность кислородом и насыщенность солями.

Биотические факторы сложны и многочисленны: отношения нематод с микроорганизмами в агробиоценозе, внутривидовые отношения (между фитонематодами и свободноживущими нематодами, хищниками и паразитами, между отдельными видами фитонематод за пищу и т. п.), отношения между патогенными грибами и бактериями, естественные враги нематод и пр.

Абиотические факторы. Климатические факторы, в том числе температура и влажность, определяют географическое распространение нематод. Для каждого вида характерны свои оптимальные и предельные температуры (низкие и высокие), при которых происходит его развитие. Широкое использование приемов выращивания растений в закрытом грунте позволило фитогельминтам продвинуться далеко на север. Большинство видов цистообразующих нематод приспособлены к температурам и влажности средней полосы. Стеблевая нематода картофеля в малой степени зависит от влажности и поэтому распространена во всех картофелепроизводящих районах.

Для нематод, у которых почва является средой для развития или сохранения и активации личинок, решающее значение имеет сочетание оптимальных показателей влажности и температуры в период массового заселения надземной части и корней всходов инвазионными личинками.

У нематод, непрерывно развивающихся в тканях растений без выхода во внешнюю среду, имеется пик численности паразита, когда температура и влажность оптимальны для быстрого развития популяции.

Большое значение для нематод имеет механический состав почвы, поскольку по ее порам нематоды передвигаются. Поэтому большинство видов нематод обитают в легких почвах.

Биотические факторы. Наиболее изучены взаимосвязи нематод с различными организмами почвенного биоценоза. Некоторые из них являются естественными врагами нематод, другие (грибы и бактерии) выступают компонентами в сложных болезнях растений, третьи используют нематод в качестве переносчиков от одного растения к другому (вирусы, бактерии, грибы).

Естественными врагами нематод могут быть хищные нематоды, клещи, грибы, вирусы и бактерии. Последние вызывают болезни нематод и их гибель. Почвенные грибы как враги нематод имеются во всех

почвах, но особенно их много в почвах, богатых органикой. Серьезный ущерб нематодам наносят хищные грибы — гифомицеты. Они ловят своих жертв с помощью клейких колец, узлов или спор. Способ поражения хищными грибами простой: споры или кольца прорастают в тепло жертвы, и гриб начинает питаться нематодой. Есть грибы, поражающие яйца в цистах нематод. В таких пораженных яйцах личинки погибают. Грибы из родов *Verticillium* и *Nematophthora*, поражающие самок цистообразующих нематод, в Великобритании предлагают использовать для борьбы с овсяной нематодой.

К врагам нематод относятся амебы, инфузории, споровики, тихоходки, членистоногие, в том числе клещи, и хищные нематоды.

К неблагоприятным условиям среды (низкой температуре зимой, сухости почвы, отсутствию растений-хозяев) нематоды приспособились по-разному. Самый распространенный способ — анабиоз (сохранение жизнеспособности в неблагоприятных условиях). В состояние анабиоза у большинства видов могут впадать личинки второго возраста, заключенные в цисты или галлы, а также личинки, находящиеся в растении, но в последнем случае анабиоз менее продолжителен.

Круглые черви — паразиты животных и человека. Значительное число круглых червей являются паразитами различных сельскохозяйственных животных, а также большинства диких позвоночных. Эти гельминты вызывают опасные заболевания, снижают продуктивность животных, причиняя существенный ущерб животноводству страны. Некоторые круглые черви паразитируют и у человека. Наиболее часто у животных и человека паразитируют нематоды из группы геогельминтов. Очень опасны для человека биогельминты трихинелла спиральная, нитчатка Банкрофта и другие. Рассмотрим биологические особенности наиболее распространенных в нашей стране и опасных паразитических видов нематод.

Аскариды (различные виды сем. *Ascaridae*) живут в кишечнике многих диких, домашних и сельскохозяйственных млекопитающих (свиньи, лошади, птица, кролики, мелкий и крупный рогатый скот, собаки и др.), а также человека, особенно часто детей. Особенностью аскарид является их видовая специфичность: каждому виду млекопитающего присущ свой вид аскариды (рис. 68). Свиная (*Ascaris suum*) и человеческая (*Ascaris lumbricoides*) аскариды весьма близки по многим признакам, но человек редко заражается свиной аскаридой, а свинья — человеческой. Одна из самых крупных аскарид паразитирует в тонком кишечнике лошадей (*Parascaris equorum*). У человека паразитирует аскарида, длина которой достигает 20 см.

Аскариды имеют веретенообразное тело длиной 20—40 см при диаметре до 3—5 мм. Хорошо развит половой диморфизм: самки значительно крупнее самцов, хвостовой отдел самцов загнут крючком. Плодовитость самок очень высокая — 200 тыс. яиц за сутки. Оплодотворение внутреннее. Оплодотворенное яйцо одевается прочными оболочками, которые хорошо защищают зародыш от неблагоприятных условий внешней среды. Погруженные в слабый раствор формалина

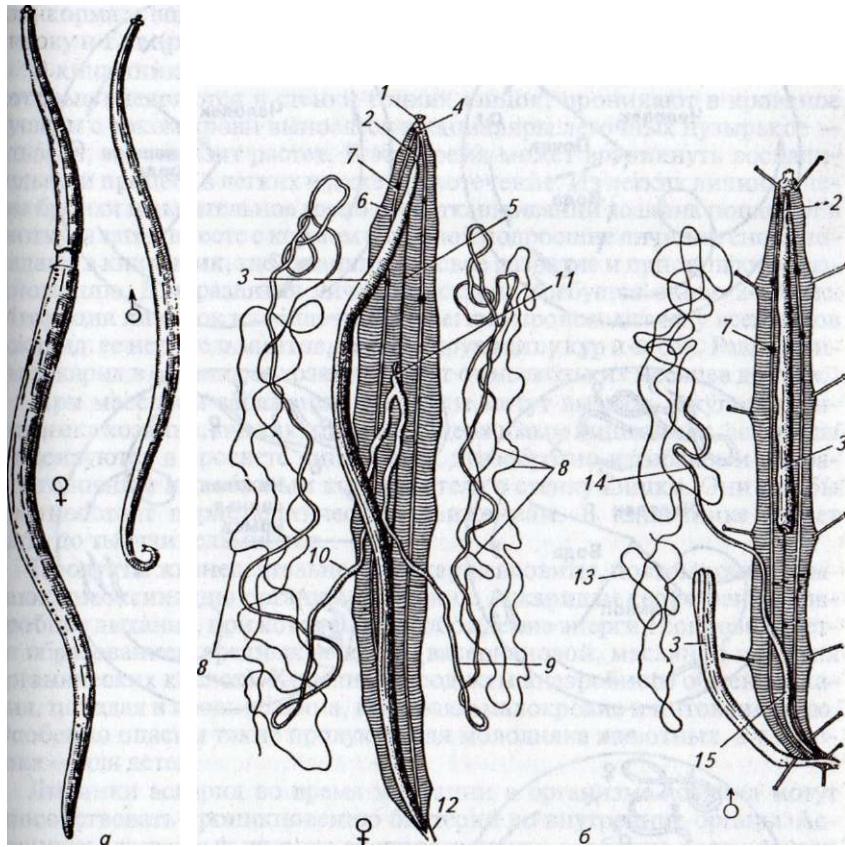


Рис. 68. Аскарида лошадиная:
 а — внешний вид; б — внутреннее строение: 1 — рот; 2 — пищевод; 3 — кишка; 4 — окологлоточное нервное кольцо; 5 — брюшной нервный тяж; 6 — боковой валик гиподермы с каналом выделительной системы; 7 — фагоцитарные клетки; 8 — яичники; 9 — яйцеводы; 10 — матка; // — влагалище; 12 — анальное отверстие; 13 — семенник; 14 — семязапровод; 15 — семязвергательный канал

яйца аскарид сохраняют жизнеспособность более месяца. Вредно действуют на яйца ультрафиолетовые лучи.

Дробление яйца начинается в теле самки аскариды, но основное развитие личинки проходит во внешней среде в течение 8—30 сут, что определяется главным образом температурой среды. Во внешнюю среду яйца попадают с фекалиями хозяина. После окончания развития личинки яйцо становится инвазионным, т. е. способным к заражению хозяина. Проглатив такое яйцо, животное может заболеть аскаридозом. Заражение происходит при потреблении загрязненных яйцами аска-



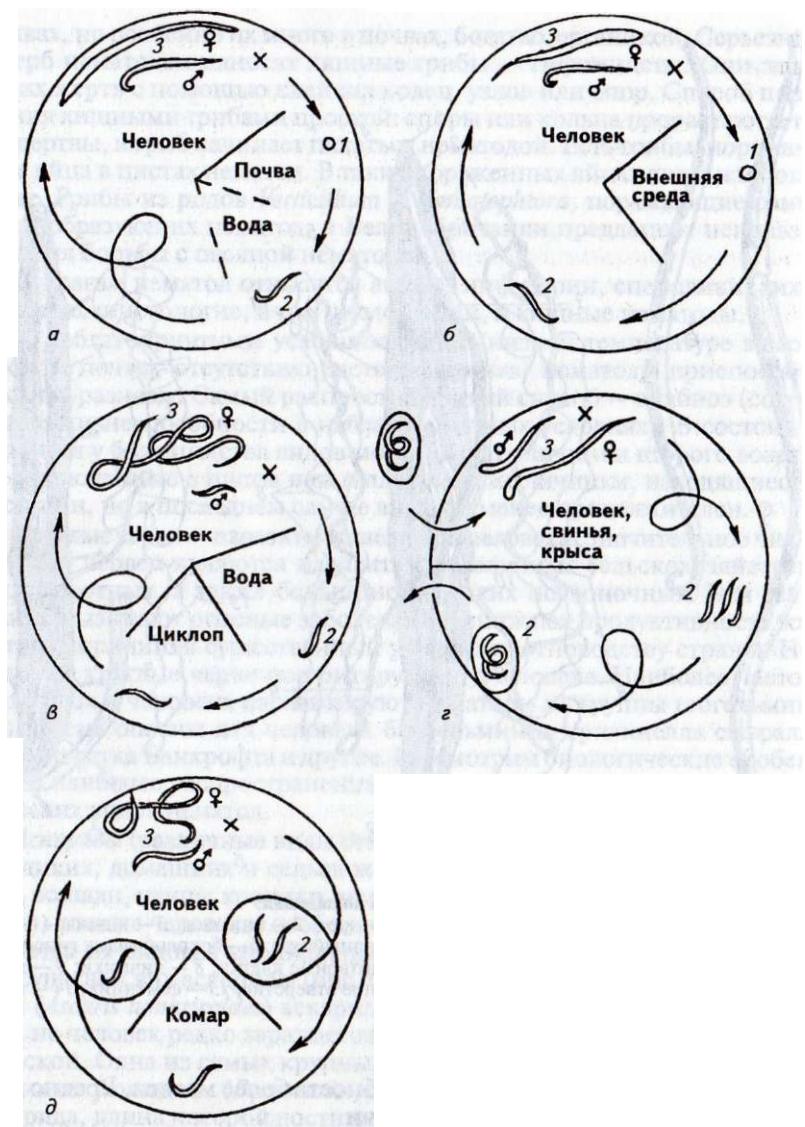


Рис. 69. Схемы жизненных циклов нематод — паразитов человека:
 а — геогельминты без смены хозяев и с миграцией личинок по крови (аскарида, власоглав и свайник); б — геогельминты без смены хозяев и без миграции личинок по крови (острица); в — геогельминты со сменой хозяев (ришта); г — биогельминты без смены хозяев (трихинелла); д — биогельминты со сменой хозяев (нитчатка Банкрофта); 1 — яйцо; 2 — личинка; 3 — половозрелые особи

рид корма и воды, а также через предметы ухода, оборудование, подстилку и т. п. (рис. 69).

В кишечнике хозяина из яиц выходят микроскопические личинки, которые внедряются в стенки тонких кишок, проникают в кровяное русло и с током крови выносятся в капилляры легочных пузырьков — ильвеол, где паразит растет. В это время может возникнуть воспалительный процесс в легких и даже кровотечение. Из легких личинки через бронхи и дыхательное горло при откашливании хозяина попадают в І лотку, а затем вместе с кормом и слюной подросшие личинки снова попадают в кишечник, где заканчивают свое развитие и приступают к размножению. Для развития личинок аскарид требуется около 2—3 мес. Миграция личинок из кишечника в легкие происходит не у всех видов аскарид: ее нет у гельминтов, паразитирующих у кур и собак. Разные виды аскарид в организме хозяина живут от нескольких месяцев до года.

При массовом заражении аскариды могут вызвать закупорку кишечника хозяина, так как, питаясь содержимым кишечника, аскариды фиксируются в просвете кишечника, дугобразно изгинаясь и упираясь головным и хвостовым концами тела в стенку кишки. Они как бы противостоят перистальтическим движениям. В кишечнике может быть до тысячи гельминтов.

Продукты жизнедеятельности аскарид ядовиты, поэтому они вызывают интоксикацию организма хозяина. Аскаридам свойственно анаэробное дыхание, при котором высвобождение энергии сопровождается образованием вредных веществ (валериановой, масляной и других органических кислот). Конечные продукты анаэробного обмена аскарид, попадая в кровь хозяина, вызывают малокровие и интоксикацию. Особенно опасны такие продукты для молодняка животных, а у человека — для детей.

Личинки аскарид во время миграции в организме хозяина могут способствовать проникновению бактерий во внутренние органы. Аскаридозы животных широко распространены, особенно часты случаи заболевания лошадей и свиней. В борьбе с аскаридозами важное значение имеют меры профилактики: чистота в помещении, правила гигиены, содержание животных (кроликов, птиц и др.) на сетчатых полах и др. Особое внимание уделяют удалению навоза, чистоте туалетов и борьбе с мухами — переносчиками яиц гельминтов.

Установлена роль земляных червей в распространении аскаридоза: черви заглатывают яйца аскарид вместе с растительными остатками и почвой. В кишечнике червей из яиц выходят личинки, которые сохраняют в червях жизнеспособность до года. Поедая дождевых червей, свиньи заражаются аскаридами. Подсосные пороссята заражаются через вымя. Личинки аскарид могут сохраняться в организме мух, личинках жуков и других насекомых, которые, как и дождевые черви, являются резервуарными хозяевами аскарид.

В последние годы получены данные, что при заглатывании человеком инвазионных яиц свиной аскариды личинки могут выходить из яиц и мигрировать по всему организму человека, оседая в различных

внутренних органах, например в мозге, печени и т. п. Это вызывает у человека тяжелое заболевание и общую интоксикацию организма.

В кишечнике кур паразитирует небольшая нематода, внешне похожая на аскариду, — *аскарида* (*Ascaridia galli*). Самки этой нематоды достигают в длину 11 см, самцы меньше — до 7 см. Яйца аскаридии после выхода с пометом наружу становятся инвазионными на 7—20-е сут. Куры и цыплята заражаются, поедая загрязненный инвазионными яйцами корм. В кишечнике птицы личинки высвобождаются из оболочек яиц, затем внедряются в стенки кишечника, где развиваются в течение 16—20 сут. Выйдя снова в просвет кишечника, личинки достигают зрелости. Весь цикл развития длится 35—58 сут. Особенностью развития аскаридий является отсутствие в их цикле миграции по телу хозяина.

Аскаридиоз вызывает массовую гибель молодняка, снижает яйценоскость взрослых кур. Резкому снижению заболеваемости аскаридиозом способствует содержание птицы в клетках или на сетчатых полах.

Острицы (различные виды сем. *Oxyuridae*) паразитируют в толстом отделе кишечника позвоночных и человека. Это мелкие паразиты, имеющие вздутие пищевода (бульбус) и тонкий заостренный хвостовой конец. Длина человеческой остирицы (*Enterobius vermicularis*) составляет менее 2 см, тогда как лошадиная остирица (*Oxyura equi*) достигает 6—18 см за счет своего длинного хвоста. Острицы *O. equi* вызывают болезнь оксиуроз у лошадей, мулов и зебр. Паразитируют оксиуры в большой ободочной кишке, но могут заселять слепую кишку и даже тонкий отдел кишечника. В результате у лошадей нарушается деятельность пищеварительного тракта и поражается кожа у корня хвоста. Паразиты распространены повсеместно.

После оплодотворения самки самцы погибают. Переполненные зрелыми яйцами самки вместе с фекалиями спускаются к анальному отверстию лошади. Они выходят из кишечника пассивно. Часть самок падают на землю и откладывают яйца на поверхность испражнений, а часть задерживаются в складках слизистой оболочки вокруг ануса и откладывают яйца в перианальной области под корнем хвоста лошади. Самки после откладывания яиц погибают. Клейкая слизистая масса, в которой находятся яйца, образует сероватый налет на перианальной области в ее складках.

Под хвостом и в области промежности через 2—3 сут яйца становятся инвазионными. При движении хвоста лошади яйца попадают во внешнюю среду, загрязняя подстилку, траву, стены денника, кормушки и т. п. Если инвазионные яйца попадут в кишечник лошади, то из них выходят микроскопические личинки и развиваются во взрослых гельминтов. Чаще оксиурозу подвержен молодняк и старые лошади. Больные животные испытывают сильный зуд в области хвоста, они расчесывают эти места о выступающие части денника, способствуя осипанию яиц.

У человека, чаще всего у детей, которые легко самозаражаются этим гельминтом, в толстой и задней кишке паразитирует детская остирица (*Enterobius vermicularis*). Это мелкие черви длиной 5—10 мм белого цвета. Оплодотворенные самки выползают ночью из прямой кишки и

откладывают яйца на кожу вокруг ануса. Уже через 10—12 ч яйца становятся инвазионными. Самка откладывает около 11 тыс. яиц. Яйца попадают на белье, постельные принадлежности, на пол и т. д. Из-за сильного зуда в области анального отверстия дети расчесывают это место и легко самозаражаются (автоинвазия).

Nematoda Toxascaris leonina из сем. Ascaridae во взрослой стадии паразитирует у молодняка старших возрастов и взрослых домашних и диких плотоядных животных, вызывая заболевание токсасскаридоз. Это черви светло-желтого цвета длиной 6—10 см (самки) и 4—6 см (самцы).

Выделенные с калом во внешнюю среду яйца при благоприятных условиях дозревают до инвазионной стадии в течение недели. У заглотивших их зверей в тонком отделе кишечника из яиц выходят личинки; они внедряются в стенку кишечника и линяют. Через некоторое время личинки возвращаются в просвет кишечника и через 3—4 нед достигают половой зрелости.

Токсасскаридозом не болеют новорожденные и молодые щенки, кроме щенков песца. Это болезнь взрослых плотоядных, в том числе кошек и собак.

У молодых щенков паразитирует *Toxocara canis*, а у кошек — *T. mystax*, вызывая заболевание, называемое токсокарозом. Токсокары (в отличие от токсасскарид, развивающихся прямым путем) в личиночной стадии мигрируют по организму (кишечник — кровяное русло — легкие — трахея — гортань — кишечник) в течение месяца.

Заражение плотоядных токсокарозом и токсасскаридозом возможно не только прямым путем, но и через резервуарных хозяев, которыми могут быть мыши и другие грызуны. Личинки *Toxascaris leonine* у грызунов концентрируются в стенках желудка или кишечника. У песцов наблюдается внутриутробный путь заражения животных.

Яйца токсокар и токсасскарид очень устойчивы к воздействиям неблагоприятных факторов среды: при обработке фекалий 5 %-ным раствором фенола они погибают только через 3 нед. При сильном заражении токсокары и токсасскариды оказывают токсическое воздействие, вызывая воспаление кишечника, а иногда и его закупорку. Из кишечника токсасскариды могут проникать в желчные протоки печени, протоки поджелудочной железы, желудок и даже трахею. Токсокары более опасны, так как часть мигрирующих с кровью личинок попадают в различные органы и ткани хозяина и там инкапсулируются, долго сохраняя жизнеспособность. Хищники, поедая животных, инвазированных цистами токсокар, заражаются ими. Есть токсокары, которыми заражаются дети, играя в песке на детских площадках, где испражняются больные кошки.

Стронгиляты (ряд видов подотряда Strongylata). Обширная и разнообразная группа круглых червей небольших размеров, паразитирующих в толстом отделе кишечника у разных позвоночных животных. В огромных количествах они встречаются и у сельскохозяйственных животных, в том числе лошадей и других непарнокопытных. Именно сельскохозяйственные животные в силу скученности их содержания и

ограниченности территорий пастбищ представляют для паразитических червей благоприятную среду для заселения.

Насчитывают около 45 видов нематод — возбудителей кишечных стронгилятозов лошадей. Все они относятся к геогельминтам и имеют сходное развитие во внешней среде. Кишечные стронгилятозы — самые распространенные и повсеместно встречающиеся гельминтозы. Почти все лошади с самого раннего возраста поражаются этими болезнями. Интенсивность заселения паразитами (от нескольких сотен особей до многих десятков тысяч) зависит от возраста, условий содержания и кормления животных. Часто стронгилятозы приносят существенный ущерб: от отставания в росте до летального исхода.

Паразиты имеют небольшие размеры: самцы достигают 0,5—4,5 см, самки крупнее. Оплодотворенные самки продуцируют множество яиц, которые с фекалиями попадают во внешнюю среду. При благоприятных условиях среды (8—38 °C) в яйце формируется личинка, которая выходит из яйца, развивается и линяет, достигая инвазионной стадии. При достаточной влажности личинки мигрируют горизонтально и вертикально в почве и по стеблям растений. Лошади заражаются, потребляя траву и воду, загрязненные инвазионными личинками. Заражение происходит в теплое время года на пастбище или в течение всего года в утепленных конюшнях.

В организме лошадей развитие различных стронгилят протекает неодинаково. У некоторых видов (деляфондии) личинки совершают миграцию: кишечник — кровеносные сосуды — тромбы в сосудах (где личинки развиваются) — кровяное русло — стенка кишечника — просвет кишечника. У других стронгилят (*Strongylus equities*) — самых крупных стронгилид лошадей (самцы достигают в длину 25—35 см, самки — 35—45 см), личинки через слизистую оболочку кишечника мигрируют в поджелудочную железу, где развиваются в течение 8 мес. Затем возвращаются в толстый кишечник. Весь срок развития составляет почти год.

В пищеварительном тракте жвачных животных паразитирует большое число видов нематод из подотряда Strongylata. Эти паразиты тоже относятся к геогельминтам. Развитие их во внешней среде протекает так же, как и у кишечных стронгилят лошадей.

Нематоды подотряда Strongylata вызывают также стронгилятозы органов дыхания у сельскохозяйственных животных. Заболевание диктиокаулез у жвачных животных вызывают нематоды рода *Dictyocaulus*. Это довольно крупные круглые черви с нитевидным телом белого цвета, паразитирующие в легких крупного рогатого скота, овец и других млекопитающих. Один из представителей, *Dictyocaulus viviparus*, вызывает опасное заболевание дыхательных путей у крупного рогатого скота. Эти нематоды паразитируют в бронхах и трахее животных. Болеет в основном молодняк крупного рогатого скота. Заболевание характеризуется развитием бронхита и бронхопневмонии.

Во внешней среде личинки становятся инвазионными через 4—10 сут после их выхода с фекалиями из организма хозяина. Личинки редко покидают фекалии. Их распространению способствует гриб

Pilobolus, который, раскрываясь, разбрасывает личинок на расстояние до 3 м. Распространению личинок способствуют также паводковые воды. С ними личинки попадают в водоемы. Особую опасность представляют мелкие лужи на пастбищах.

Телята, проглатившие личинок с травой или водой, заражаются дикиоикаулезом. Через стенку кишечника личинки с кровью мигрируют в легкие, где завершают свое развитие. Из легких половозрелые паразиты выбрасываются в кишечник.

В легких свиней паразитируют метастронгилиды из семейства *Metastrongylidae*, личинки которых живут в дождевых червях. Поедая дождевых червей, свиньи заражаются этими гельминтами, которые паразитируют в бронхах свиней. У поросят гибель может достигать 30 %.

Среди биогельминтов наибольшую опасность представляет *трихинелла спиральная* (*Trichinella spiralis*), жизненный цикл которой проходит полностью в организме хозяина. У трихинеллы различают две стадии: кишечные трихинеллы и мышечные трихинеллы. Хозяевами трихинелл могут быть хищники, парнокопытные, в том числе свиньи, насекомоядные, ластоногие, грызуны и человек. У человека эти гельминты вызывают заболевание трихинеллез. Человек заражается в основном от свиней и редко от других, в частности диких животных, потребляя мясо, пораженное мышечными трихинеллами (рис. 70). В мясе зараженных свиней рассеяны небольшие овальные капсулы. В каждой капсule находится скрученная в спираль микроскопическая трихинела длиной около 0,5 мм.

Если такое трихинеллезное мясо будет плохо термически обработано и съедено хозяином, то в его желудке под действием желудочного сока капсулы растворяются и молодые трихинеллы выходят из них. Попав в тонкий отдел кишечника, трихинеллы растут и через 2—3 сут превращаются в половозрелых гельминтов. Самки достигают в длину 3—4 мм, а самцы — 1,5 мм. Черви внедряются в ткань кишечника и приступают к размножению. После спаривания самцы погибают.

Оплодотворенные самки закрепляются головным отделом в

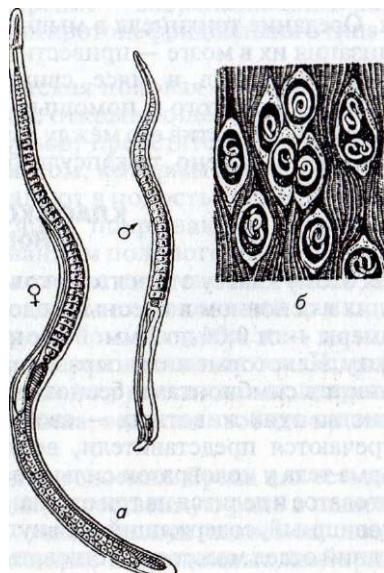


Рис. 70. Трихинелла *Trichinella spiralis*:
а - самка и самец; б—личинки, инкапсунированные в мышцах свиньи

слизистой оболочке кишки. Самки живут около 2 мес, и за это время каждая отрождает примерно 2 тыс. личинок (трихинеллам свойственно яйцекиворождение). Личинки проникают в лимфатические сосуды стенки кишечника и затем в кровяное русло. С кровью личинки разносятся по всему организму и обычно попадают в мышцы. Личинки, активно двигаясь, внедряются в волокна поперечно-полосатой мускулатуры, где пытаются и растут, разрушая мышечные волокна. Затем трихинеллы закручиваются в спираль и постепенно окруждаются соединительной тканью капсулой. Примерно через год в стенках капсул откладывются соли углекислого кальция и капсулы принимают белый цвет. Так основной хозяин превращается в промежуточного.

Жизненный цикл трихинеллы может быть иным. Если сопротивляемость организма хозяина снижается, то отродившиеся личинки трихинеллы внедряются в ворсинки кишечника и, закончив там свое развитие, вновь возвращаются в просвет кишечника, где достигают половой зрелости. Это значительно увеличивает число паразитов в кишечнике и продлевает срок кишечной инвазии, а также усиливает поражение мышц.

Свиньи заражаются трихинеллезом, поедая зараженных трихинеллой дохлых крыс или свиные отходы с боен. Крысы же заражаются, питаюсь тканями павших от трихинеллеза других крыс или свиными отходами, попавшими на свалку или закопанными в землю на небольшую глубину. Зараженные трихинеллами туши свиней уничтожают, так как паразиты в капсулах чрезвычайно устойчивы к самым жестким режимам термической обработки мяса.

Для человека наиболее опасна мышечная стадия развития трихинеллы. Инкаспулирование личинок сопровождается болями в мышцах. Оседание трихинелл в мышцах глаз может вызвать слепоту, а локализация их в мозге — привести к смертельному исходу. Наличие личинок трихинелл в мясе свиней можно проверить в домашних условиях. Для этого с помощью острой бритвы делают тонкий срез мышцы, и, поместив его между двумя стеклами, рассматривают в лупу. Если мясо заражено, то капсулы будут видны.

КЛАСС КОЛОВРАТКИ (*Rotatoria*)

К этому классу относятся около 1,5 тыс. видов круглых червей, живущих в основном в пресных водоемах и имеющих микроскопические размеры — от 0,04 до 2 мм. Реже их можно встретить в морях, болотах, во мху. Некоторые виды паразитируют у беспозвоночных, а некоторые являются симбионтами беспозвоночных. Однако подавляющее большинство этих животных — свободноживущие черви, хотя среди них встречаются представители, ведущие прикрепленный образ жизни. Форма тела у коловраток сильно варьирует, но у большинства оно продолговатое и делится на три отдела: головной с мерцательным аппаратом, туловищный, содержащий все внутренности, и задний, или ножной. Последний отдел может отсутствовать. Совокупность двух венчиков ресни-

Рис. 71. Схема строения коловратки:
 а — вид спереди; б — вид сбоку; 1 — гионные чувствительные щупальца; 2 — коловорачательный аппарат; 3 — рот; 4 — глотка с жевательным аппаратом; 5 — слюнные железы; 6 — пищевод; 7 — желудочные железы; 8 — желудок; 9 — яичник; 10 — протонефридий; 11 — задняя кишечка; 12 — мочевой пузырь; 13 — отверстие клоаки; 14 — семенные железы; 15 — ножной ганглий; 16 — пальцы ноги; 17 — клоака; 18 — спинные чувствительные щупальца; 19 — надглоточный ганглий

чек образует коловорачательный аппарат, вызывающий водоворот, который направляет мелкие частицы ко рту коловратки (рис. 71).

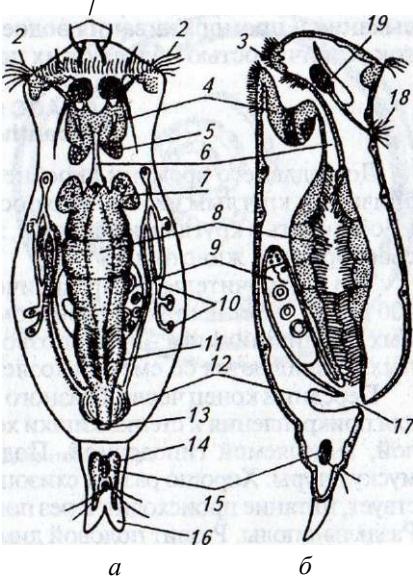
С помощью ноги коловратки, недущие прикрепленный образ жизни, временно или постоянно прикрепляются к субстрату. В ноге хорошо развиты мышцы. Кожно-мускульный мешок отсутствует.

Пищеварительная система представлена ртом, глоткой, пищеводом, объемистым желудком и кишкой, которая заканчивается анальным отверстием. Глотка имеет расширение — зоб, снабженный хитиновыми челюстями. Органы выделения протонефридиального типа открываются в клоаку.

Коловратки раздельнополы. Женская половая система состоит из яичника, желточника и яйцевода, открывающегося в клоаку. У самцов имеется семенник, семяпровод, предстательные железы и копулятивный орган с острым стилетом, которым самец прокалывает покровы самки. Спермии попадают в полость тела и проникают в яичник. Оплодотворенные яйца покрываются скорлупой. Размножение происходит с чередованием полового и партеногенетического поколений. В последнем случае получаются только самки. После нескольких партеногенетических поколений самки дают яйца с гаплоидным набором хромосом: из таких яиц выходят карликовые самцы и самки. Оплодотворенные самки откладывают покоящиеся яйца. Самцы обнаружены не у всех видов коловраток.

Из яиц сидячих форм выходит личинка, ведущая свободноплавающий образ жизни.

Коловратки многочисленны в наших водоемах. Среди них есть донные и планктонные формы, играющие существенную роль в пищевых цепях водоемов. Они же являются очистителями водоемов, поедая большое количество бактерий. Почвенные и придонные коловратки



способны к анабиозу, перенося таким образом длительное время пересыхания и промораживания водоемов. У планктонных видов коловраток устойчивостью обладают их покоящиеся яйца.

КЛАСС СКРЕБНИ (*Acanthocephala*)

До недавнего времени скребней выделяли в самостоятельный тип, близкий к круглым червям. Но последние исследования подтверждают их общность с круглыми червями, хотя морфологически скребни очень своеобразные животные.

Это исключительно паразитические черви, насчитывающие около 500 видов, обитающих во взрослом состоянии в кишечнике позвоночных, а в личиночном — у беспозвоночных животных, рыб и земноводных. Развиваются со сменой хозяев.

Передний конец червеобразного тела превращен в хоботок с крючьями для прикрепления к стенке кишки хозяина. Тело покрыто нежной кутикулой, выделяемой гиподермой. Под гиподермой располагается два слоя мускулатуры. Хорошо развит схизоцель. Пищеварительная система отсутствует, питание происходит через покровы тела. Органы чувств не развиты. Раздельнополы. Развит половой диморфизм. Развитие с метаморфозом.

Гигантский скребень (*Macrocanthorhynchus hirudinaceus*) длиной 25—60 см паразитирует в кишечнике свиней. Яйца попадают во внешнюю среду. Для дальнейшего развития яйца должны быть проглочены личинками бронзовок, майских жуков и др. Личинки жуков и взрослые жуки, зараженные личинками скребня, съедаются свиньями при выгульном их содержании. Сильное заражение свиней может привести к гибели животных.

Известно много скребней, которые во взрослом состоянии обитают в кишечнике грызунов, водоплавающих птиц, рыб, тюленей и др. (рис. 72). У некоторых скребней число промежуточных хозяев может достигать трех.

Например, у скребней, паразитирующих в организме тюленей, два промежуточных хозяина — мелкие ракообразные и рыба.

ФИЛОГЕНИЯ ПЕРВИЧНОПОЛОСТНЫХ ЧЕРВЕЙ

Считается, что первичнополосные круглые черви ведут свое начало от турбелляриеподобных предков. В разных классах круглых червей сохранились признаки, общие с плоскими червями: участки ресничного эпителия, протонефридии, участки паренхимы в схизоцеле. Наиболее близки к предкам современные коловратки и брюхоресничные. Схема экологической радиации круглых червей представлена на рис. 73.

ТИП НЕМЕРТИНЫ (*Nemertini*)

Немертины — свободноживущие морские черви, реже паразитические и пресноводные, предками которых предположительно были свободноживущие плоские черви. Известно около 750 видов немертин,

Рис. 72. Скрепни:
а — великан; б — четковидный

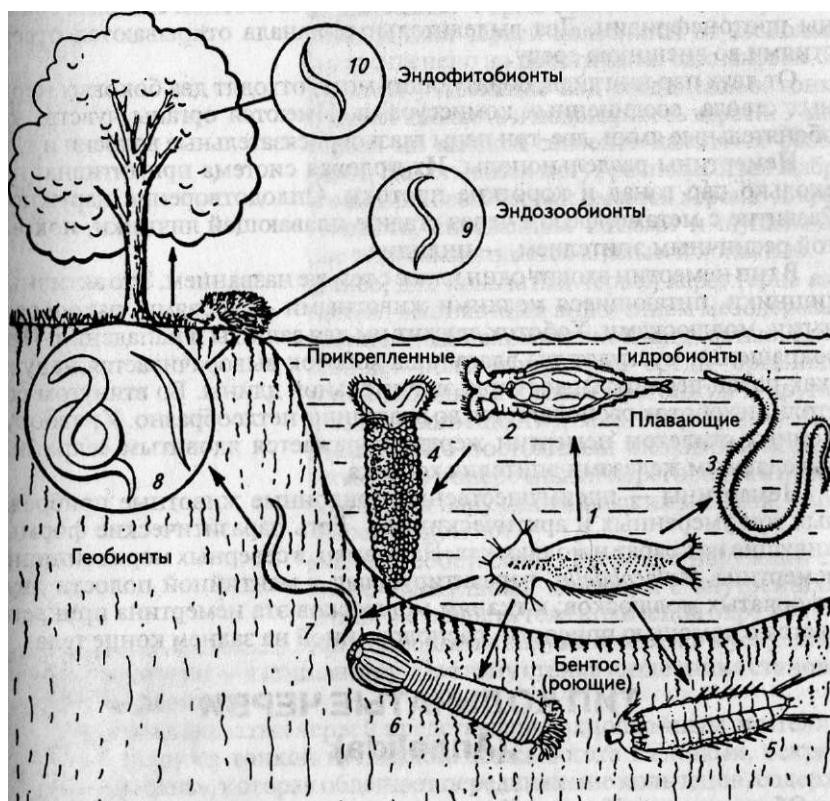


Рис. 73. Экологическая радиация Nemathelminthes:
1 — брюхоресничный червь; 2 — коловратка; 3 — волосатик; 4 — яйцо; 5 — киноринха;
6 — приапулида; 7 — нематода водная; 8 — почвенные нематоды; 9,10 — паразитические
нематоды

обитающих преимущественно в прибрежной зоне морей. Червеобразное тело (до 2 м) покрыто мерцательным эпителием (как у турбеллярий), под которым находятся кольцевые и продольные слои мышц (кожно-мускульный мешок). На переднем конце тела имеется выворачивающийся хоботок, зачастую вооруженный стилетом. С его помощью немертины захватывают добычу, которую затем отправляют в щелевидный рот, расположенный сentralной стороны у основания хоботка.

Полость тела отсутствует, промежутки между органами заполнены паренхимой, лишь влагалище хоботка выстлано целомическим эпителием. Пищеварительная система состоит из передней, средней и задней кишок. Аналльное отверстие располагается на заднем конце тела. Средняя кишка имеет боковые карманы.

У немертин появляется развитая кровеносная система, состоящая из трех основных сосудов: спинного и двух боковых, связанных между собой кольцевыми сосудами. С сосудами кровеносной системы связаны протонефридии. Два выделительных канала открываются отверстиями во внешнюю среду.

От двух пар ганглиев, образующих мозг, отходят два боковых нервных ствола, соединенных комиссарами. Имеются органы чувств; это обонятельные ямки, две-три пары глазков, осязательные волоски и т. п.

Немертины раздельнополы. Их половая система примитивна: несколько пар гонад и короткие протоки. Оплодотворение наружное. Развитие с метаморфозом через стадию плавающей личинки, покрытой ресничным эпителием, — пилидия.

В тип немертин входит один класс с тем же названием. Это активные хищники, питающиеся мелкими животными — червями, ракообразными, моллюсками. Хоботок служит им для защиты и нападения. При сокращении мускулатуры влагалища хоботок выворачивается наружу (как палец перчатки), достигая значительной длины. Во втянутом состоянии хоботок располагается во влагалище петлеобразно. У невооруженных стилетом немертин жертва поражается ядовитым секретом, выделяемым железами эпителия хоботка.

Немертины — преимущественно придонные животные прибрежных зон умеренных и арктических вод. Есть паразитические формы, живущие на крабах и моллюсках. Например, в северных морях типична немертина *Malacobdella*, паразитирующая в мантийной полости двустворчатых моллюсков; к тканям моллюсков эта немертина прикрепляется с помощью присоски, расположенной на заднем конце тела.

ТИП КОЛЬЧАТЫЕ ЧЕРВИ (*Annelida*)

Общая характеристика. Кольчатые черви, или кольчечты, — наиболее высокоорганизованная группа червей с усложненными по сравнению с другими типами червей системами органов. Это двустороннесимметричные животные, характеризующиеся наличием у них вторич-

Июй полости тела, или целома. Метамерия у кольчатых червей выражается в том, что их тело снаружи расчленено на ряд однотипных сегментов, в каждом из которых повторяются многие органы. Известно около 12 тыс. видов кольчатых червей, ведущих свободный образ жизни главным образом в морях, а также в пресных водоемах и в почве. Многие из них имеют важное значение, так как служат кормом для бес-
I юзовоночных и позвоночных животных, участвуют в процессах почво-образования, служат объектом разведения, используются в медицинских целях и т. д. Паразитических видов немного. Кольчцы активно участвуют в деструкции органического вещества, вовлекая тем самым высвободившиеся биогенные элементы в круговороты. Особенно многообразны морские формы, которые живут на разных глубинах до 10 км.

Тип Кольчатые черви включает три класса: Многощетинковые черви (*Polychaeta*), Малощетинковые черви (*Oligochaeta*) и Пиявки (*Hirudinea*).

Строение и жизненные отправления. Форма тела кольчатых червей вытянутая и слегка уплощенная. Длина червей колеблется от нескольких миллиметров до метра. Тело расчленено на практически одинаковые сегменты (гомономная сегментация), имеющие вид соединенных тонкой кожей колец, что обеспечивает гибкость и подвижность червей. У многощетинковых морских червей на каждом сегменте находятся особые парные выросты — параподии. На сегментах могут располагаться жабры. У большинства видов многощетинковых червей имеется хорошо дифференцированная головная лопасть, снабженная глазами и шупальцами (рис. 74). Сегментированное тело заканчивается анальной лопастью.

Как уже указывалось ранее, для кольчатых червей характерна *вторичная полость тела*, или *целом*, выстланный эндотелием мезодермального происхождения и заполненный целомической жидкостью. Целомическая жидкость выполняет роль внутренней среды организма, функции гидроскелета, в ней находятся клетки (фагоциты и другие), она участвует в переносе различных веществ и защите организма. В целоме поддерживается относительно постоянный биохимический режим. Целом разделен посегментно поперечными перегородками на камеры: каждый сегмент имеет свою пару целомических мешков; у многих представителей этих перегородок нет.

Первичная полость не имеет собственных стенок: с наружной стороны ее ограничивает кожно-мускульный мешок, а с внутренней — стенка кишечника. Вторичная же полость тела кольчецов окружена однослоистым эпителием, прилегающим снаружи к кожно-мускульному мешку, а внутри — к кишечнику. Поэтому стенка кишечника становится как бы двойной.

Покровы кольчатых червей представлены однослойным эпителием, одетым снаружи тонкой кутикулой. Кожа богата железами, секретирующими слизь, которая облегчает передвижение кольчецов, содержит половые аттрактанты, ядовитые вещества и т. п. У морских представителей эти секреты используются при постройке домиков.

Нервная система развита лучше, чем у других червей. Она представлена парными спинными мозговыми ганглиями и брюшной нервной

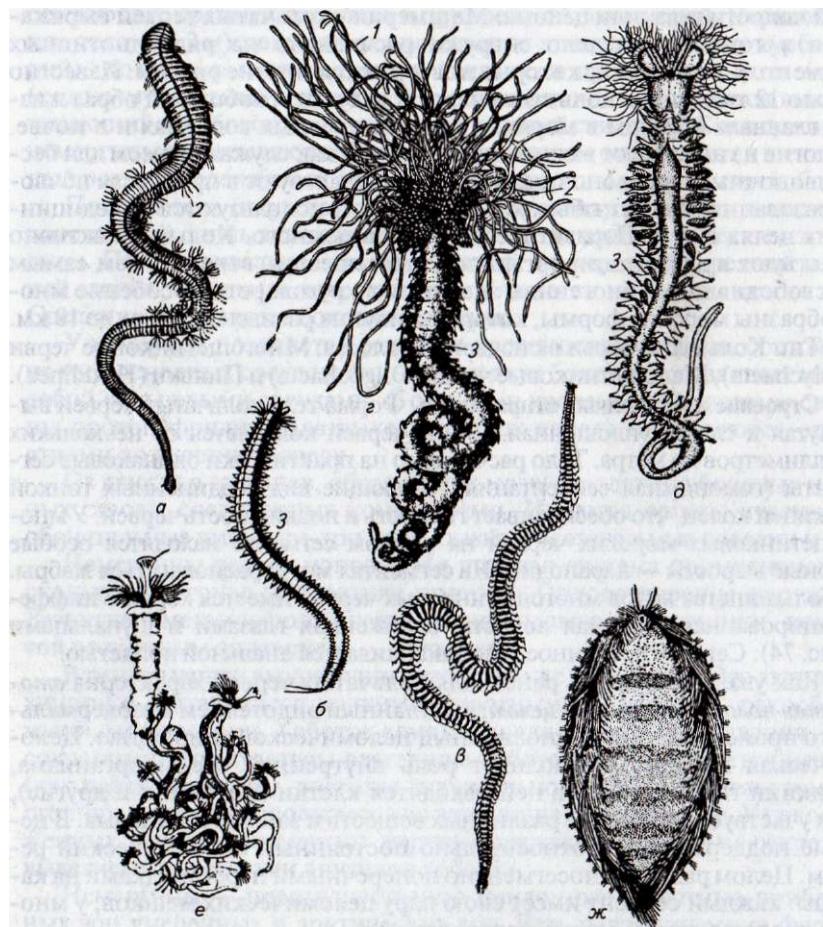


Рис. 74. Виды многощетинковых кольчатах червей:
а—пескожил (*Arenicola*); б—нереис (*Nereis*); в—филлодоце; г—амфитrite (*Amphitrite*);
д—сабеллария (*Sabellaria*); е—серпула (*Serpula*); ж—морская мышь (*Aphrodite*); 1—шу-
пальца; 2—жабры; 3—параподии

цепочкой с метамерно повторяющимися парными ганглиями в каждом сегменте. От ганглиев отходят нервы к различным органам. Нервная система типичных кольчатах червей развита значительно лучше и устроена сложнее (рис. 75). Появление головного мозга, расположенного дорсально над глоткой, существенно отличает кольчатах червей от плоских. Парные спинные доли мозга кольчецов разделены на передний, средний и задний ганглии.

Органы чувств у почвенных червей представлены многочисленными чувствующими клетками в кожном покрове. У морских многощетинковых

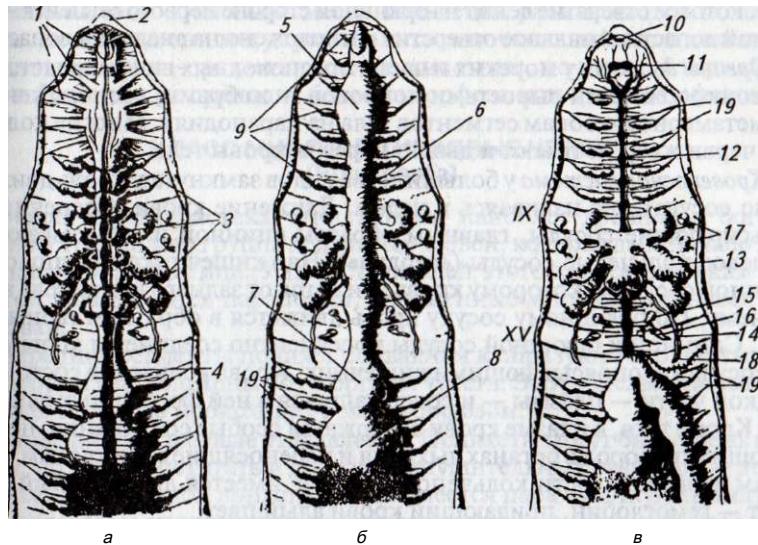


Рис. 75. Передняя часть тела дождевого черва:
и — червь с обнаженными спинным и кольцевыми кровеносными сосудами; б — пищеварительная система; в — нервная и половая системы; / — связки между глоткой и стенкой тела; 2 — ротовая лопасть; 3 — кольцевые сосуды («сердца»); 4 — спинной кровеносный сосуд; 5 — глотка; 6 — пищевод; 7 — зоб; 8 — мускульный желудок; 9 — метанефриди; 10 — рот; 11 — надглоточный ганглий и окологлоточное нервное кольцо; 12 — брюшная нервная цепочка; 13 — семенные мешки; 14 — семяпровод; 15 — яичник; 16 — воронка яйцевода; 17 — семяприемники; 18 — отверстие семяпровода; 19 — перегородки между сегментами; IX—XV — сегменты тела

червей хорошо развиты глаза и щупальца. Кожа кольчецов чувствительна к световым раздражителям. У некоторых форм имеются органы равновесия.

Мускулатура входит в состав хорошо развитого кожно-мышечного мешка; снаружи лежат кольцевые мышечные волокна, а внутри — продольные. У некоторых форм между этими слоями мышечных волокон может лежать третий слой диагональных мышечных волокон. С их помощью черви могут не только изгибать тело в разных направлениях, но и укорачивать или удлинять его. Мускулы входят в структуру части внутренних органов. Движению кольчецов способствуют особые придатки: щетинки и параподии. Щетинки имеют хитиновую природу (они приводятся в действие специальными пучками мышц).

Пищеварительная система. Кишечник состоит из трех отделов; первый и последний отделы выстланы эпителием эктодermalного происхождения, а средний отдел — энтодермального происхождения. Благодаря кровеносной системе процессы переваривания и всасывания питательных веществ идут более активно, что улучшает обеспечение органов и тканей. У части видов средняя кишка имеет глубокое втячивание (тифлозоль), увеличивающее поверхность кишечника. Имеется рот и анальное отвер-

стие. Ротовое отверстие лежит на брюшной стороне первого сегмента — головной лопасти. Анальное отверстие расположено на анальной лопасти.

Органы дыхания у морских и части пресноводных видов представлены тонкостенными выростами покровов — жабрами, расположеными метамерно по бокам сегментов тела на параподиях. Многие кольчатые черви жабр не имеют и дышат через покровы тела.

Кровеносная система у большинства видов замкнутая: кровь движется по сосудам, не изливаясь в целом. Движение крови обеспечивают пульсирующие сосуды, главным образом спинной и опоясывающие пищевод кольцевые сосуды («сердца»). Над кишечником расположен спинной сосуд, по которому кровь движется от заднего конца тела к переднему. По брюшному сосуду кровь движется в обратном направлении. Спинной и брюшной сосуды посегментно соединены кольцевыми сосудами, опоясывающими кишечник. Кровь кольчецов состоит из жидкой части — плазмы — и содержащихся в ней форменных элементов. Кроме того, в плазме крови содержатся особые соединения, поглощающие кислород в органах дыхания и переносящие его к тканям и органам червей. У части кольчецов в плазме имеется дыхательный пигмент — гемоглобин, придающий крови алый цвет.

Органами выделения служат посегментные метанефридиальные эктодермальные образования. Обычно в каждом сегменте имеется одна пара метанефридиев. Каждая пара метанефридиев начинается в одном сегменте воронками с ресничками, открытыми в целом, от которых выделительные каналы продолжаются в следующем сегменте и открываются там наружу парными отверстиями. Метанефридии — это не только органы выделения, но и органы регулирования водного баланса в организме червей. В каналах метанефридиев происходит концентрация конечных продуктов обмена (аммиак превращается в мочевую кислоту), а освободившаяся вода снова поступает в целомическую жидкость. Это особенно важно для почвенных и наземных кольчецов (рис. 76). Помимо метанефридиев на стенах целома разбросаны специальные (хлорагенные) клетки, которые поглощают из целомической жидкости конечные продукты обмена. Много таких клеток и на стенах средней кишки. Нагруженные конечными продуктами обмена хлорагенные клетки могут выводиться через метанефридии наружу.

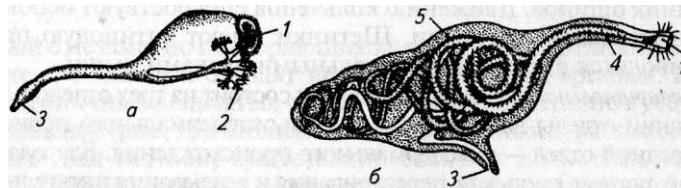


Рис. 76. Строение метанефридия кольчецов:
а — метанефридий с пламенными клетками; б — типичный метанефридий: 1 — воронка;
2 — пламенные клетки; 3 — наружное отверстие; 4 — канал органа; 5 — кровеносные со-
суды, разветвляющиеся в стенах канала

Размножение. Основная масса кольчецов размножается половым путем, но иногда они могут размножаться почкованием или делением. Кольчечные черви — раздельнополые животные, но много и гермафродитов. Развитие их проходит без метаморфоза или с метаморфозом.

КЛАСС МНОГОЩЕТИНКОВЫЕ ЧЕРВИ (*Polychaeta*)

Общая характеристика. Это главная, наиболее древняя и богатая (около 8 тыс. видов) группа кольчечных червей, которая дала начало другим классам этого типа. У представителей этого класса по бокам сегментов тела имеются параподии, снабженные многочисленными щетинками.

Передние сегменты полихет сливаются и образуют головной отдел, на котором расположены рот и органы чувств. Это раздельнополые животные. Развитие происходит с метаморфозом.

Строение и жизненные отравления. Полихеты могут быть очень мелкими, но могут достигать и довольно внушительных размеров — до 1 м и более. На головной лопасти всегда имеется пара чувствующих щупиков, которые у сидячих форм превратились в крону щупальцевидных пришлаков. На головной лопасти расположена пара осязательных щупалец — антенн. Форма тела вытянутая, туловище состоит из разного числа сегментов — от 5 до 800.

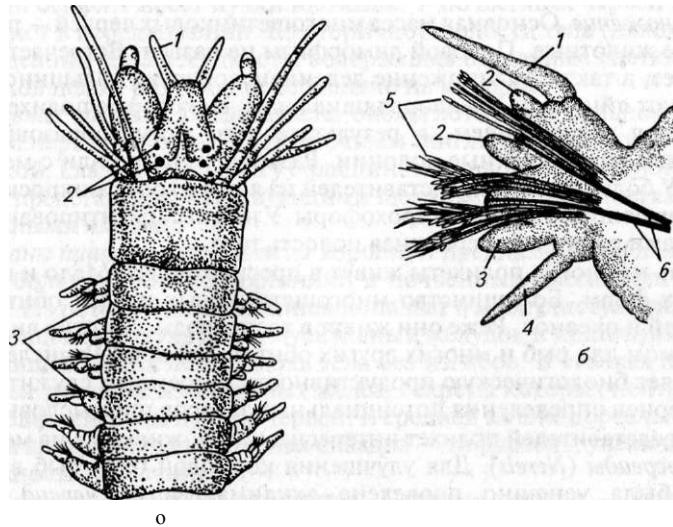


Рис. 77. Голова и параподия многощетинкового кольчечного черва нерейс:
а — голова: 1 — щупальца; 2 — глаза; 3 — параподии; б — параподия: 1 — спинной усик;
2 — спинная лопасть; 3 — брюшная лопасть; 4 — брюшной усик; 5 — щетинки; 6 — опорные щетинки

Передвигаются полихеты с помощью параподий (рис. 77), образованных основной нерасчлененной частью и двумя лопастями: спинной и брюшной. От основания спинной и брюшной лопастей каждой параподии отходит по тонкому усiku, выполняющему обонятельные и осязательные функции. Параподии соединены с телом червя подвижно и действуют по типу простого рычага. В каждой лопасти есть пучок упругих и тонких щетинок. Кроме того, в толще основной части имеется пара опорных щетинок, более толстых и длинных. С помощью параподий черви ползают по дну водоемов, у плавающих форм параподии выполняют функцию плавников. У некоторых полихет параподии частично или полностью редуцированы (роющие виды; виды, живущие в домиках).

Покровы у полихет, живущих на дне водоемов, представлены хорошо развитой кутикулой. У активно плавающих форм, живущих в домиках или зарывающихся в грунт, кутикула тонкая. Органы чувств хорошо развиты: на голове одна-две пары глаз, осязательные усики, обонятельные ямки и щупальца. У примитивных форм эпителий местами может быть ресничным.

Органы дыхания. Дышат полихеты жабрами или поверхностью тела. У большинства функцию дыхания берут на себя участки параподий.

Кровеносная система представлена спинным и брюшным сосудами, а также кольцевыми сосудами. Кровеносная система замкнута. Движение крови по телу обеспечивается сокращениями главным образом спинного сосуда. Кровь может быть окрашена в красный цвет.

Размножение. Основная масса многощетинковых червей — раздельнополые животные. Половой диморфизм не развит. Встречается партеногенез, а также размножение делением поперек. Большинство откладывают яйца, есть и живородящие виды. Некоторые полихеты размножаются почкованием, в результате чего могут образовываться временные разветвленные колонии. Развитие прямое или с метаморфозом. У большинства представителей из яиц выходят микроскопические плавающие личинки — трохофоры. У них несегментированное тело с рядами ресничек, первичная полость тела.

Очень немногие полихеты живут в пресных водах. Мало и паразитических форм. Большинство многощетинковых червей обитают на дне морей и океанов. Реже они живут в толще воды. Многие виды служат кормом для рыб и многих других обитателей вод. Их численность определяет биологическую продуктивность водоемов и служит одним из критериев определения потенциальных запасов промысловых рыб.

Из представителей полихет интересны черви, живущие на мелководье, — *нереиды* (*Nereis*). Для улучшения кормовой базы рыб в нашей стране была успешно проведена акклиматизация *нереид* (*Nereis diversicolor*) в Каспийском море, куда их завезли из Азовского. Многощетинковый червь *пескожил* (*Arenicola marina*) во множестве заселяет песчаные отмели, живя в заиленном песке и питаясь органикой; подобно дождевому черви он пропускает грунт через свой пищеварительный тракт.

У тихоокеанского многощетинкового червя *пололо* (*Eunice viridis*) период размножения сегменты задней части тела заполняются половыми продуктами. Затем эти части тела отрываются и всплывают на поверхность океана. Яйца и спермин выходят из разрывов сегментов в воду, где и происходит оплодотворение. Из зигот образуются плавающие личинки, которые развиваются во взрослых червей и опускаются на дно.

КЛАСС МАЛОЩЕТИНКОВЫЕ ЧЕРВИ (*Oligochaeta*)

Общая характеристика. К этому классу принадлежат многие водные и почвенные формы, в том числе дождевые черви. Известно более 5 тыс. видов малощетинковых червей, из которых в морских водах встречается очень небольшое их число. У малощетинковых червей нет параподий. По бокам тела у них имеются небольшие пучки щетинок (щетинок нет на передних и задних сегментах). У дождевых червей на каждом сегменте имеется по четыре пары щетинок. Головной отдел не обособлен. У большинства нет щупалец. Гермафродиты. Развитие прямое.

Строение и жизненные отправления. Цилиндрическое и сильно вытянутое тело малощетинковых червей состоит из похожих друг на друга сегментов, число которых может колебаться от 5 до 600. На переднем конце расположено ротовое отверстие, на заднем — анальное. Некоторые тропические виды могут достигать 2 м и более.

В покровах червей расположено много желез, выделяющих слизь. Каждый сегмент несет пучки щетинок. У почвенных червей щетинки участвуют в передвижении. Во вторичной полости тела (целоме) находится целомическая жидкость, содержащая отдельные клетки. У многих видов целом разделен посегментно на камеры.

Нервная система представлена окологлоточным кольцом, надглоточным нервным узлом, подглоточным ганглием и брюшной нервной цепочкой. Глаза и щупальца у большинства видов отсутствуют. Органы чувств представлены чувствующими щетинками, статоцистами и обонятельными ямками.

Органы пищеварения развиты хорошо и предназначены для пропускания больших объемов органики в почвенных массах или донного грунта. Из ротовой полости пища попадает в мускулистую глотку и затем в пищевод, оттуда в зоб, мускульный желудок и кишечник. Все органы пищеварения лежат вдоль тела без изгибов. В стенках пищевода имеются три пары известковых желез, секреты которых нейтрализуют гуминовые кислоты в пище червей. В средней кишке дорсально расположена внутренняя продольная складка — тифлозоль, увеличивающая поверхность кишечника.

Кровеносная система замкнутая. Главные сосуды — брюшной и спинной. В покровах малощетинковых червей имеется густая сеть капилляров, из которых обогащенная кислородом кровь собирается в сосуд, лежащий под брюшной нервной цепочкой. Таким образом, у малощетинковых червей, за редким исключением, дыхательная система отсутствует. В отличие от полихет у малощетинковых червей кольцевые

сосуды в области пищевода пульсируют и называются «сердцами». Кровь содержит гемоглобин, который растворен в плазме крови (у млекопитающих гемоглобин находится в эритроцитах).

Половая система. Малощетинковые черви гермафродиты, которым свойственно перекрестное оплодотворение, что и определяет сложность строения половой системы. Тело этих червей слегка уплощено и состоит из 50—250 внешне сходных сегментов. В области 32—37-го (от головной лопасти) сегментов имеется скопление одноклеточных железок, образующих кольцевидное утолщение — поясок (рис. 78). Эти клетки выделяют слизь для образования яйцевого кокона и белковую жидкость для питания зародышей.

В 10-м и 11-м сегментах тела дождевого червя располагается по паре семенников (см. рис. 75), которые прикрыты тремя парами семенных мешков. В семенных мешках созревают и накапливаются спермии. Из семенных мешков спермии поступают в мерцательные воронки семяпроводов. Семяпроводы сливаются попарно по левой и правой сторонам тела и образуют два продольных канала, открывающиеся на брюшной стороне двумя отверстиями на 15-м сегменте тела.

Рис. 78. Дождевой червь:
1 — женское половое отверстие; 2 — мужское половое отверстие; 3 — поясок

Женская половая система образована paarой мелких яичников, расположенных в 13-м сегменте. От яичников отходят два коротких яйцевода с воронками, которые открываются на брюшной стороне 14-го сегмента двумя половыми отверстиями. В 13-м сегменте яичники и воронки яйцеводов прикрываются яйцевыми мешками. Помимо этого к женской половой системе относятся две пары семяприемников, расположенные в 9-м и 10-м сегментах и открывающиеся на брюшной стороне двумя парами отверстий.

Спаривание у дождевых червей сводится к обмену спермой. В период размножения сначала все особи становятся самцами, поскольку у них развиты только семенники. Во время спаривания два червя двигаются головными концами друг к другу и соприкасаются брюшными сторонами, при этом поясок каждого червя располагается на уровне семяприемников, происходит обмен спермой. После этого черви расходятся. Затем у каждого из них на пояске, представляющем собой желобистое утолщение кожи нескольких определенных сегментов, образу-

ется муфта. Эта муфта сокращениями мускулатуры тела сдвигается к головному концу черва. Когда муфта проходит мимо 14-го сегмента, в нее откладываются яйца, а из 9—10-го сегментов туда попадают спермин. Происходит перекрестное оплодотворение. Наконец, муфта сбрасывается через головную часть черва, края ее смыкаются и она становится яйцевым коконом, в котором и происходит развитие зародышей. Кокон дождевых червей по форме напоминает лимон желто-бурого цвета. Диаметр кокона составляет около 4—5 мм.

Из яиц, развивающихся в коконе, выходит сформировавшийся червячок. У низших олигохет в коконе может быть несколько яиц. У высших, как правило, одно яйцо, несколько бывает редко. Помимо полового размножения у олигохет встречается и бесполое размножение: тело черва поперечно делится на две части, недостающие части регенерируют. У дождевых червей хорошо выражена способность к регенерации, причем легко восстанавливается задний конец тела, головной отдел восстанавливается редко.

Дождевые черви ведут активную и полезную для почвы деятельность. Их ходы способствуют аэрации почвы, по ним проникает вода, черви разрыхляют почву и удобряют ее остатками своей жизнедеятельности, богатыми гумусными кислотами. При благоприятных условиях на 1 м² площади луга обитают 50—100 дождевых червей. Существуют около 200 видов дождевых червей. Мелкие (менее 1 см) беловатые кольчатые черви семейства энхитреид (*Enchytreidae*) чаще всего встречаются в почве, но есть виды, обитающие в пресных водоемах. Почвенных энхитреид насчитывают около 400 видов. Плотность энхитреид в почве может составлять 150 тыс. на 1 м². Их легко разводить в искусственных условиях в качестве корма для рыб. Питаются эти черви органическими остатками.

Множество олигохет живет на дне водоемов, питаясь органикой. У некоторых видов водных малощетинковых червей наблюдается почкование. В этом случае образуются сложные цепи почкующихся осо-бей. Водные малощетинковые черви служат кормом для обитателей водоемов, а наземные — для многих наземных обитателей, в том числе для множества позвоночных.

Дождевые черви и биогумус. Дождевой червь ведет ночной образ жизни; в рыхлом грунте он делает норки, уплотняя землю. В плотном грунте черви вынужден делать норки путем выедания почвы. Часть ее, пропущенную через пищеварительный тракт, червь выбрасывает наружу. Разыскивая пищу, червь заднюю часть тела держит в норке. Найдя пищу (растительные остатки), затаскивает ее в норку и затем поедает. Черви не выносят сухости, ибо испытывают кислородное голодание. В проточной богатой кислородом воде могут жить несколько дней.

Идея промышленного культивирования дождевых червей принадлежит американскому врачу Барретту, который в 1947 г. опубликовал результаты своих опытов. В 1975 г. в Италии была создана промышленная технология. Только в США работают тысячи специализированных производств по переработке навоза в биогумус с помощью червей. Био-

гумус (переработанный дождевыми червями и другими организмами подстилочный навоз) — высокооценное органическое удобрение.

Поглощая вместе с почвой огромное количество распадающихся растительных остатков, микробов, грибов, водорослей, простейших, нематод и других почвенных организмов, дождевые черви переваривают их и выделяют с копролитами (копрос — испражнение, литое — камень) большое количество собственной кишечной микрофлоры, ферментов, витаминов и других биологически активных веществ. Биологически активные вещества обладают антимикробными свойствами, препятствующими развитию патогенной (болезнетворной) микрофлоры, гнилостных процессов, выделению зловонных газов, обеззаражающими почву и придающими ей приятный запах земли.

В процессе переваривания растительных остатков в пищеварительном тракте червей формируются *гумусные вещества*, которые отличаются по химическому составу от гумуса, образующегося в почве при участии только микроорганизмов. В кишечнике червей развиваются процессы полимеризации низкомолекулярных продуктов распада органических веществ и формируются молекулы *гуминовых кислот* и *фульвокислот* (последние неблагоприятны, и чем их меньше, тем ценнее гумус). Эти кислоты образуют с металлами соли — *гуматы* и *фульваты*.

Гуматы лития, калия и натрия растворимы и легко вымываются водой; они представляют собой наиболее ценную часть гумуса, которая легко доступна для растений. Гуматы кальция, магния, кремния и тяжелых металлов нерастворимы и составляют ту часть гумуса, которую можно назвать «консервами» почвенного плодородия. Они накапливались в черноземах весь послеледниковый период. Эти гуматы могут растворяться под влиянием ферментов корневой системы растений, но лишь в таких количествах, которые удовлетворяют их потребности. Эти соли не подвержены гидролизу, но оказывают большое влияние на создание ценной, прочной и пористой структуры, не подверженной влиянию эрозии.

Следует отметить, что гуматы тяжелых металлов еще более устойчивы к гидролизу ферментами корневой системы растений и практически не усваиваются ими. Это одно из главных свойств гумуса — связывание в почве тяжелых металлов и предохранение живого на Земле от их токсического действия, в том числе от действия тяжелых радионуклидов. Чем больше в почве гумуса, тем сильнее выражено это ее свойство. Поэтому пищевая продукция, выращенная на высокогумусных почвах, является экологически безопасной для животных и человека.

Деятельность дождевых червей замедляет вымывание из почвы подвижных питательных элементов, закрепляет тяжелые металлы и препятствует развитию водной и ветровой эрозии. В копролитах червей естественных популяций содержится 11—15 % гумуса в расчете на сухое вещество. В естественных местах обитания плотность популяции дождевых червей варьирует от 100 до 600 и более особей на 1 м². За летний период популяция из 50 червей в пахотном слое почвы на 1 м² прокладывает 1 км ходов и выделяет на поверхность копролиты слоем около 3 мм. Еще больше их остается в толще почвы. Каждый червь пропуска-

ст через пищеварительный тракт за сутки столько почвы, сколько весит он сам (средняя масса червя в почвенном слое около 0,5 г). В средней полосе активная деятельность червей продолжается 200 дней; за этот период, следовательно, каждый червь пропустит через себя около 100 г субстрата, т. е. на 1 га приходится 50 т переработанной почвы.

Особенно важным условием для жизни червей является достаточная влажность субстрата. Влажность почвы ниже 30—35 % тормозит из развитие, а при влажности 20—22 % они погибают в течение недели. Максимальную массу и наибольшее число коконов получают при 70—85 % влажности субстрата (столько же воды содержится и в теле червя).

В кислой среде (рН 5) или сильно щелочной (рН более 9) черви погибают в течение недели. Оптимум для дождевых червей — нейтральная среда (рН около 7). Велика потребность червей в азоте: в богатой азотом почве их численность растет. Вот почему их много в навозе и на пастбище. Концентрация растворимых солей более 0,5 % смертельна для них. Такие соли, как углекислый кальций, углекислое железо, сернокислый алюминий и хлористое железо безвредны даже в больших количествах. Итак, оптимальными для развития дождевых червей условиями являются следующие: температура 15—22 °С, влажность — 60-75 % и рН 7,3-7,6.

Дождевые черви живут в верхних слоях почвы. Они не уходят в нижние слои на спячку до тех пор, пока земля не промерзнет на глубину 5—6 см и не появится снежный покров. При длительной оттепели черви могут выползать даже на снег. Обычно при 5 °С черви перестают питаться, освобождают кишечник и уползают в нижние слои почвы, где оцепеневают. Просыпаются они под воздействием вешних вод и теплого воздуха, проникающих к ним в норки.

Дождевые черви очень плодовиты. Откладка коконов происходит с весны до начала лета и затем осенью до ноября. За лето червь откладывает по 18—24 кокона, в каждом из которых по 1—24 яйца. Через 2—3 нед из яиц выплываются молодые особи, которые по прошествии 7—12 нед уже сами способны размножаться. Благоприятные условия жизни ускоряют половое созревание молодых червей. Взрослые черви могут жить до 10 и даже 15 лет, их длина достигает десятков сантиметров, а масса — до 10 г. Молодые черви при достижении половой зрелости весят до 1 г. Крупные черви являются самой ценной частью популяции, но из-за несовершенной технологии обработки почвы именно эта часть дождевых червей погибает в пахотном слое.

Калифорнийский червь был выведен в 1959 г. в США. Этот червь отличается от своих диких сородичей тем, что обладает способностью размножаться в наземных культуваторах без всяких построек или теплиц. Калифорнийский червь в теплом климате дает 16—18-кратное воспроизводство за цикл культивирования под открытым небом и 512-кратное в условиях закрытых теплиц (дикие сородичи дают лишь 4—6-кратное воспроизводство). Калифорнийский червь стал предметом экспорт-импорта вместе с технологией его культивирования.

В нашей стране первые работы по культивированию калифорнийского червя начались лишь в 1984 г. под руководством А. М. Игонина. Им были разработаны звенья технологического процесса производства биогумуса с помощью калифорнийского червя, а также, что может быть самым главным, с использованием местных дождевых (навозных) червей. После переработки 1 т компостированного навоза получают 0,5 т гумусного удобрения 50 %-ной влажности и 6—10 кг живых червей.

КЛАСС ПИЯВКИ (*Hirudinea*)

Общая характеристика. Пиявкам свойственно своеобразное кольчатое строение. Их тело уплощено и не имеет четко выраженного головного отдела. Наружная кольчатость пиявок не соответствует более крупной внутренней сегментации тела. Сегментация тела однородная (гомономная). Каждому истинному сегменту соответствует 3—5 наружных колец. Тело пиявок состоит из 30—33 сегментов. Это придает им большую гибкость и позволяет вести активный образ жизни. Щетинки на теле отсутствуют. У большинства представителей этого класса имеются присоски: передняя и задняя. Передняя присоска окружает рот. Анальное отверстие находится над задней присоской.

Известно около 400 видов пиявок, живущих в основном в пресных водоемах и являющихся эктопаразитами беспозвоночных и позвоночных животных. Большинство пиявок питаются кровью разных животных, но на своих жертвах остаются недолго. В основном же паразиты ведут свободный образ жизни. Много пиявок, которые не сосут кровь, а являются хищниками. Есть сухопутные виды, распространенные в тропиках. В тропических лесах живут древесные и почвенные пиявки, нападающие на теплокровных животных и человека. В фауне нашей страны встречается 50 видов пресноводных пиявок.

Строение и жизненные отправления. Строение пиявок отвечает полу-паразитическому образу жизни, который они ведут (рис. 79).

Покровы представлены плотной кутикулой. Под ней лежит богатый железистыми клетками однослойный эпителий, образующий кутикулу. У основания эпителиального слоя разбросаны пигментные клетки, придающие пиявкам соответствующую окраску.

Мускулатура развита очень хорошо: кожно-мускульный мешок состоит из трех слоев мышечных волокон. Полость тела частично редуцирована и представлена системой лакун, заполненных паренхимой. Плотные покровы и паренхима защищают тело от высыхания и позволяют пиявкам длительное время пребывать на сушке, совершая миграции.

Нервная система. У пиявок имеется брюшная нервная цепочка. Глаза, если они есть, примитивны; в покровах располагаются чувствующие клетки и нервные окончания.

Дышат пиявки через покровы тела, но у некоторых видов имеются жабры.

Пищеварительная система хорошо приспособлена к образу жизни пиявок как паразитов. В ротовой полости у части видов имеются три

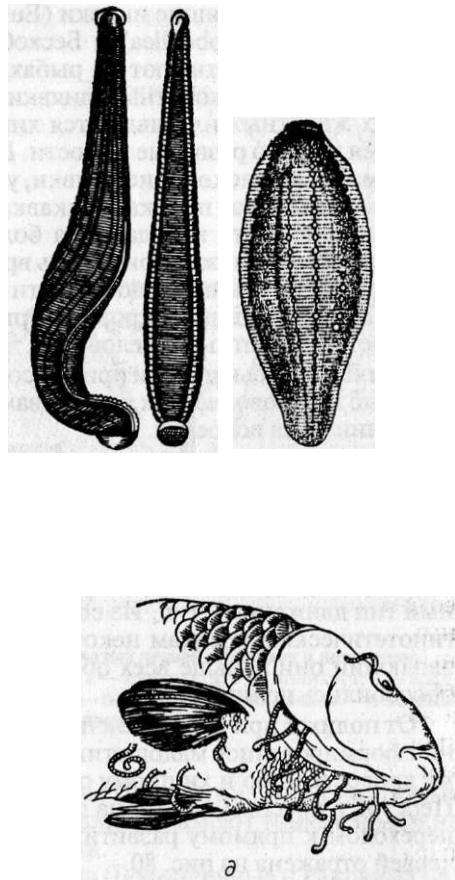
Рис. 79. Виды пиявок:
 а, б — медицинская пиявка *Hirudo medicinalis* (со спинной и брюшной стороны); в — клепсина; г — ложноконская пиявка; д — рыбья пиявка *Piscicola geometra*

челюсти со множеством зубчиков (челюстные пиявки), у других пиявок есть хоботок, с помощью которого черви внедряются в покровы своих жертв (хоботные пиявки). Глотка пиявок выполняет функции сосущего аппарата, в который открываются протоки слюнных желез. У кровососущих пиявок (медицинская пиявка) в слюне содержится белковое вещество, препятствующее свертыванию крови в кишечнике червей, — гирудин. Передняя кишка имеет карманообразные боковые выросты, позволяющие сделать значительный запас крови: медицинская пиявка делает запас на 2—3 мес жизни, поскольку кровь долго сохраняется под действием гирудина в свежем виде. Процессы переваривания происходят в энтодермальном среднем отделе кишечника, который превращается в объемистый желудок с карманами.

Органы выделения — метанефридии.

Кровеносная система развита только у низших пиявок и частично у хоботных. У челюстных пиявок кровеносная система редуцируется, а ее роль выполняет лакунарная система целомического происхождения.

Половая система. Пиявки гермафродиты. Размножаются только половым путем весной около водоемов в сырых местах. Оплодотворение внутреннее и перекрестное. У пиявок, как и у малощетинковых червей, на 9—11-м сегментах находятся поясок и железы, выделяющие слизистые муфты для образования коконов. Коконы, похожие на мелкие желеуди, пиявки откладывают на землю. Развитие прямое и в коконе длится около 5 нед. Живут пиявки до 20 лет.



Подкласс Настоящие пиявки (*Euchirudinea*) делится на два отряда: Хоботные (*Rhynchobdellea*) и Бесхоботные (*Arhynchobdellea*). Хоботные пиявки паразитируют на рыбах, птицах, лягушках, моллюсках и ракообразных. Бесхоботные пиявки паразитируют только на позвоночных животных или являются хищниками. У них нет хоботка, но имеются хорошо развитые челюсти. Наиболее часто встречаются большая и малая ложноконские пиявки, улитковая пиявка, в южных регионах — медицинская пиявка, в Закавказье — конская пиявка. В природе пиявки чаще всего нападают на больных и ослабленных животных. Конская пиявка может причинить вред лошадям и скоту: во время водопоя из естественных водоемов эти пиявки проникают в носоглотку, гортань и могут вызвать кровопотери и удушье. Медицинская пиявка нападает на животных и человека.

Птицы и рыбы пиявки при массовом заселении могут вызывать гибель рыб в рыбоводческих хозяйствах и водоплавающей птицы при содержании ее на водоемах.

ФИЛОГЕНИЯ И ЭКОЛОГИЧЕСКАЯ РАДИАЦИЯ КОЛЬЧАТЫХ ЧЕРВЕЙ

В разных классах кольчатых червей есть признаки, которые свидетельствуют о родстве аннелид с низшими червями: первичная полость тела, протонефридии у части кольчатых червей и их личинок, ресничный тип движения и т. п. Из современных аннелид наиболее близки к гипотетическим предкам некоторые архианнелиды. На раннем этапе эволюции они раньше всех обособились от других аннелид. Позднее обособились полихеты.

От полихет при переходе к пресноводному и наземному образу жизни сформировались малощетинковые черви. При переходе к активному кровососанию и хищному образу жизни обособился класс пиявок. Переход в пресные воды и на сушу пиявок и олигохет сопровождался переходом к прямому развитию. Экологическая радиация кольчатых червей отражена на рис. 80.

ТИП МОЛЛЮСКИ (*Mollusca*)

Общая характеристика. Моллюски, или мягкотельые, — вторичнополостные животные с несегментированным телом, в большинстве случаев заключенным в раковину. Известно более 113 тыс. видов моллюсков, живущих в морских и пресных водах и на суше. Большинство водных моллюсков — обитатели дна. Это животные, ведущие начало от кольчатых червей. Тело моллюсков состоит из трех отделов: головы, туловища и ноги. Важнейшей их особенностью является наличие мантии — складки кожи, свешивающейся со спины, с разнообразными и многочисленными железами, которые вырабатывают секреты, используемые при построении раковины. Между телом и мантией образуется мантийная

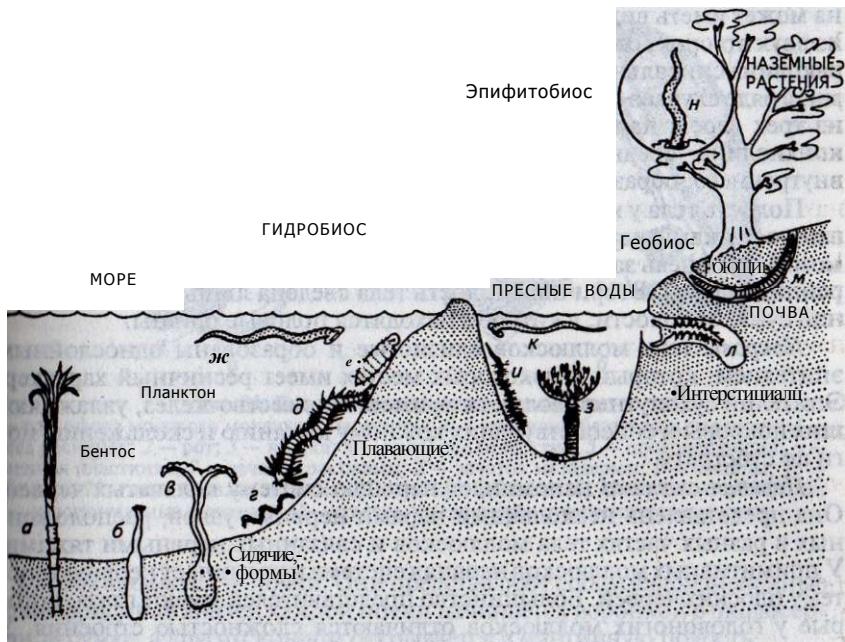


Рис. 80. Морфоэкологическая эволюция кольчатых червей и близких к ним типов:
а — погонофора; б — силиконкулида; в — бонелля; г — олигохета; д — полихета; е — архиннелида; ж — пиявка морская; з — полихета; и — олигохета; к — пиявка; л — пещерная полихета; м — дождевой червь; и — древесная пиявка

полость, в которой расположены жабры (или легкие — у сухопутных форм) и некоторые органы чувств. Моллюски имеют тонкие и мягкие покровы, богатые слизистыми железами.

Тип Моллюски включает два подтипа (Боконервные и Раковинные) и несколько классов, из которых наиболее распространены и представляют интерес три класса: Двустворчатые моллюски (Bivalvia), Брюхоногие моллюски (Gastropoda) и Головоногие моллюски (Cephalopoda), относящиеся к подтипу Раковинные (Conchifera).

Строение и жизненные направления. Размеры и форма тела моллюсков весьма разнообразны, что связано с особенностями их сред обитания. У части моллюсков тело имеет двустороннюю симметрию, но у многих, заключенных в спиральную раковину, тело асимметрично. Большая часть внутренних органов расположена в туловище. На голове находятся органы чувств, а внутри заключены крупные головные нервные узлы. У многих моллюсков голова не обособлена от тела. Нога служит для передвижения животного.

У большинства моллюсков имеется известковая раковина, образующаяся за счет выделений мантии. Раковина выполняет защитные функции, к ней прикрепляются мускулы и некоторые органы. Ракови-

на может иметь вид колпачка; у некоторых моллюсков она образована из двух створок, соединенных зубчатым замком или связкой. У других раковина спирально закручена или представлена несколькими щитками; в ряде случаев она может подвергаться редукции. Раковина состоит из трех слоев: наружного, построенного из органического вещества конхиолина, среднего, сложенного из известковых образований, и внутреннего, образованного тонким слоем перламутра.

Полость тела у моллюсков смешанная — *миксоцель*, так как образована остатками первичной полости и сильно редуцированным целомом. Миксоцель заполнен паренхимой, в которой расположены внутренние органы. Вторичная полость тела сведена лишь к околосердечной сумке и полости, в которой находятся половые органы.

Покровы тела моллюсков слизистые и образованы однослойным эпителием, который в некоторых местах имеет ресничный характер. Эпителий сухопутных моллюсков имеет множество желез, увлажняющих покровы и способствующих кожному дыханию и скольжению ноги по субстрату.

Нервная система напоминает нервную систему кольчатых червей. Она представлена несколькими парами нервных узлов, расположенных в разных частях тела моллюсков и связанных нервными тяжами. У примитивных моллюсков нервная система напоминает нервную систему плоских червей. Большинство представителей имеют глаза, которые у головоногих моллюсков отличаются сложностью строения. У мягкотелых есть органы осязания, органы химического чувства и равновесия (статоцисты).

Мускулатура слагается из гладких мышечных волокон, что обуславливает замедленное движение тела. У ведущих активный образ жизни головоногих моллюсков есть поперечно-полосатые мышцы, особенно хорошо развитые в ноге.

Органы дыхания у большинства водных видов представлены жабрами, расположенными в мантийной полости. Сами жабры — это видоизмененные участки мантии. Жабры имеют вид лепестков, расположенных на оси жабры. Все сухопутные и вторичноводные формы дышат легкими, которые также образованы участками мантии, и снабжены большим количеством кровеносных сосудов.

Кровеносная система незамкнутая. В околосердечной сумке (целоме) расположено сердце, имеющее желудочек и одно или несколько предсердий. От желудочка отходят артерии, разносящие кровь по всему телу. Кровь изливается в систему лакун полости тела. Отсюда кровь заасасывается в венозные сосуды и по ним поступает в жабры или легкое. Окисленная кровь возвращается в сердце по сосудам.

Органы пищеварения представлены ротовым отверстием, глоткой и пищеводом, из которого пища попадает в желудок. Для большинства видов характерно наличие в глотке аппарата для размельчения пищи — терки; нередко могут быть развиты и хитиновые челюсти. За желудком начинается кишечник, в который впадает проток печени. Кишечник оканчивается анальным отверстием.

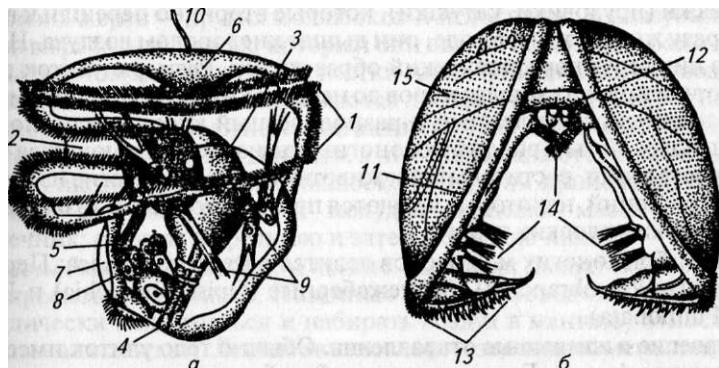


Рис. 81. Личинки двустворчатых моллюсков:
и — трохофора морского моллюска-парусника; б — глохидий беззубок и перловиц; 1 — нога с ресничками; 2 — рот; 3 — кишка; 4 — анальное отверстие; 5 — протонефридий; 6 — теменная пластинка; 7 — зародыш раковины; 8 — зародыш мезодермы; 9 — мышцы; 10 — реснички; 11 — личиночная раковина; 12 — мускул-замыкатель; 13 — зубцы с зубчиками; 14 — чувствительные щетинки; 15 — железы, выделяющие прикрепительные нити

Органы выделения — почки. Они представляют собой видоизмененные метанефриции. Каналец каждой почки начинается воронкой в околосердечной сумке, а другим концом открывается в мантийную полость.

Органы размножения находятся в целоме. Протоки половых желез открываются либо в мантийную полость, либо в протоки почек.

Большинство моллюсков раздельнополы. Оплодотворение яйце-клеток осуществляется в мантийной полости или внутри материнского организма. Развитие моллюсков происходит с метаморфозом или без него. Живущие в воде моллюски откладывают яйца в воду, а сухопутные формы — в почву и на ее поверхность. Есть живородящие виды. У низших форм из яйца развивается трохофорная личинка. У болыпинства моллюсков из яйца выходит личинка парусник, имеющая пучок ресничек и парус с лопастями, несущими реснички. Парус служит для движения личинки (рис. 81). У части морских, у большинства пресноводных и у всех сухопутных моллюсков развитие прямое.

КЛАСС БРЮХОНОГИЕ МОЛЛЮСКИ (Gastropoda)

Общая характеристика. У брюхоногих моллюсков, или улиток, нога имеет широкую подошву; такая нога используется этими моллюсками для ползания. При наличии раковины последняя спирально завита, что придает телу моллюска асимметричную форму. Голова имеет одну-две пары щупалец, у многих хорошо развиты глаза. Дышат брюхоногие моллюски жабрами или легкими. Первично брюхоногие моллюски — обитатели моря, но некоторые из них в процессе эволюции приспособились к жизни в пресных водоемах и на суше. Имеются пресноводные

моллюски (прудовики, катушки), которые вторично перешли к водному образу жизни: живя в воде, они дышат кислородом воздуха. Немногие из них ведут паразитический образ жизни. Размеры улиток варьируют от нескольких миллиметров до нескольких десятков сантиметров. Это самый многочисленный и разнообразный класс моллюсков. Известно более 90 тыс. видов брюхоногих моллюсков, питающихся гниющими остатками, растительной и животной пищей. Многие из них вредители растений, некоторые являются промежуточными хозяевами паразитических плоских червей.

Класс брюхоногих моллюсков делится на три подкласса: Переднезаберные (*Prosobranchia*), Заднезаберные (*Opisthobranchia*) и Легочные (*Pulmonata*).

Строение и жизненные отправления. Обычно тело улиток имеет разнообразную форму. Голова хорошо обособлена от туловища, которое образует сверху вырост в виде внутренностного мешка, закрученного спирально. Одна-две пары щупалец на голове несут осознательные функции и способны втягиваться. Хорошо развиты глаза, которые могут быть расположены на вершинах щупалец. Есть органы химического чувства и равновесия — статоцисты, расположенные в ноге. Передвигаются брюхоногие моллюски скольжением за счет волнобразных изгибов широкой и плоской подошвы ноги, представляющей мускулистый брюшной вырост.

Цельные, обычно спирально закрученные раковины имеют разнообразные форму и расцветку. У части плавающих брюхоногих раковина может быть рециклирована в той или иной степени. Нет ее и у наземных слизней, ведущих сумеречный и ночной образ жизни. В раковине размещено только туловище моллюска, но при опасности в раковину втягивается все тело. Раковина состоит из наружного тонкого органического слоя, под которым расположен минеральный слой известковой природы. Углекислая известь извлекается моллюсками из воды и пищи. Вещество для раковины выделяется известковыми железами мантии. У некоторых моллюсков раковина имеет еще и третий слой — внутренний перламутровый, или эмалевый, разной окраски. Мантийная полость расположена в нижних витках раковины. В мантийную полость открываются анальное отверстие, мочеточники, иногда и проток половых органов. У водных видов в мантийной полости расположены жабры. У наземных и вторичноводных моллюсков мантийная полость стала легким, которое открывается наружу специальным дыхательным отверстием.

Органы пищеварения. Брюхоногие моллюски питаются растительной пищей, детритом или являются хищниками. Ротовое отверстие расположено на нижней стороне головы и ведет в глотку, которая имеет роговые челюсти и мускулистый валик — язык с теркой (радулой), имеющей вид пластинки с мелкими зубчиками. С ее помощью моллюск отделяет частицы пищи или субстрата. В глотку впадают протоки слюнных желез. У некоторых хищных форм в слюне содержится серная кислота, с помощью которой моллюски растворяют раковины или пан-

цири своих жертв — других моллюсков и иглокожих. У ядовитых моллюсков вырабатывается яд, который они вводят особыми зубами в тело жертвы и потом поедают ее. Из глотки пища попадает в пищевод и затем в желудок, в который открываются протоки печени. Секрет печени характеризуется амилолитической активностью. Печень способна всасывать часть питательных веществ, в ней откладываются резервные липиды и гликоген. У низших моллюсков в печени происходит внутриклеточное пищеварение. Из желудка пищевая масса поступает в кишечник: сначала в среднюю и затем в заднюю кишку.

Органы дыхания — жабры и легкие. Жабрами дышат все морские и часть пресноводных видов. Наземные и вторичноводные (вынуждены периодически подниматься и набирать воздух в мантию) относятся к легочным моллюскам. Легкие моллюсков — это видоизмененная мантийная полость, стенки которой пронизаны сетью кровеносных сосудов. Воздух поступает в легкое через особое дыхальце, которое при погружении в воду закрывается. У моллюсков имеется и кожное дыхание.

Кровеносная система незамкнутая; ее образуют сердце, находящееся в околосердечной сумке, сосуды и лакуны. Кровь обычно бесцветна и содержит амебоциты. В околосердечную сумку открываются воронки двух почек; мочеточники выводят мочу в мантийную полость сбоку от анального отверстия.

Органы размножения. Большинство морских брюхоногих моллюсков раздельнополы, а наземные и многие пресноводные — гермафродиты. Половые железы непарные. Оплодотворение яйцеклеток осуществляется в материнском организме. Часто оплодотворенные яйца окружаются оболочками или студенистыми коконами и соединяются в кладки. У гермафродитов половая система устроена сложно. Так, у виноградной улитки имеется гермафродитная железа,рабатывающая яйцеклетки и спермии. Спаривание сводится к обмену спермой (подобно дождевым червям), которая поступает в семяприемники другой особи. Поэтому оплодотворение у них всегда перекрестное. Развитие протекает без стадии личинки (прямое) или с метаморфозом, реже наблюдается живорождение.

Брюхоногие моллюски участвуют в круговороте веществ в водоемах и на суше, потребляя органические остатки и перерабатывая их. Многие моллюски служат кормом для водных позвоночных. Морские трубачи являются источником черного и розового жемчуга, высоко ценящегося на мировом рынке.

В морях и океанах моллюски живут на разных глубинах. Сухопутные виды легко переносят разные климатические условия, что обусловлено их способностью впадать в спячку — зимнюю (на севере) или летнюю и зимнюю (южные регионы). При этом улитка заползает в почву, втягивается целиком в раковину и заклеивает в нее вход.

Некоторых брюхоногих моллюсков используют в пищу и они являются объектом промысла (в Черном море обитает моллюск морское блюдечко — *Patella*). Встречающаяся в нашей стране виноградная улитка

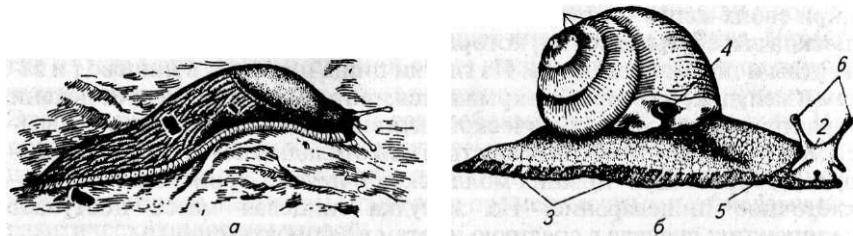


Рис. 82. Представители наземных брюхоногих моллюсков:
а — слизень оранжево-желтый; б — виноградная улитка; 1 — раковина; 2 — голова с двумя парами щупалец; 3 — нога; 4 — дыхательное отверстие; 5 — половое отверстие; 6 — глаза

(*Helix pomatia*) относится к числу вредителей виноградной лозы, а в ряде стран этих улиток употребляют в пищу, для чего специально разводят их.

Многочисленные многоядные слизни наносят вред различным растениям, поедая листья, ягоды и клубни. Тело слизней вытянуто и лишено раковины, правда, у некоторых видов под кожей имеются остатки раковины. Особый ущерб наносят полевым и кормовым культурам полевой слизень (*Agriolimax agrestis*), крупный окаймленный слизень (*Arion circumscriptus*) и другие (рис. 82). Обычно это ночные животные, днем прячущиеся в укрытиях. Слизни — гермафродиты. Дают за летний период несколько кладок по 9—50 яиц в каждой. Зимуют на стадии яйца и взрослой особи. Продолжительность жизни слизней — до 3 лет.

Наземные и пресноводные брюхоногие моллюски являются промежуточными хозяевами плоских паразитических червей — сосальщиков. Это малый прудовик (*Limnaea truncatula*), обыкновенный прудовик (*L. stagnalis*), битиния (*Bithynia leachii*), хелицелла (*Helicella candidula*) и многие другие (рис. 83). Роль этих брюхоногих моллюсков весьма вели-

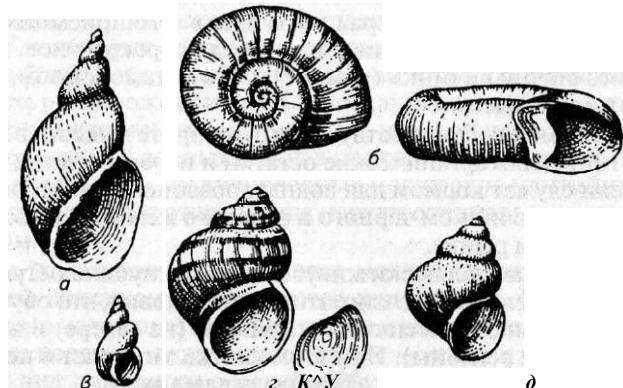


Рис. 83. Раковины различных пресноводных брюхоногих моллюсков:
а — прудовик обыкновенный (*Limnaea stagnalis*); б — катушка обыкновенная (*Planorbis corneus*); в — малый прудовик (*Limnaea truncatula*); г — лужанка живородящая; д — битиния (*Bithynia leachii*)

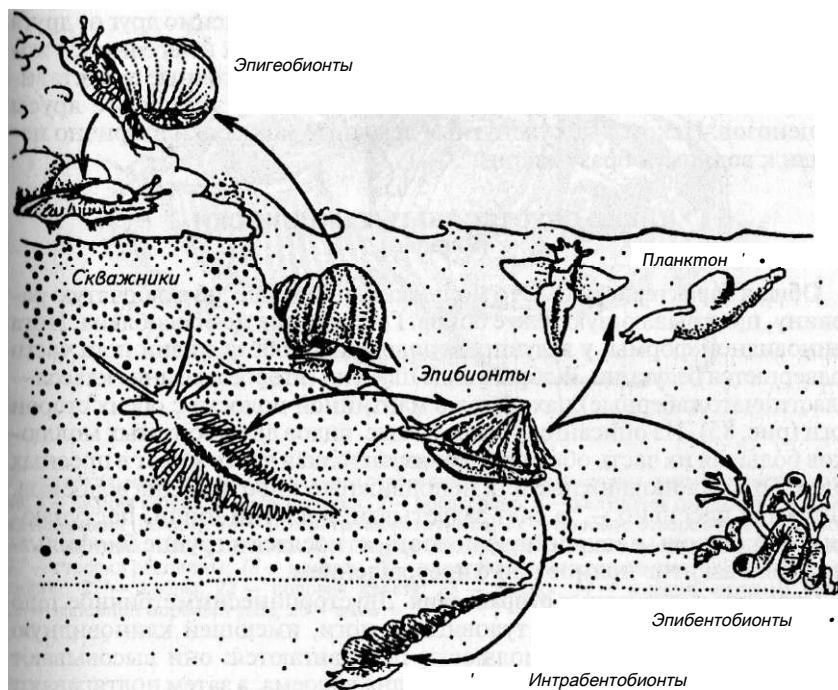


Рис. 84. Экологическая радиация брюхоногих моллюсков

ка в распространении гельминтных заболеваний, так как эти мягкотелые весьма многочисленны в природе. Например, малый прудовик живет во всех водоемах, начиная с маленьких луж и кончая крупными водоемами и достигает численности нескольких миллионов на гектар. Моллюски зимуют, зарываясь в грунт дна водоемов. При этом более половины из них могут быть поражены личинками фасциол. Моллюски служат косвенным индикатором загрязнения водоемов.

Филогения брюхоногих моллюсков. Считают, что предками брюхоногих моллюсков были небольшие примитивные гастроподы с раковиной в виде колпачка. В процессе эволюции увеличивались размеры моллюсков и раковина спирально закручивалась. Легочные и заднежаберные моллюски представляют наиболее продвинутые ветви эволюции. Экологическое разнообразие гастропод превосходит экологическое разнообразие других классов моллюсков (рис. 84).

В процессе эволюции происходило совершенствование специализации ползающих по дну, а также образование роющих видов с раковиной в виде буравчика со множеством оборотов спирали. Особый интерес представляет кораллобионтная группа моллюсков, приспособившихся к жизни на коралловых полипах. К плавающему образу жизни

перешли киленогие и крылоногие моллюски. Независимо друг от друга перешли к жизни на сушу часть гребенчатожаберных и легочных, у которых развитие стало протекать без метаморфоза. Легочные представители расселились по всем географическим зонам, заняли все ярусы биоценозов. Некоторые сухопутные легочные моллюски вторично перешли к водному образу жизни.

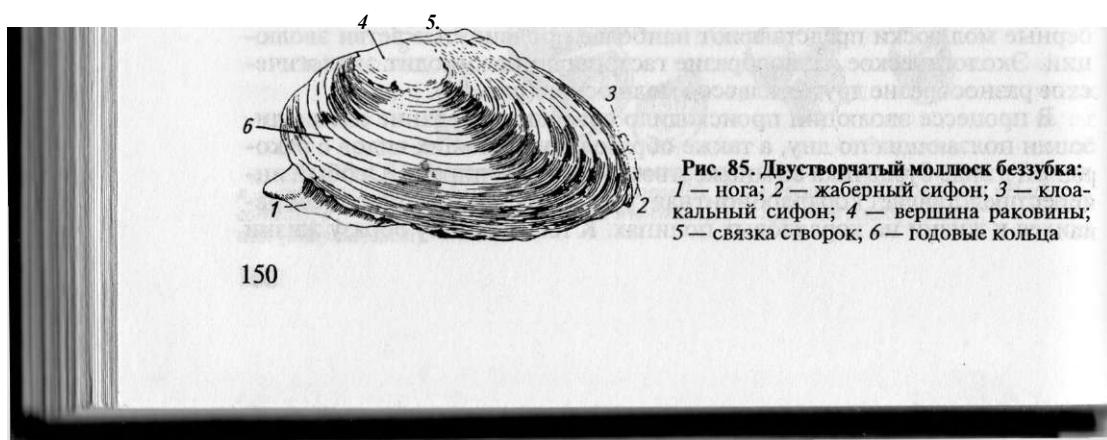
КЛАСС ДВУСТВОРЧАТЫЕ МОЛЛЮСКИ (Bivalvia)

Общая характеристика. Это моллюски, имеющие двусторчатую раковину, прикрывающую тело с боков. Голова у них обособлена. Нога клиновидной формы, у ведущих неподвижный образ жизни нога часто подвергается редукции. Жабры в виде пластин (второе название класса Пластинчатожаберные) находятся в мантийной полости с обеих сторон ноги (рис. 85). Из описанных более 20 тыс. видов двусторчатых моллюсков большая их часть обитает в морях, меньшая часть живет в пресных водах. Это малоподвижные или неподвижные животные дна водоемов. Питаются пассивно — за счет захватывания частиц пищи, поступающих с током воды в мантийную полость, относятся к группе биофильтраторов. Развитие прямое или с превращением.

Строение и жизненные отравления. Двустороннесимметричное тело этих животных состоит из туловища и ноги, имеющей клиновидную форму. С помощью ноги моллюски передвигаются: они высасывают ногу из раковины, зарывают ее в грунт дна водоема, а затем подтягивают к ней тело. У неподвижных видов нога в разной степени редуцирована.

Покровы богаты различными железами, вырабатывающими слизь, кислоту для разрушения известковых скал, материал для прикрепления к субстрату и ряд других веществ. Мантия в виде двух складок свешивается с боков тела. Между телом и мантией образуется мантийная полость, в которой расположены жабры и нога, в нее открываются задняя кишечная, мочевые и половые протоки.

В мантийную полость вода поступает через жаберный сифон, омывает жабры и удаляется через выводной (колоакальный) сифон. Движение воды обеспечивает реснички мерцательного эпителия, который покрывает мантию, жабры и сифоны. Вода приносит в мантийную полость пищевые частицы и кислород.



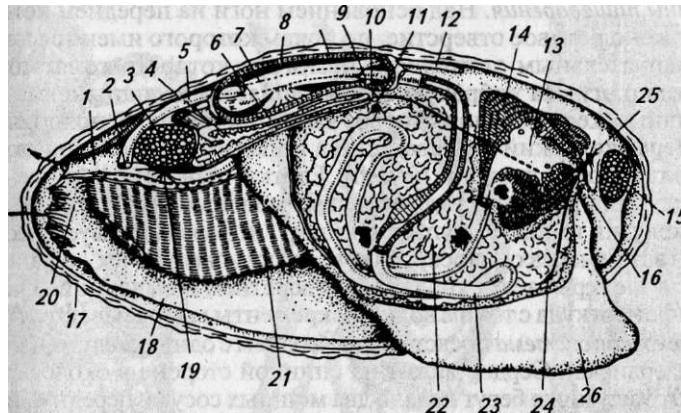


Рис. 86. Анатомия беззубки:
 / — клоакальный сифон; 2 — анальное отверстие; 3 — задний мускул-замыкатель; 4 — почки; 5 — задняя кишка; 6 — мочеточник; 7 — предсердие; 8 — желудочек сердца; Р — воронка нефридия (почки); 10 — отверстие мочеточника; 11 — половое отверстие; 12 — аорта; 13 — желудок; 14 — печень; 15 — передний мускул-замыкатель створок; 16 — рот; 17 — створка раковины; 18 — мантия; 19 — висцеральный нервный узел; 20 — жаберный сифон; 21 — жабры; 22 — половая железа; 23 — кишка; 24 — ножной нервный узел; 25 — головной нервный узел; 26 — нога

Две створки раковины соединены на вершине эластичной связкой (лигаментом) или замком, расположенными по верхнему краю створок. У большинства двустворчатых моллюсков створки имеют одинаковые размеры и форму, но у неподвижных форм они могут различаться. У живого моллюска створки раковины способны раскрываться и замыкаться. Захлопывание (закрытие) створок обеспечивают мускулы-замыкатели, связывающие обе створки. При их сокращении створки закрываются, а при расслаблении открываются за счет рогового эластичного лигамента, который действует как пружина.

Нарастание раковинных створок происходит по наружному их краю **за** счет секретов желез наружного эпителия мантии. Зимой моллюски 11 практически не растут, и поэтому на створках образуются две годичные полосы (летняя и зимняя), по которым легко определить возраст моллюска. Раковина имеет три слоя: органический, фарфоровидный (из углекислого кальция) и перламутровый.

Нервная система состоит из трех пар нервных ганглиев, расположенных над глоткой, в ноге и в задней части туловища (рис. 86) и связанных комиссарами. Органы чувств развиты слабо. В покровах разбросаны чувствующие клетки, на жабрах есть хеморецепторы, в ноге расположены органы равновесия — статоцисты. У некоторых видов по краям мантии разбросаны многочисленные глазки. Головные щупальца и глаза отсутствуют.

Органы пищеварения. Над основанием ноги на переднем конце тела расположено ротовое отверстие, по бокам которого имеются две лопасти с мерцательным эпителием, реснички которого гонят пищевые частицы ко рту. На жабрах и ротовых лопастях имеются органы вкуса и ресничные желобки, по которым частички пищи транспортируются в рот. Через короткий пищевод пища попадает в небольшой желудок, куда открываются протоки печени. Глотка, терка и слюнные железы из-за редукции головы у двустворчатых моллюсков отсутствуют.

Из желудка пища поступает в кишку, которая образует несколько петель и затем через околосердечную сумку и желудочек сердца проходит назад и открывается анальным отверстием в выводной (клоакальный) сифон, откуда с током воды экскременты выбрасываются наружу.

Кровеносная система представлена сердцем с одним желудочком и двумя предсердиями. Сердце лежит на спинной стороне в околосердечной сумке. От желудочка берут начало два мощных сосуда: передняя и задняя аорты. Из лакун полости тела венозная кровь направляется в приносящие жаберные сосуды. Окисленная артериальная кровь из жабер по выносящим сосудам возвращается в сердце. Часть крови проходит в почки, миная жабры, где освобождается от конечных продуктов обмена и вливается в выносящие жаберные сосуды, которые впадают в предсердие.

Органы выделения представлены двумя почками, которые лежат под сердцем. Каждая почка начинается выстланной мерцательным эпителием воронкой в околосердечной сумке. Мочеточники открываются в мантийную полость.

Органы размножения. Большинство двустворчатых моллюсков раздельнополы. Половые железы парные, а их протоки открываются в мантийную полость. Оплодотворение яйцеклеток наружное. Мужские половые клетки из мантийной полости самцов наружу попадают через выводной сифон. У пресноводных форм оплодотворение происходит в мантийной полости самки, куда через жаберный сифон с током воды заносятся спермии. Такое оплодотворение возможно при тесном заселении моллюсков.

Развитие у большого числа видов происходит с метаморфозом. У морских представителей личинка парусник похожа на личинку колышатых червей: прозрачное тело округлой формы с диском, покрытым поясами ресничек, — парус.

У пресноводных моллюсков (беззубки, перловицы) личинки глохидии имеют двустворчатую раковину с зубцами по краям (см. рис. 81). Яйца откладываются в жабры, где и развиваются глохидии. Глохидии появляются осенью и всю зиму остаются в мантийной полости материнского организма. Покидают глохидии мантийную полость весной с током воды через клоакальный сифон и закрепляются на жабрах рыб. Ткани рыб обрастают глохидиев, которые превращаются в эктопаразитов, питаясь соками рыбы-хозяина. Через 2—3 мес молодые моллюски покидают хозяина, опускаются на дно и начинают самостоятельную жизнь. Эктопаразитизм глохидиев на рыбах обеспечивает расселение моллюсков в водоемах.

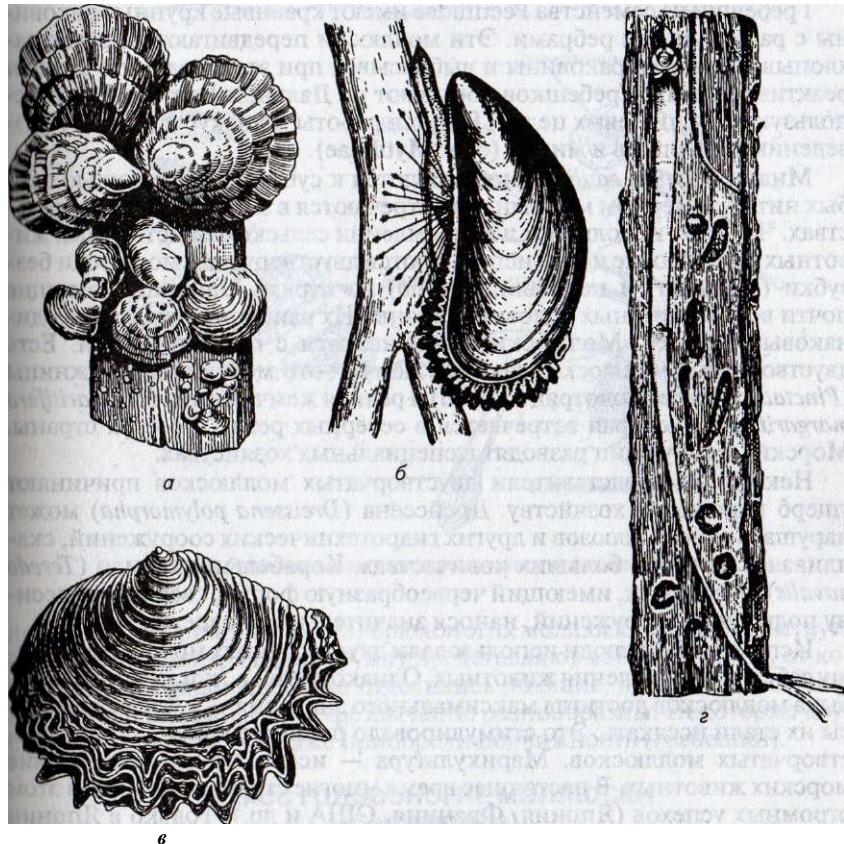


Рис. 87. Представители двустворчатых моллюсков:
а — устрицы (*Ostrea*) на куске дерева; б — мидии (*Mytilus edulis*), прикрепленная к субстрату (юмошью нитей биссуса); в — жемчужница (*Pteriina*); г — корабельный червь (*Teredo navalis*)

Пластинчатожаберные моллюски являются очень эффективными естественными биофильтраторами воды. Например, при высокой плотности заселения мидии, живущие на 1 м² поверхности дна, пропускают через себя более 250 т воды.

Многих двустворчатых моллюсков используют в пищу (устрицы, гребешки, мидии и др.; рис. 87). Устрицы (сем. Ostreidae) ведут неподвижный образ жизни в морских водах, имеют асимметричную раковину. Большой выпуклой створкой они соприкасаются с субстратом, образуя огромные скопления на мелководьях. Во многих странах мясо устриц, характеризующееся высокой питательной ценностью, широко используют в пищу. В прибрежных странах эффективно развивается промышленное разведение устриц.

Гребешки из семейства Pectinidae имеют красивые крупные раковины с радиальными ребрами. Эти моллюски передвигаются, резко захлопывая створки раковины и выбрасывая при этом воду, что создает реактивную тягу. Гребешков добывают на Дальнем Востоке РФ и используют для пищевых целей. Ведутся работы по искусственноому разведению гребешков и мидий (сем. Mytilidae).

Мидии (*Mytilus edulis*) прикрепляются к субстрату с помощью особых нитей. В Черном море мидии встречаются в значительных количествах. Часто их используют для кормления сельскохозяйственных животных. Для этих же целей используются двустворчатые моллюски беззубки (*Anodonta*) и перловицы (*Unio*) из отряда Unionida, живущие почти во всех пресных водоемах страны. Их раковины имеют две одинаковые створки. Моллюски передвигаются с помощью ноги. Есть двустворчатые моллюски, дающие жемчуг: это морские жемчужницы (*Pinctada* и *Pteria*, подотряд *Pteriina*) и речная жемчужница (*Margaritifera margaritifera*), которая встречается в северных реках и озерах страны. Морских жемчужниц разводят в специальных хозяйствах.

Некоторые представители двустворчатых моллюсков причиняют ущерб народному хозяйству. Дрейссена (*Dreissena polymorpha*) может нарушать работу шлюзов и других гидротехнических сооружений, скапливаясь на них в больших количествах. Корабельное точило (*Teredo navalis*) — моллюск, имеющий червеобразную форму, сверлит древесину подводных сооружений, нанося значительные повреждения.

Испокон веков люди использовали двустворчатых моллюсков в пищу себе и для кормления животных. Однако с 1962 г., когда мировая добыча моллюсков достигла максимального значения — 1,7 млн т, ресурсы их стали иссякать. Это стимулировало развитие марикультуры двустворчатых моллюсков. Марикультура — искусственное разведение морских животных. В настоящее время многие страны добились в этом огромных успехов (Япония, Франция, США и др.). Только в Японии путем выращивания жемчужниц ежегодно получают более 100 тыс. жемчужин. Ведутся такие работы и в нашей стране.

Двустворчатые моллюски выполняют и очистительные функции в водоемах. Они поглощают и накашивают в своем теле тяжелые металлы и очищают воду от химических загрязнений. В среднем один моллюск пропускает за 1 ч около 1 л воды. Перловицы и беззубки являются действующими биофилtrаторами. В этом отношении они представляют большую ценность, чем как источник мяса.

Филогения двустворчатых моллюсков. Наиболее примитивны по своей организации первичножаберные двустворчатые моллюски, которые и берут начало от древних первичножаберных предков. В дальнейшем в процессе эволюции при переходе к неподвижному образу жизни у моллюсков редуцировалась нога; у перегородчатожаберных редуцировались жабры, а функции дыхания перешли к наджаберным полостям.

У предков двустворчатых раковина, видимо, была цельной. Жизнь на субстрате заставила моллюсков защитить свое тело по бокам. Эко-

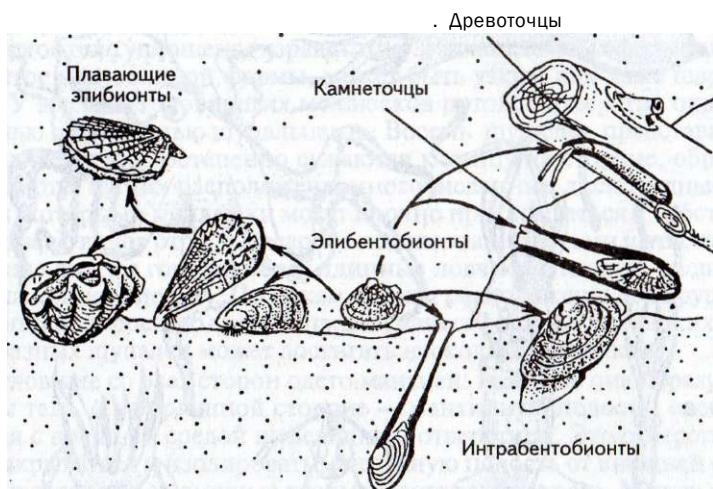


Рис. 88. Экологическая радиация двустворчатых моллюсков

логическая радиация (рис. 88) брюхоногих моллюсков свидетельствует о том, что их центральную группу представляют донные формы, от которых путем специализации отделились роющие, древо- и камнеточцы. Неподвижные формы чрезвычайно разнообразны. Некоторые двустворчатые моллюски даже приобрели подвижность (гребешки).

КЛАСС ГОЛОВОНОГИЕ МОЛЛЮСКИ (Cephalopoda)

Общая характеристика. К этому классу принадлежит около 700 видов крупных моллюсков, живущих исключительно в морях и отличающихся наиболее сложной организацией. Из-за совершенных приспособлений к жизни в морской стихии и сложности поведения головоногих моллюсков часто называют «приматами моря» среди беспозвоночных животных. Обычно это свободноплавающие и подвижные хищники, предпочитающие воды теплых морей и океанов. Среди них мало ползающих видов. Их размеры колеблются от нескольких сантиметров до 18 м (гигантские кальмары).

Тело отчетливо подразделяется на голову и туловище. Нога же превращена в щупальца (руки), которые вторично сместились на голову и окружают ротовое отверстие (отсюда и их название — головоногие). Другая часть ноги преобразовалась в воронку, лежащую у входа в мантийную полость на брюшной стороне тела.

У примитивных форм раковина наружная, у высших представителей она внутренняя, может быть частично или полностью редуцирована.

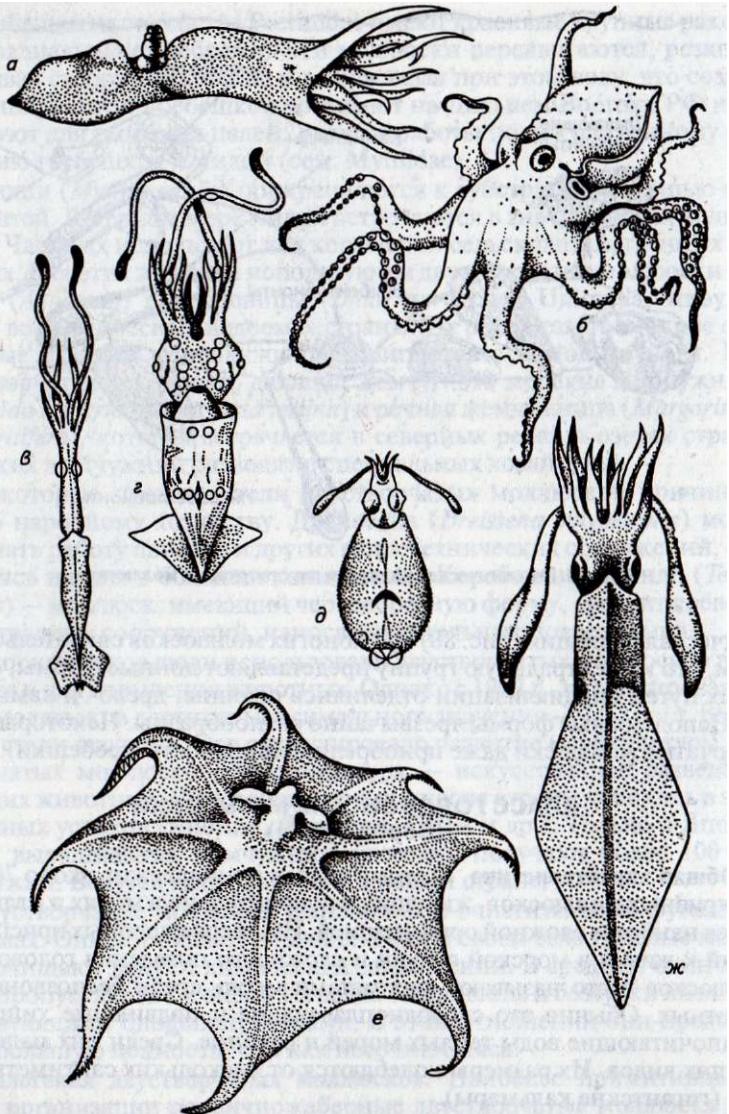


Рис. 89. Различные головоногие:
 а — глубоководный плавающий осьминог (*Amphitretus pelagicus*); б — осьминог (*Benthoctopus profundorum*); в — планктонный кальмар (*Doratopsis sagitta*); г — глубоководный пелагический кальмар со светящимися органами (*Lycoteuthis diadema*); д — планктонный кальмарчик (*Cranchia scabra*); е — донный осьминог (*Cirrothauma murrayi*); ж — пелагический кальмар (*Loligo edulis*)

Строение и жизненные отправления. Моллюски, обитающие в толще воды, имеют тело торпедовидной формы (кальмары), у бентосных форм тело мешкообразной формы (осьминоги), у других универсальных видов тело уплощено (каракатицы). У планктонных форм тело студенистое медузиодной формы, может быть узким или даже шаровидным. У высших головоногих моллюсков ротовое отверстие окружено восемью или десятью щупальцами. Восемь щупалец представителей отряда Octopoda постепенно сужаются к концу на стороне, обращенной ко рту, на них расположены многочисленные дисковидные присоски, которыми моллюски могут прочно присасываться к субстрату и к жертве. У видов отряда Decapoda помимо таких восьми щупалец имеется еще два, но гораздо более длинные ловчие щупальца, расширенные на конце (рис. 89). По бокам головы расположены два крупных и сложных по строению глаза. У примитивных форм число гладких и червеобразных щупалец может достигать нескольких десятков.

Туловище со всех сторон одето мантией: на спине она образует покровы тела, а на брюшной стороне — мантийную полость, сообщающуюся с внешней средой щелевидным отверстием. Это отверстие может закрываться и изолировать мантийную полость от внешней среды. Закрывается оно с помощью особых «застежек-кнопок». Между «кнопками» на брюшной стороне из этой щели выступает воронка в виде мускулистой трубы. Расширенный конец воронки открывается в мантийную полость, а узкий — наружу. Воронка (производное ноги) служит для особого реактивного движения. Когда мантийная щель закрыта замыкателями с помощью многочисленных мышц, мантия прижимается к туловищу. Вода из мантийной полости с силой выталкивается через воронку, толкая моллюска в обратную сторону (реактивная тяга). Воронка может изгибаться в разные стороны, что позволяет моллюску менять направление движения. Роль дополнительного руля выполняют щупальца и плавники в виде складки кожи. Ритмические сокращения мантии и выталкивания воды позволяют моллюску не только плавать, но и интенсивно омывать жабры водой.

В мантийную полость на брюшной стороне головоногих моллюсков открываются половые и мочевыводящие протоки, а также анальное отверстие (рис. 90).

У современных головоногих раковина сильно редуцирована и обрастает боковыми складками мантии, становясь внутренней. У некоторых представителей (каракатица *Sepia*) раковина в виде известковой пластинки залегает под покровами на спинной стороне туловища. У кальмара (*Loligo*) от раковины остается лишь скрытый под покровами спинной роговой листок. У некоторых видов раковина остается лишь у самок или исчезает вовсе.

Покровы представлены однослойным эпителием и слоем соединительной ткани под ним. Головоногие моллюски способны к быстрой и резкой смене своей окраски, что обуславливается наличием в соединительнотканном слое кожи многочисленных пигментных клеток — хроматофоров. Механизм смены окраски контролируется нервной сис-

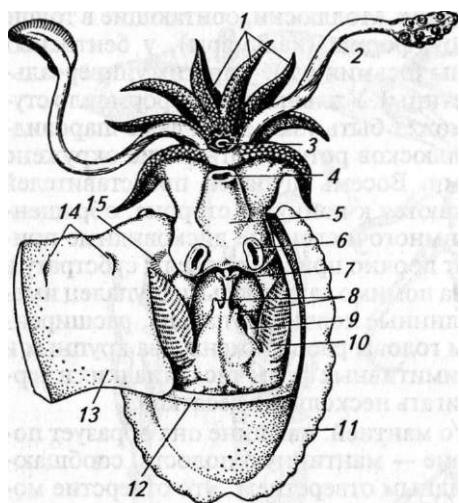


Рис. 90. Каракатица *Sepia officinalis* со вскрытой мантийной полостью; вид с брюшной стороны:

1 — руки с присосками; 2 — ловчая рука; 3 — рот; 4 — отверстие воронки; 5 — воронка; 6 — хрящевые ямки кнопок; 7 — анальное отверстие; 8 — почечные сосочки; 9 — непарный половой сосочек; 10 — жабры; 11 — плавник; 12 — линия отреза мантии; 13 — отогнутая мантия; 14 — хрящевые бугорки кнопок; 15 — мантийный звездчатый ганглий

темой, получающей информацию по зрительным нервам.

Нервная система головоногих моллюсков устроена наименее сложно. Нервные ганглии образуют крупное окологлоточное скопление — мозг,

заключенный в хрящевую капсулу (соответствует по выполняемой функции черепу позвоночных). От заднего отдела ганглиозной массы отходят два крупных мантийных нерва.

Органы чувств хорошо развиты: обонятельные ямки под глазами, обладающие высокой чувствительностью, пара статоцистов внутри хрящевой головной капсулы, крупные и сложно устроенные глаза, способные к аккомодации. Глаза по своему строению напоминают глаза млекопитающих (пример конвергенции между беспозвоночными и позвоночными животными). Глазное яблоко сверху покрыто роговицей, имеющей отверстие в переднюю камеру глаза. Радужная оболочка образует отверстие — зрачок, через который свет попадает на хрусталик. Аккомодация глаза происходит за счет удаления хрусталика от сетчатки или приближения его (у млекопитающих аккомодация осуществляется путем изменения кривизны хрусталика). Глаза окружены хрящевой капсулой. На коже имеются особые органы свечения, по строению напоминающие глаза.

Органы пищеварения также сложно устроены и несут черты специализации к питанию животной пищей. Ротовое отверстие, лежащее в центре венца щупалец, ведет в мускулистую глотку, в которой находится язык с теркой. В глотке расположены две толстые роговые челюсти, загнутые в виде крючка и напоминающие клюв попугая. В глотку открываются протоки одной-двух пар слюнных желез, секрет которых обладает амилолитической и протеолитической активностью, может содержать яды. Головоногие моллюски питаются только полужидкой пищей, поскольку у них узкий пищевод, который проходит через мозг моллюска. Пища сначала разгрызается роговыми челюстями, а затем обильно смачивается слюной и перетирается теркой. Длинный пищевод может иметь расширение — зоб.

Из пищевода пища попадает в мускулистый энтодермальный желудок, имеющий слепой мешковидный отросток. От желудка отходит тонкая кишка, переходящая в заднюю кишку, оканчивающуюся анальным отверстием в мантийную полость. В желудок впадают протоки печени, секрет которой имеет весь набор пищеварительных ферментов. Гистия и поджелудочная железа в виде небольших придатков в протоках печени.

В заднюю кишку перед анальным отверстием открывается проток чернильного мешка, в котором образуется черная жидкость. Выбрасывая эту чернильную жидкость через анальное отверстие, а затем и из мантийной полости через воронку наружу, моллюски окружают себя темным облаком, что позволяет им скрыться от врагов. Питаются головоногие моллюски в основном рыбой, крабами и двустворчатыми моллюсками, схватывая их щупальцами и убивая челюстями и ядом.

Органы дыхания — жабры, расположенные в мантийной полости симметрично по бокам туловища. Обмен воды осуществляется сокращением мантийных мышц и работой воронки, через которую вода выталкивается наружу. По числу жабр головоногие моллюски делятся на две группы: четырехжаберные (*Tetrabranchia*) идвужаберные (*Dibranchia*).

Кровеносная система представлена сердцем с одним желудочком и двумя или четырьмя предсердиями (по числу жабр). Кровь движется за счет сокращений сердца, а также за счет пульсации участков сосудов. От переднего и заднего концов желудочка сердца отходят головная и мнутренностная аорты. Капилляры вен и артерий в коже и мышцах переходят друг в друга и лишь в некоторых местах сохраняются лакунарные пространства; таким образом, кровеносная система почти замкнутая. Кровь на воздухе голубеет, поскольку содержит гемоцианин (богатое мембрено соединение, соответствующее по физиологическим функциям гемоглобину позвоночных).

Выделительная система состоит из двух или четырех почек, берущих начало отверстиями в целоме (околосердечной сумке). Конечные продукты обмена поступают из жаберных вен и околосердечной сумки и выделяются в мантийную полость рядом с анальным отверстием.

Половая система. Головоногие моллюски — раздельнополые животные, у которых часто хорошо выражен половой диморфизм. Половые железы и их протоки непарные. Половые продукты накапливаются в целоме и выводятся через половые протоки. Спермин склеиваются в тинерматофоры — пакеты с плотной оболочкой.

Оплодотворение обычно происходит в мантийной полости самки, роль копулятивного органа играет одно из щупалец, которое у самцов отличается наличием особого ложкообразного придатка. С помощью этого щупальца самец вводит сперматофоры в мантийную полость самки. Все развитие зародышей протекает внутри яиц, которые самка откладывает на дне. У некоторых головоногих проявляется забота о потомстве: самка аргонавта вынашивает яйца в выводковой камере, осьминоги охраняют кладку яиц.

Современные головоногие относятся к двум подклассам: подкласс Наутилиды (*Nautiloidea*) и подкласс Колеоиды (*Coleoidea*).

Головоногие моллюски отличаются крупными размерами: от нескольких сантиметров до нескольких метров. Удалось обнаружить 10-метровое щупальце головоногого моллюска. Живут моллюски только в морях и ведут разнообразный образ жизни. Большинство относятся к пелагическим животным, живущим в толще воды. У донных видов (часть осьминогов) между щупальцами есть перепонка, придающая телу моллюска вид диска, лежащего на дне. Все головоногие — хищники, нападающие на ракообразных и рыб, которых они схватывают щупальцами, убиваючи членами и ядом слюнных желез.

Многие головоногие являются объектом промысла: кальмаров, каракатиц и осьминогов человек использует в пищу, поскольку их мясо обладает высокой пищевой ценностью. Мировой улов головоногих достигает более 1,6 млрд т в год.

Наутилиды включают лишь один отряд *Nautilida*, к которому относится всего несколько видов, обитающих в тропических областях океанов. Наутилиды характеризуются многими примитивными чертами: наружная многокамерная раковина, многочисленные без присосок щупальца, проявление метамерии и т. п. Наутилус плавает реактивным способом. Является объектом промысла из-за красивой раковины.

Подкласс Колеоиды (*Coleoidea*) включает около 650 видов жестко-кожих моллюсков, лишенных раковины. У них сросшаяся воронка и щупальца вооружены присосками, кроме того, у них две жабры, две почки и два предсердия.

Характерным представителем отряда являются каракатицы (*Sepia*), имеющие десять щупалец, из которых два ловчие. Обитают вблизи дна и ведут активный плавающий образ жизни.

К отряду Кальмары (*Teuthida*) относятся многие промысловые виды (*Todarodes*, *Loligo* и др.) У них иногда сохраняетсяrudиментарная раковина в виде роговой пластинки под кожей. Кальмары имеют десять щупалец. Это торпедовидные обитатели толщи океанских вод.

Следов раковины нет у наиболее эволюционно прогрессивных головоногих моллюсков — представителей отряда Восьминогие (*Octopoda*). У них восемь щупалец, одно из которых у самцов превращено в половое. Большинство осьминогов обитают в придонном слое воды. Среди осьминогов есть представители, имеющие выводковую камеру (аргонавт).

Филогения головоногих моллюсков. Самые древние представители головоногих — наутилиды, раковины которых обнаруживаются в ископаемых кембрийских отложениях. Считается, что головоногие произошли от древних ползающих раковинных моллюсков. В процессе эволюции сформировалась группа головоногих моллюсков, лишенных раковины, с новым реактивным типом движения, со сложной нервной системой и сложными органами чувств.

От примитивных раковинных бенто-пелагических форм определилось несколько путей экологической специализации (рис. 91). Проис-

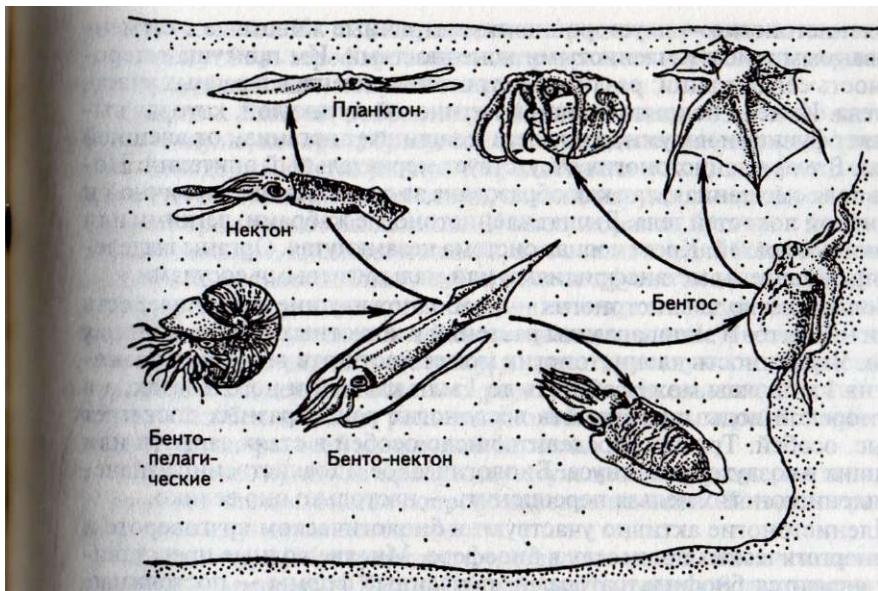


Рис. 91. Экологическая радиация головоногих моллюсков

ходит переход к бенто-нектонным формам, у которых раковина становится внутренней и ее функция как плавательного аппарата ослабевает, но развивается новая модель движителя — воронка. Они-то и дали начало безраковинным моллюскам, которые образуют бенто-нектонные (каракатицы, осьминоги), нектонные (кальмары, осьминоги и каракатицы), бентосные и планктонные (зонтикообразные осьминоги, палочковидные кальмары) формы.

ТИП ЧЛЕНИСТОНОГИЕ (Arthropoda)

Общая характеристика. Это наиболее распространенная и процветающая группа животных, обладающих членистыми конечностями и сегментированным телом. Насчитывают более 1,5 млн видов членистоногих, населяющих моря, океаны, пресные водоемы, поверхность суши, почву и воздушную среду. Среди них много паразитов. Членистоногие освоили все способы движения, среди них имеются плавающие, роющие, ползающие, бегающие, летающие, прыгающие, реже малоподвижные и неподвижные формы. Чрезвычайно разнообразен спектр их питания: от обычной животной или растительной пищи до самых труднопереваримых объектов — древесины, рога, волоса, пера и т. п.

Членистоногие — двусторонне-симметричные животные с сегментированным телом и членистыми конечностями. Им присуща гетерономность сегментации: различное строение сегментов в разных участках тела. Их тело покрыто прочной хитиновой кутикулой, которая выполняет функции наружного скелета и защищает организм от внешней среды. В теле членистоногих отсутствует мерцательный эпителий. Половость тела смешанная, так как образована за счет слияния первичной и вторичной полостей тела. Дышат членистоногие жабрами, легкими или с помощью трахей. Кровеносная система незамкнутая. Органы выделения представлены метанефридиями или мальпигиевыми сосудами.

Большинство членистоногих — свободноживущие животные, есть среди них экто- и эндопаразиты растений и животных. Много хищных форм. Численность членистоногих может достигать огромных размеров: на 1 м² почвы может обитать до 1 млн клещей и ногохвосток, а в 1 м³ морской воды численность веслоногих ракообразных достигает 30 тыс. особей. Трудно определить число особей в стаях саранчи или парящих в воздухе тучах гусениц. Биологическое и хозяйственное значение членистоногих нельзя переоценить — настолько оно велико.

Членистоногие активно участвуют в биологическом круговороте и биоэнергетических процессах в биосфере. Многие водные представители являются биофильтраторами, почвенные формы — постоянные участники почвообразовательных процессов. Среди членистоногих много объектов промысла и промышленного разведения, постоянно вовлекаются в сельскохозяйственное производство новые виды насекомых (пчелы, шмели, наездники, осы, хищные формы, паразитические виды и др.). Среди членистоногих есть опасные вредители лесов и сельскохозяйственных культур, паразиты и переносчики заболеваний животных и человека.

Тип Членистоногие подразделяют на четыре подтипа, из которых наибольший интерес представляют три подтипа: Жабродышащие (Branchiata), Хелициеровые (Chelicera) и Трахейные (Tracheata).

Строение и жизненные отправления. Форма тела членистоногих чрезвычайно разнообразна. Тело состоит из сегментов, следующих друг за другом. Каждый сегмент покрыт четырьмя склеритами (твердыми хитиновыми пластинками): спинная пластинка — тергит, брюшная пластинка — стернит и две боковые пластинки. Между склеритами расположены мягкие сочлененные мембранны, придающие подвижность каждому сегменту. У некоторых представителей сегменты сходны, но у большинства видов они различны, что позволяет выделить отделы тела: голову, грудь и брюшко. Головной отдел состоит из акрона и четырех сегментов. У некоторых представителей сегменты тела сливаются в нерасчлененные головогрудь и брюшко, а у клещей сегментация тела не выражена, и они имеют нерасчлененное тело.

Как правило, каждый сегмент тела членистоногих несет по паре членистых конечностей, что и определило название типа. Конечности подвижно соединяются с телом с помощью суставов и состоят из нескольких члеников, образуя многоколенный рычаг, способный к силь-

ным движениям. На отдельных сегментах или даже отделах тела конечности могут быть атрофированы или превращены в различные органы — **ротовые**, яйцеклады, копулятивные и др.

Покровы. Главная особенность членистоногих — наличие хитиновой кутикулы, которая образуется за счет выделения наружного покрова — гиподермы. Кутикула прочна, эластична и защищает тело членистоногих от внешних воздействий, служит наружным скелетом, предохраняет организм от высыхания. У высших раков кутикула пропитана солями кальция, кремния и железа, в ее состав входят белковые, жиро-подобные, воскоподобные, дубильные и другие вещества. Особенно характерен для кутикулы полисахарид хитин, в молекуле которого в отличие от других углеводов содержатся атомы азота. По своему строению хитин напоминает клетчатку, инкрустирующую клетки растений, что может свидетельствовать о родстве растительного и животного царств. Особенno важна роль хитиновой кутикулы в работе конечностей и как опоры всей мышечной системы.

Хитиновая кутикула препятствует увеличению размеров тела, поэтому рост членистоногих сопровождается линькой. Животное растет, пока новая кутикула не затвердеет.

Мускулатура. Мускулатура представлена отдельными мышечными пучками — мышцами. Кожно-мускульного мешка у большинства членистоногих нет. Мыши прикреплены к наружному покрову. Мускулатура внутренних органов представлена гладкими мышечными волокнами, а остальная мускулатура имеет поперечно-полосатую структуру, что отличает членистоногих от червей, имеющих гладкую мускулатуру. Строение мускулатуры членистоногих является примером конвергенции с позвоночными животными. Но относительная сила мышц членистоногих значительно выше, чем у млекопитающих. Мышцам членистоногих свойственна весьма высокая частота сокращений (крыловые мышцы насекомых).

Полость тела — смешанная (миксоцель). Появление членистых конечностей и поперечно-полосатой мускулатуры позволило многим членистоногим вести активный образ жизни. Каждый членик ноги, **любой** сегмент приводится в движение определенными мышцами, работу которых контролирует нервная система.

Нервная система во многом идентична нервной системе кольчатых Червей. Над глоткой в головном отделе расположены парные надглоточные ганглии, образующие хорошо развитый мозг. Имеются около-Глоточное нервное кольцо и брюшная нервная цепочка. При слиянии ряда сегментов тела соответственно происходит слияние и нервных узлов брюшной цепочки. Узлы брюшной нервной цепочки членистоногих выполняют разные функции, так как у этих животных происходит Дифференцировка тела на разные отделы. Этим членистоногие отличаются от кольчатых червей, у которых функции всех узлов нервной цепочки одинаковы.

Для членистоногих характерно сложные поведение и ориентация В пространстве, что привело к развитию высокоспециализированных

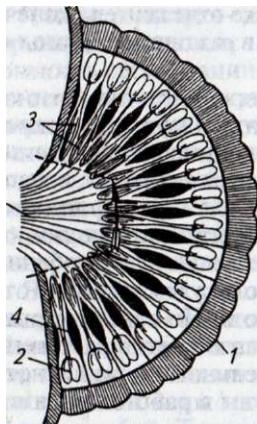


Рис. 92. Схема строения сложного глаза членистоногого:
1 — роговица; 2 — хрустальный конус; 3 — светочувствительные элементы глаза;
4 — пигментированные прослойки между омматидиями; 5 — зрительный нерв

ется на лежащие ниже светочувствительные клетки. Омматидии отделены друг от друга пигментными клетками, которые позволяют различать не только форму, но и цвета предметов. Каждый омматидий воспринимает лишь часть предмета, так как их нервные клетки воспринимают только те лучи, которые падают перпендикулярно к поверхности омматидия. Поэтому изображение предмета в сложном глазу составляется из отдельных частей (как мозаика). Отсюда и название такого зрения — мозаичное. У некоторых раков сложные глаза расположены на особых стебельках.

Хорошо развиты у членистоногих органы равновесия, слуха, осязания и обоняния. У высших членистоногих обоняние достигает совершенства.

Органы пищеварения. Ротовое отверстие находится на головном отделе и вооружено видоизмененными конечностями — ротовыми частями, позволяющими добывать, удерживать, размельчать, переваривать и заглатывать пищу. Строение и форма ротовых частей чрезвычайно разнообразны. Пищеварительная система имеет три отдела. В переднем отделе обособлены глотка и пищевод, а также жевательный желудок, покрытые кутикулой. В среднем отделе происходят процессы переваривания и всасывания пищи. Роль пищеварительных желез выполняют печень или особые пилорические придатки. Задний отдел достигает значительной длины и заканчивается анальным отверстием. Передний и задний отделы имеют эктодермальное происхождение и выстланы изнутри хитиновой кутикулой, предохраняющей внутреннюю поверхность кишечника от травмирования жесткими остатками пищи.

органов чувств. У многих членистоногих наблюдается сложное инстинктивное поведение, а высшие членистоногие способны к быстрому образованию новых условных рефлексов. Особенно сложные формы поведения и взаимоотношения присущи общественным насекомым.

У большинства представителей членистоногих имеются простые или сложные глаза или одновременно и те, и другие. *Простые глаза* имеют форму бокала, устье которого закрыто хрусталиком. Дно бокала выстлано светочувствительными клетками, от которых отходит зрительный нерв. Свет через хрусталик направляется на светочувствительный слой клеток.

Сложные глаза состоят из множества глазков — омматидиев, имеющих коническую или цилиндрическую форму (рис. 92). Наружная часть омматидиев представлена прозрачной роговицей (прозрачная часть хитиновой кутикулы). Под роговицей расположен конусовидный прозрачный хрусталик, через который луч света направляется на светочувствительные клетки. Омматидии отделены друг от друга пигментными клетками, которые позволяют различать не только форму, но и цвета предметов. Каждый омматидий воспринимает лишь часть предмета, так как его нервные клетки воспринимают только те лучи, которые падают перпендикулярно к поверхности омматидия. Поэтому изображение предмета в сложном глазу составляется из отдельных частей (как мозаика). Отсюда и название такого зрения — мозаичное. У некоторых раков сложные глаза расположены на особых стебельках.

Органы дыхания. У основной массы водных обитателей — жабры, роль наружных покровов в газообмене, как правило, незначительна. Снижение роли наружных покровов в дыхании объясняется наличием плотной хитиновой кутикулы, через которую проникновение кислорода ограничено или просто невозможно, если кутикула уплотнена и пропитана минеральными солями. Даже при достаточной проницаемости кутикулы потребность в кислороде у ведущих активный образ жизни членистоногих в большинстве случаев не может быть удовлетворена только за счет дыхания через покровы тела. Это возможно лишь у небольших по размерам членистоногих, у которых очень тонкая кутикула и относительно большая площадь поверхности тела.

У сухопутных и некоторых водных членистоногих органами дыхания служат легкие или трахеи. Легкие представляют собой тонкостенные мешки, внутри которых имеются многочисленные тонкие листочки. Через покровы этих листочек и происходит газообмен. Трахеи имеют вид ветвящихся трубочек, которые открываются во внешнюю среду специальными отверстиями — дыхальцами (рис. 93). Спадаться при дыхании им не позволяют упругие спиральные хитиновые нити. Конечные разветвления трахей лишены этих нитей и проникают во все ткани и органы членистоногих, снабжая их кислородом воздуха. Только у части членистоногих дыхание осуществляется через покровы тела.

Кровеносная система у членистоногих незамкнутая, что обусловлено наличием смешанной полости тела. Активизация жизнедеятельности требует ускорения транспорта питательных веществ, что способствует

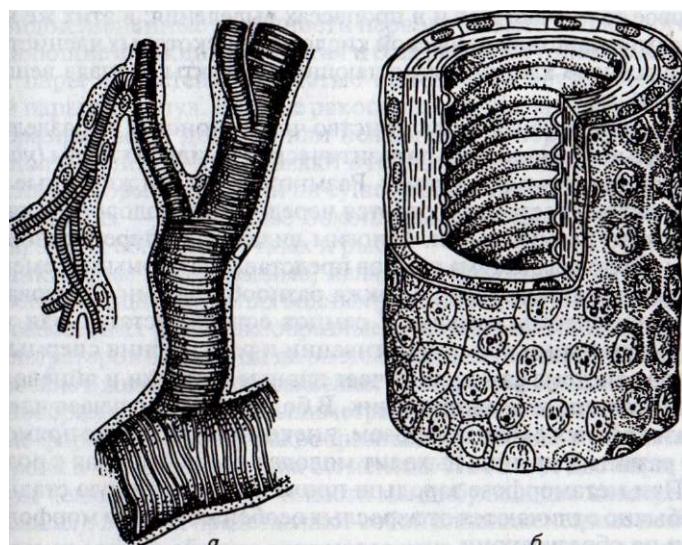


Рис. 93. Трахеи насекомых:
а — участок трахеи; б — микроскопическое строение трахеи

появлению специального органа — сердца, развивающегося из части спинного сосуда. Сердце расположено на спинной стороне животного. Сердце сокращается очень часто — до ста сокращений в минуту, что обеспечивает быстрый оборот крови (гемолимфы). Из сердца гемолимфа выталкивается через артерии в синусы миксоцеля, где омывает все внутренние органы. Обратно к сердцу гемолимфа проходит по лакунам, откуда засасывается через специальные отверстия — остии — с закрывающимися клапанами, которые имеются в стенках сердца. Гемолимфа — жидкость, частично соответствующая крови и частично целомической жидкости, выполняющая функции крови.

У некоторых членистоногих кровеносная система отсутствует или представлена одним сердцем. У трахейнодышащих функции крови ограничиваются доставкой всосавшихся в кишечнике питательных веществ. Цвет крови чаще всего желтоватый, но может быть красным, голубым и др. В крови содержатся различные клетки.

Органы выделения. У ракообразных выделительные функции выполняют видоизмененные метанефриды — почки, расположенные в головном отделе. У насекомых и многоножек органами выделения являются мальпигиевые сосуды — тонкие слепые отростки кишечника. Через стенки мальпигиевых сосудов происходит удаление конечных продуктов обмена и излишков воды. У паукообразных имеются и метанефриды, и мальпигиевые сосуды.

У многих членистоногих существенную роль в обмене веществ играет жировое тело. Оно представляет собой скопление крупных клеток, в пазме которых образуются капли жира как энергетического резерва. Но жировое тело участвует и в процессах выделения: в этих же клетках скапливаются кристаллы мочевой кислоты. У некоторых членистоногих есть специальные клетки, поглощающие продукты распада веществ — нефроциты.

Органы размножения. Большинство членистоногих — раздельнополые животные, и лишь часть паразитических и сидячих форм (усоногие раки) являются гермафродитами. Размножаются эти животные только половым путем. Часто наблюдается чередование полового и партеногенетического размножения. Многим видам характерен половой диморфизм. Половая система самцов представлена парными семенниками и двумя семяпроводами, а также разнообразными образованиями для передачи спермы самкам. У самцов есть предстательная железа, секретирующая жидкость для активации и разжижения спермы.

Половая система самок включает парные яичники и яйцеводы, непарное влагалище и семяприемник. В большинстве случаев членистоногие развиваются с метаморфозом, у некоторых развитие прямое. При прямом развитии из яйца выходит молодая особь, сходная с родительскими. При метаморфозе зародыш проходит личиночную стадию; личинки обычно отличаются от взрослых особей не только морфологически, но и по образу жизни.

Рост и развитие членистоногих связаны с периодическими линьками. Хитиновая кутикула после ее образования быстро затвердевает.

Поэтому увеличение размеров тела может происходить только после сбрасывания старой кутикулы и до затвердевания новой. Иными словами, рост членистоногих происходит периодически и тесно связан с периодичностью линьки. Перед началом линьки между кутикулой и телом членистоногих образуется жидкость, которая растворяет часть старой кутикулы (эндокутикула). Часть нерастворившейся кутикулы (эктокутикула) лопается и сбрасывается животным. Членистоногие в период линьки становятся беззащитными и поэтому почти не питаются и прячутся в укромных местах.

ПОДТИП ЖАБРОДЫШАЩИЕ (*Branchiata*)

Из обширного подтипа жабродышащих водных членистоногих в настоящее время сохранился только один класс — Ракообразные, насчитывающий более 40 тыс. видов.

КЛАСС РАКООБРАЗНЫЕ (*Crustacea*)

Общая характеристика. Основная масса ракообразных — обитатели соленых и пресных водоемов, и лишь немногие живут во влажных местах на суше (мокрицы). Тело ракообразных делится на голову, грудь и брюшко. Часто голова и грудь, сливаясь, образуют головогрудь. На голове имеются две пары усиков: антеннулы — придатки акрона, и антены — видоизмененные конечности первого сегмента головного отдела, выполняющие функции обоняния и осознания. На голове располагаются три пары челюстей. Членистые конечности двуветвистые, кроме первой пары антеннул. Водные ракообразные дышат жабрами. Многие ракообразные ведут донный или пелagicкий образ жизни. Они активно ползают по дну или плавают в толще воды, но встречаются и прикрепленные формы. К жизни на суше хорошо приспособились мокрицы, в тропиках — почвенные бокоплавы и наземные формы крабов. **Есть паразиты беспозвоночных и рыб.**

Планктонные ракообразные, являясь фитофагами, служат важнейшим звеном в пищевых цепях водных экосистем, составляя основу пищи для промысловых рыб. Ракообразные — самая многочисленная группа биофильтраторов воды. Они являются важным объектом промысла.

Строение и жизненные отравления. Размеры и форма тела чрезвычайно разнообразны: от долей миллиметра (обитатели толщи воды) до метра (донные формы). Голова у ракообразных образована в результате слияния акрона и четырех передних сегментов. На голове две пары усиков и три пары челюстей (верхние челюсти мандибулы и две нижние челюсти — максиллы), все они представляют собой видоизмененные конечности.

Сегменты груди обычно сливаются друг с другом или с головой, образуя головогрудь. У высших раков голову и грудь сверху и по бокам закрывает хитиновый щит — карапакс, защищающий жабры (рис. 94, 95).

Рис. 96. Конечности самца речного рака:

1 — первая пара усиков; 2 — вторая пара усиков;
3 — жвалы (верхние челюсти); 4,5 — первая и вторая па-
ры нижних челюстей; 6—8 — ногочелюсти; 9—13 — хо-
дильные ноги; 14—19 — брюшные конечности

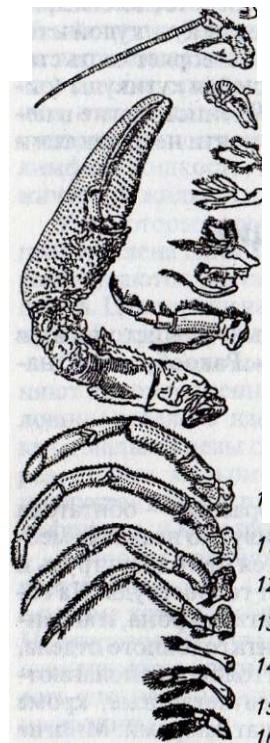
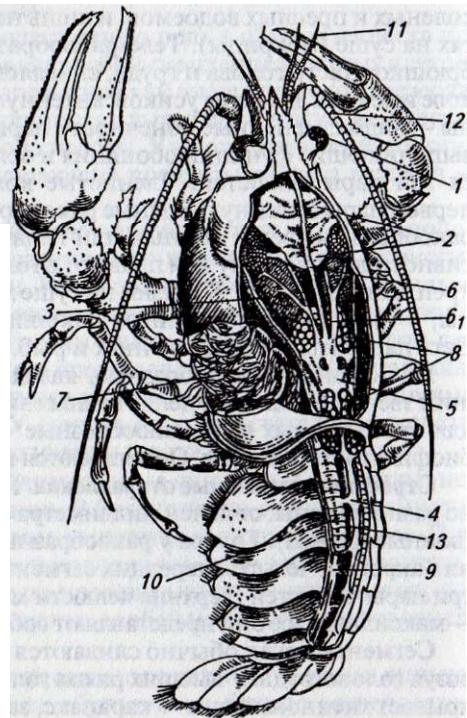


Рис. 97. Вскрытая самка речного рака:

1 — сложный глаз; 2 — желудок; 3 — печень;
4,6,6₁ — кровеносные сосуды; 5 — сердце;
7 — жабры; 8 — яичник; 9 — брюшная нервная цепочка;
10 — мышцы брюшка; 11 — первая пара
усиков; 12 — вторая пара усиков; 13 —
задняя кишка



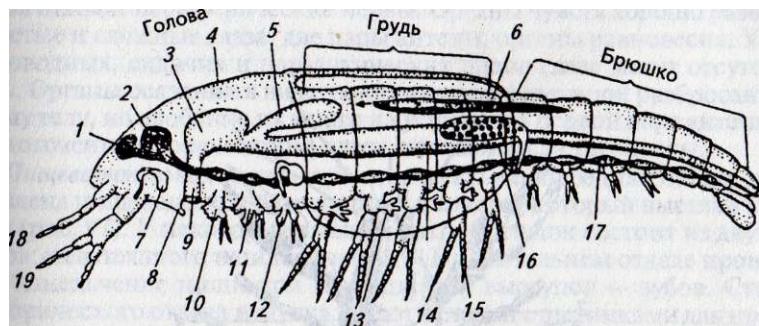


Рис. 95. Схема организации ракообразных:
1, 2—мозг; 3—желудок; 4—голова; 5—печень; 6—сердце; 7— почка; 8—губа; 9—рот;
10—мандибулы; 11, 12—максиллы; 13—эпиподит; 14, 15—экзоподит и эндоподит;
16—гонада; 17—брюшная нервная цепочка; 18—антенула; 19—антенна

Каждый сегмент груди несет по паре членистых конечностей, различающихся по форме и строению в зависимости от их функций.

Сегменты брюшка обычно не сливаются, и у высших раков каждый сегмент несет по паре ножек. У низших раков на брюшке ножек нет. Брюшко заканчивается анальной лопастью — тельсоном. У крабов брюшной отдел редуцирован. Конечности ракообразных выполняют самые разные функции (рис. 96). Они служат опорой при хождении, используются для плавания, захвата и измельчения пищи, защиты, при спаривании и т. п. Ноги имеют основную непарную часть (протоподит) и две ветви (наружная — экзоподит и внутренняя — эндоподит). Такая двуветвистая форма конечностей ракообразных сходна с двулопастной формой параподий многощетинковых кольчатых червей, но ноги ракообразных состоят из ряда члеников, что обеспечивает высокую подвижность этих животных.

Двуветвистые конечности, покрытые щетинками, характеризуются большой поверхностью и поэтому удобны для использования в качестве весел. У крупных раков ветви задней пары ног превратились в две широкие пластинки, которые вместе с широким последним члеником брюшка хорошо действуют при загребании воды брюшком.

Покровы. Наружным скелетом служит хитиновая кутикула, которая у высших раков пропитывается карбонатом кальция и превращается в прочный панцирь. У низших раков кутикула тонкая и прозрачная. Кутикула состоит из двух слоев: внутреннего — эндокутикулы и наружного — эктокутикулы. Эктокутикула пропитана дубильными веществами и обладает высокой прочностью. Эндокутикула во время линьки растворяется и всасывается гиподермой, а эктокутикула целиком сбрасывается. В состав хитиновой кутикулы входят разнообразные пигменты.

Нервная система представлена парным надглоточным узлом — головным мозгом, окологлоточным нервным кольцом и брюшной нервной цепочкой (у низших форм она в виде лестницы). От всех нервных

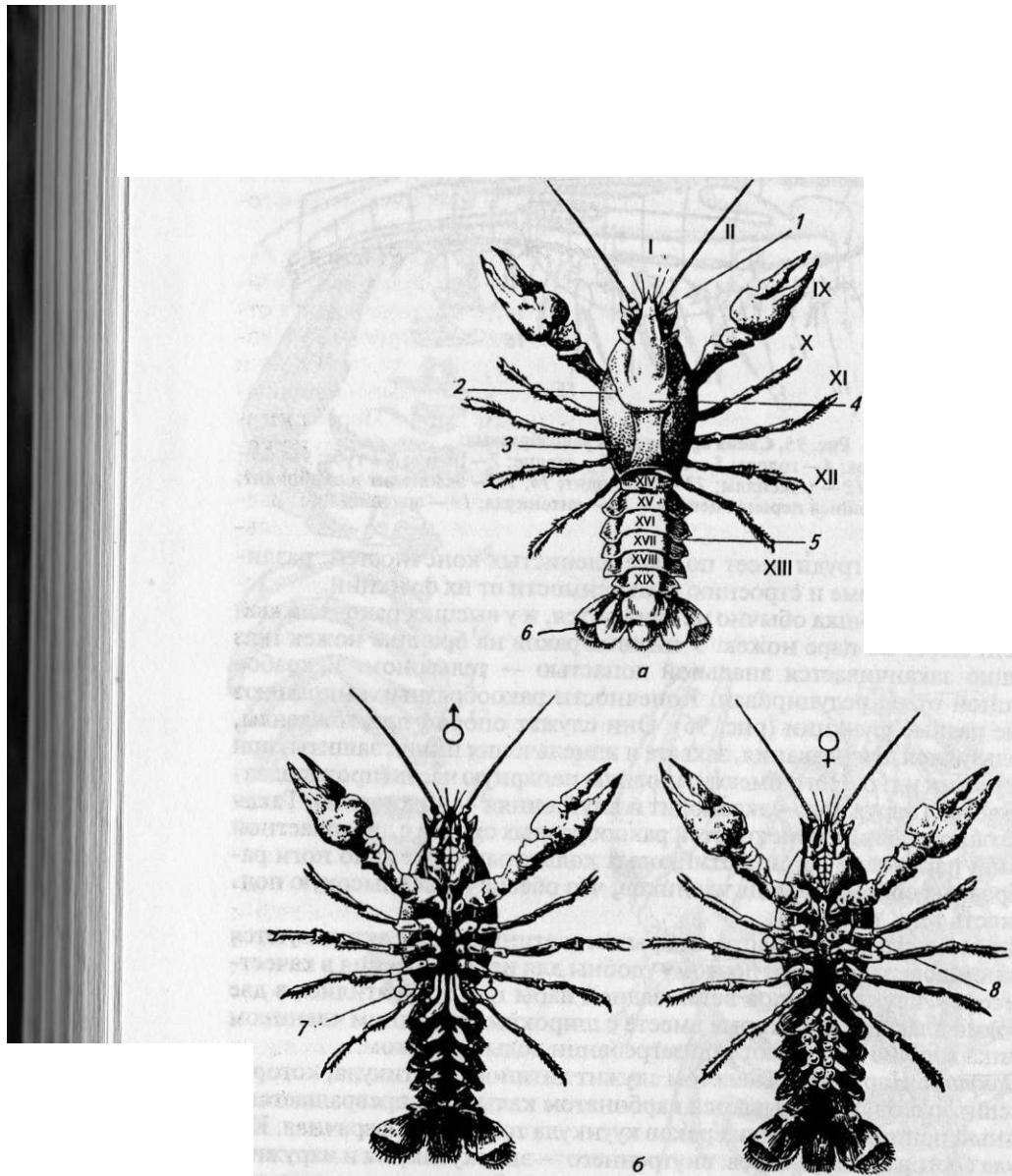


Рис. 94. Речной рак:
а — со спинной стороны; б — с брюшной стороны; 1 — рострум; 2 — головогрудной щит;
3 — края этого щита, покрывающие жабры; 4 — головогрудь; 5 — брюшко; 6 — хвостовой
плавник; 7 — мужское половое отверстие; 8 — женское половое отверстие; конечности и
сегменты: I — антеннула; II — антenna; IX—XIII — ходильные ноги; XIV—XIX — сегмен-
ты брюшка

у злов отходят периферические нервы. Органы чувств хорошо развиты: простые и сложные глаза, две пары антенн, органы равновесия. У глубоководных, сидячих и паразитических видов глаза могут отсутствовать. Органы осязания в виде чувствительных щетинок разбросаны по всему телу, но особенно их много на усиках. На первой паре антенн со средоточены хеморецепторы, здесь же находятся статоцисты.

Пищеварительная система. Эктодермальная передняя кишечная система представлена пищеводом, переходящим в желудок, который выстлан хитином (рис. 97). У некоторых высших раков желудок состоит из двух отделов: жевательного и пилорического. В жевательном отделе происходит измельчение пищи при помощи трех выступов — зубов. Стенки пилорического отдела желудка имеют складки с щетинками для процеживания мелкоизмельченной и жидкой фракций пищи. Переваривание и всасывание измельченной пищи происходит в относительно короткой эндодермальной кишке, куда открываются протоки печени, секрет которой выполняет функции сока поджелудочной железы. Задняя кишечка прямая, выстлана кутикулой и открывается наружу анальным отверстием.

Мелкие раки, ведущие планктонный образ жизни, захватывают частицы пищи с помощью усиков, ротовых конечностей, грудных ножек, создающих ток воды. У дафний задние грудные ножки бьют до 300 раз в 1 мин, обеспечивая постоянное поступление пищи в рот.

У ракообразных ротовые конечности выполняют разнообразные функции. Так, у раков хорошо развиты верхние челюсти — жвалы, имеющие зазубренный край для перетирания пищи. Две пары нижних челюстей также принимают участие в механической переработке пищи. Три пары ногочелюстей, расположенные на сегментах груди, удерживают добычу и подносят ее ко рту. Большие раки захватывают добычу первой парой ходильных ног, вооруженных мощными клемшами.

Органы дыхания. Большинство ракообразных дышат кожными жабрами, которые представляют собой придатки грудных конечностей и имеют вид тонких выростов у основания ножек. У высших раков жабры образуются не только на ногах, но и на стенке тела в жаберных полостях I юд хитиновым щитом — карапаксом. Движение воды около жабр происходит за счет движения ног, у основания которых они находятся, а также тех ножек, которые не имеют жабр. У сухопутных мокриц на брюшных ножках находятся глубокие ветвящиеся впаячивания — псевдодрагеи, в которых происходит газообмен, требующий повышенной влажности воздуха (до 90 %). Мелкие низшие раки дышат поверхностью тела. Сухопутным крабам также необходима высокая влажность воздуха.

Кровеносная система. У высших раков кровеносная система представлена мешковидным и вытянутым вдоль спинной стороны тела сердцем, имеющим отверстия (остии), через которые из полости тела засасывается кровь (гемолимфа). От сердца по артериям кровь выливается в лакуны миксоцеля тела. В полости тела гемолимфа отдает кислород тканям и насыщается диоксидом углерода. Частично кровь

отдает конечные продукты обмена почкам. В крови ракообразных содержится гемоцианин или гемоглобин, связывающие кислород. Из полости тела венозная кровь собирается в систему венозных сосудов и по жаберным приносящим сосудам поступает в систему капилляров в жабрах. Здесь кровь освобождается от диоксида углерода и насыщается кислородом. По выносящим жаберным сосудам кровь поступает в пикоардиальный синус, окружающий сердце.

У некоторых раков кровеносная система представлена только сердцем или кровь перемещается за счет работы мышц тела или за счет движения кишечника. Иногда у мелких раков кровеносная система может полностью редуцироваться.

Органы выделения. В головном отделе располагается две пары почек — видоизмененных и более сложно устроенных метанефридиев. Выводные протоки первой пары почек открываются у основания второй пары антенн (антеннальные железы), второй пары почек — у основания второй пары нижних челюстей — максилл (максиллярные железы). Каждая почка состоит из мешочка и выделительного канальца, который, расширяясь, может образовывать мочевой пузырь. У большинства ракообразных имеется лишь одна из двух пар почек — антеннальные или максиллярные.

Органы размножения. Большинство ракообразных раздельнополы. Встречается половой диморфизм. У немногих сидячих форм и паразитов наблюдается гермафроптизм. Развитие с метаморфозом разной степени сложности, реже развитие протекает без стадии личинки. У низших раков из яиц выходит личинка науплиус (рис. 98), имеющая три пары ног и один глаз. У высших морских раков из яиц выходит ли-

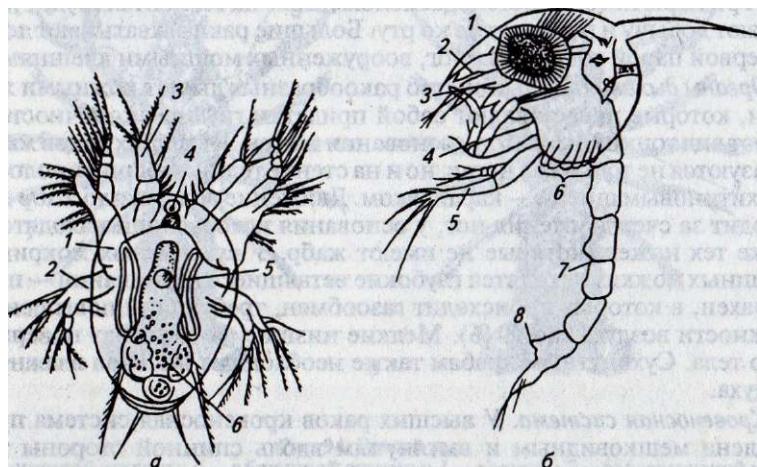


Рис. 98. Личинки раков:
 а — науплиус; 1,2,3 — конечности; 4 — простой глаз; 5 — орган выделения; 6 — кишечник; 6 — зоеа; 1 — сложный глаз; 2—5 — конечности; 6 — зарядки брюшных ног; 7 — брюшко; (У — последняя пара конечностей)

чинка зоэа, имеющая большее число сегментов и, следовательно, конечностей и два глаза. Кутикула личинок зоэа имеет шипики, увеличивающие ее поверхность и облегчающие личинкам плавание в толще воды.

Иногда у самок имеются семяприемники; некоторые самцы образуют сперматофоры, которые приклеивают к телу самок или вводят их в половые пути самок. У речного рака женские половые отверстия открываются на шестом грудном сегменте у основания третьей пары ходильных ног. Мужские половые отверстия расположены на восьмом грудном сегменте у основания пятой пары ходильных ног. У самцов две пары первых брюшных ножек превращены в копулятивные трубочки, с помощью которых самцы вводят сперму в половые отверстия самок.

Класс ракообразных делят на пять подклассов, из которых будут рассмотрены три: Жаброногие (Branchiopoda), Максиллоподы (Maxillipoda) и Высшие раки (Malacostraca).

ПОДКЛАСС ЖАБРОНОГИЕ (BRANCHIOPODA). К подклассу Жаброногие принадлежит 400 видов самых примитивных и мелких обитателей морей и пресных водоемов, у которых голова не срастается с грудными сегментами и отсутствуют конечности на брюшке. У жаброногих имеются сложные глаза и двуветвистые листовидные грудные ножки. Тельсон заканчивается вилочкой. Выделительные железы максиллярные. У основания ножек расположены жаберные лепестки; иногда жаброногие дышат через покровы тела. Развитие происходит с метаморфозом: из яйца вылупляется личинка науплиус, которая превращается во взрослую особь. Иногда развитие бывает прямым.

Наиболее типичными представителями жаброногих являются различные виды дафний (*Daphnia*) из подотряда Ветвистоусые раки (Cladocera) отряда Листоногие (Phyllopoda), в огромном количестве заселяющие пресные водоемы (рис. 99). Тело дафний мешкообразной формы и заключено в карапакс, напоминающий по форме двустворчатую тонкую хитиновую раковину. Створки карапакса приоткрыты с брюшной стороны. На голове имеется один фасеточный глаз и две пары антенн, из которых первая пара (антеннулы) небольших размеров. Вторая пара очень крупных антенн имеет двуветвистое строение и принимает участие в плавании. Взмахи антенн дают скачкообразные движения ветвистоусых, за что их называют водяными блохами.

Грудной отдел представлен четырьмя—шестью сегментами, несущими по паре коротких листовидных ножек, у основания которых расположены жаберные лепестки, а также ряды щетинок. Вода движется внутри створок раковины под действием ножек и омывает жабры. Пищевые частицы, увлекаемые током воды, отфильтровываются щетинками ног и отправляются в рот.

Наиболее распространена обыкновенная дафния (*Daphnia pulex*). В течение летнего периода дафнии размножаются партеногенетически, давая в потомстве только самок. Последние также дают партеногенетические поколения самок и т. д. Неоплодотворенные яйца откладывются в выводковую камеру, из них выходят молодые ракчи. Осенью при наступлении холодов самки откладывают порцию неоплодо-

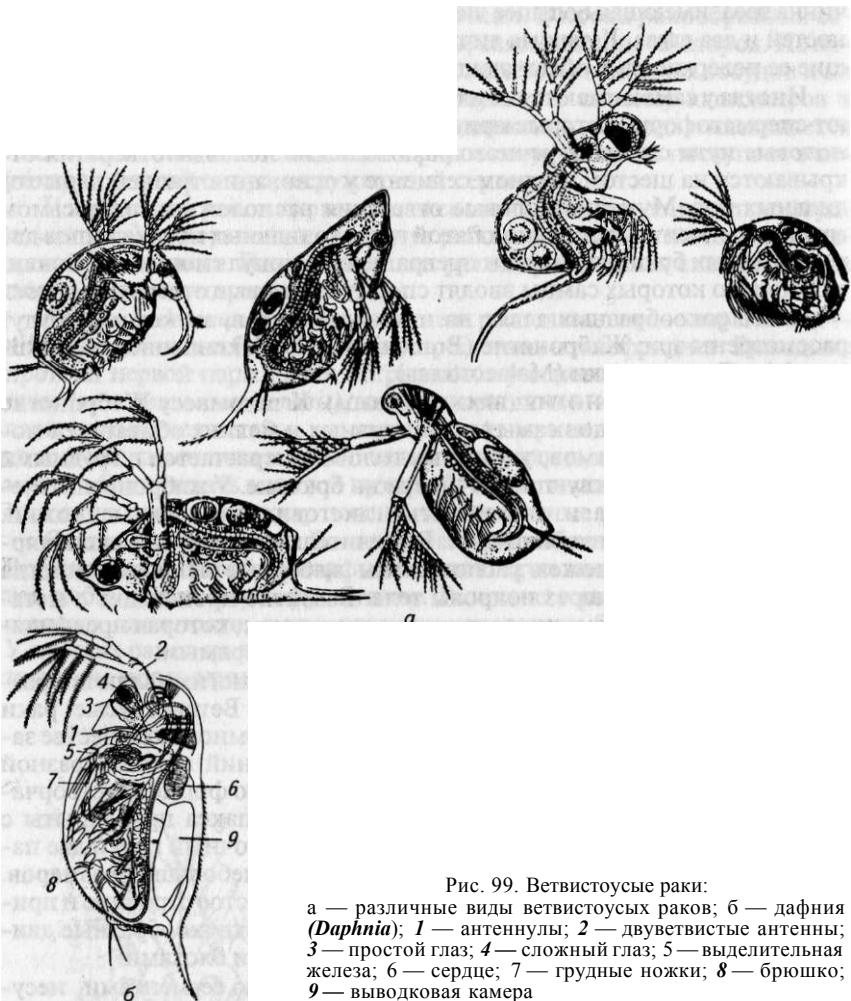


Рис. 99. Ветвистоусые раки:
а — различные виды ветвистоусых раков; б — дафния (*Daphnia*); 1 — антеннулы; 2 — двуветвистые антенны; 3 — простой глаз; 4 — сложный глаз; 5 — выделительная железа; 6 — сердце; 7 — грудные ножки; 8 — брюшко; 9 — выводковая камера

творенных яиц, из которых развиваются только самцы. Эти же самки через некоторое время откладывают в камеру вторую порцию неоплодотворенных яиц, которые претерпевают деление и получают гаплоидный набор хромосом. Эти яйца оплодотворяются молодыми самцами и в выводковой камере покрываются плотной оболочкой, образуя эфиоппий (в каждом по одному-два яйца). Эфиоппии зимуют, а весной из них снова появляются самки, которые дают новое поколение партеногенетических самок. Таким образом, у дафний происходит чередование партеногенетических и полового поколений, т. е. жизненный цикл протекает по типу гетерогонии.

ПОДКЛАСС МАКСИЛЛОПОДЫ (MAXILLOPODA). Представители подкласса Максиллоподы имеют упрощенную организацию: у них образуется головогрудь, дышат они поверхностью тела, у большинства нет сердца и сосудов. Мандибулы массивные. Максиллы представляют цедильный аппарат. Брюшных ножек нет. Мелкие ракчи со стройным удлиненным телом циклопы (*Cyclops*) и диаптомусы (*Diaptomus*) из отряда Веслоногие (Copepoda) движутся вперед с помощью двуветвистых грудных ножек, которые уплощены и снабжены щетинками. На голове расположены развитая первая пара одноветвистых антеннул и пара коротких антенн (рис. 100). Брюшко заканчивается вилочкой.

Циклопы и диаптомусы заселяют в большом количестве разнообразные водоемы и служат кормом для рыб и других обитателей водоемов. Размножаются эти ракообразные только половым путем. Оплодотворенные яйца самки склеивают в два яйцевых мешка, которые прикрепляют к нижней стороне первого брюшного сегмента. Развитие происходит со стадией науплиуса.

Известно около 1,8 тыс. видов веслоногих, живущих в морях и пресных водоемах. В морском планктоне наиболее многочисленны каланусы (*Calanus*). Они служат кормом для рыб и китов. Среди циклопов много фильтраторов, фитофагов и хищников. Яйца циклопов очень устойчивы к неблагоприятным условиям среды. Есть среди веслоногих ракообразных эктопаразиты рыб, некоторые являются промежуточными хозяевами плоских червей.

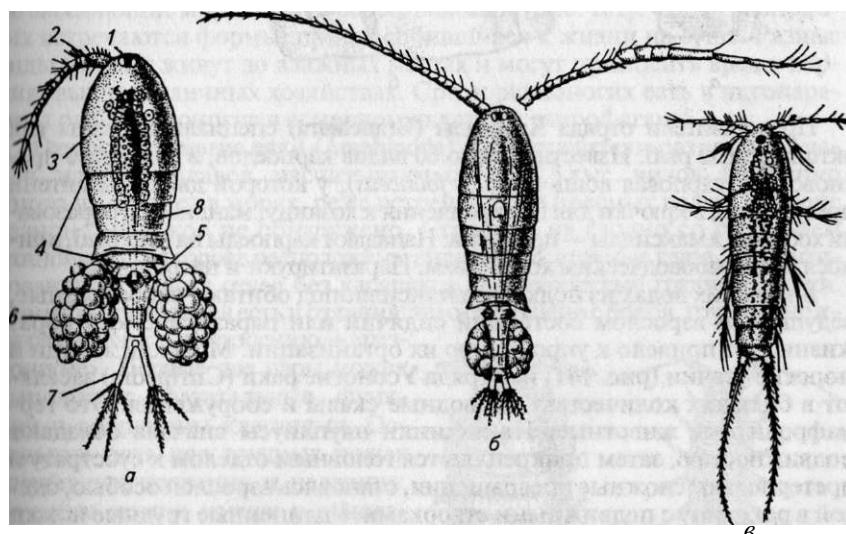
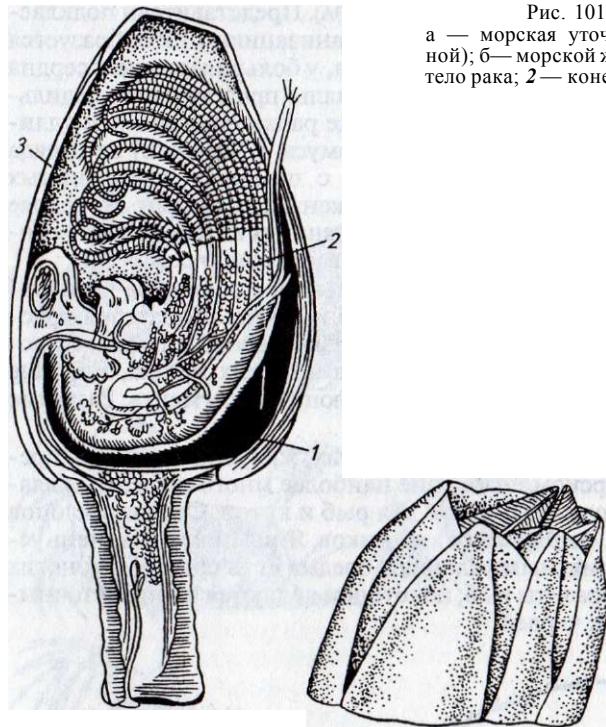


Рис. 100. Свободноживущие веслоногие ракообразные:
а - циклоп; б - диаптомус; в - кантокамптус; 1 — непарный глаз; 2 — первая пара антенн; 3 — головогрудь; 4 — свободные сегменты груди; 5 — брюшко; 6 — яйцевые мешки; 7 — вилочка брюшка; 8 — кишечник

Рис. 101. Усоногие раки:
а — морская уточка (со вскрытой раковиной); б— морской желудь (внешний вид); 1 —
тело рака; 2 — конечности; 3 — раковина



Представители отряда **Карпоеды (Branchiura)** специализированы как эктопаразиты рыб. Известно около 60 видов карпоедов, в том числе пресноводная карповая вошь (*Argulus foliaceus*), у которой две пары антенн превращены в крючки для прикрепления к хозяину, мандибулы образовали хоботок, а максиллы — присоски. Нападают карпоеды на всех рыб, принося вред рыбоводческим хозяйствам. Паразитируют и на лягушках.

В морских водах из подкласса максиллопод обитают ракообразные, ведущие во взрослом состоянии сидячий или паразитический образ жизни, что привело к упрощению их организации. Морские желуди и морские уточки (рис. 101) из отряда Усоногие раки (Cirripedia) заселяют в больших количествах подводные скалы и сооружения. Это гермафродитные животные. Их личинки науплиусы сначала обладают подвижностью, затем прикрепляются головным отделом к субстрату и претерпевают сложные превращения, становясь взрослой особью, одетой в раковину с подвижными створками. Удлиненные грудные ножки служат для передвижения воды в раковине, способствуя дыханию и процеживанию пищевых частиц. Наиболее многочисленны во всех морях морские желуди (*Balanus*) и морские уточки (*Lepas*). Они обрастают суда и подводные сооружения.

ПОДКЛАСС ВЫСШИЕ РАКИ (MALACOSTRACA). К этому классу принадлежат ракообразные средних и крупных размеров, населяющие морские и пресные водоемы; некоторые приспособились к жизни на суше. Их тело характеризуется постоянством сегментарного состава и состоит из акрона и четырех головных сегментов, восьми грудных и шести-семи брюшных сегментов. Часто головные и грудные сегменты сливаются, образуя головогрудь. Конечности имеются на всех сегментах, в том числе и на брюшных. Развитие происходит без превращения или со стадией зоэа. У низших раков развитие происходит со стадией науплиуса. Личинка зоэа в отличие от науплиуса имеет удлиненное и расчлененное тело. Рост раков сопровождается линьками. Так, речной рак в первый год жизни линяет около десяти раз, в течение второго года жизни число линек снижается до пяти, а на третий год жизни происходит всего две линьки. В последующем самцы линяют дважды, а самки — лишь раз в год. Практически рост раков прекращается к 5 годам, живут они до 20 лет.

Из 26 тыс. видов высших раков наибольший интерес представляют три отряда: Равноногие, Разноногие и Десятиногие раки.

Отряд Равноногие раки (Isopoda). В отряде 4,5 тыс. видов небольших раков с уплощенным телом. Передний грудной сегмент сливается с головным отделом. Глаза фасеточные. Карапакса нет. Сегменты груди и брюшка несут по паре коротких ножек. Ходильные грудные ножки одноветвистые и одинакового строения, что и определило их название «равноногие». Брюшные ножки двуветвистые и выполняют дыхательную функцию. К отряду равноногих относятся водяные ослики, обильно заселяющие морские и пресные водоемы (рис. 102). Среди равноногих встречаются формы, приспособившиеся к жизни на суше. Разные виды мокриц живут во влажных местах и могутносить вред в парниковых и тепличных хозяйствах. Среди равноногих есть и эктопаразиты рыб. Равноногие в основном являются сапрофагами.

Отряд Разноногие раки (Amphipoda). Представители разноногих раков, или бокоплавов, насчитывают около 4,5 тыс. видов. Особенно много их обитает в морях, реже встречаются в пресных водах. На суше разноногих раков не обнаружено. Строение их сходно со строением равноногих: на голове расположены сидячие фасеточные глаза, сегментированный грудной отдел без карапакса, одноветвистые грудные ножки. (Однако у бокоплавов есть и отличия: тело сплющено с боков, грудные ножки различаются по строению — «разноногие». Первые две пары ножек выполняют хватательные функции и вооружены клешнями. Остальные пять пар грудных ножек служат для ползания и плавания. Грудные ножки имеют у основания

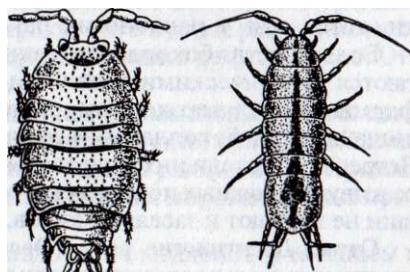


Рис. 102. Равноногие раки:
ч — мокрица; б — водяной ослик

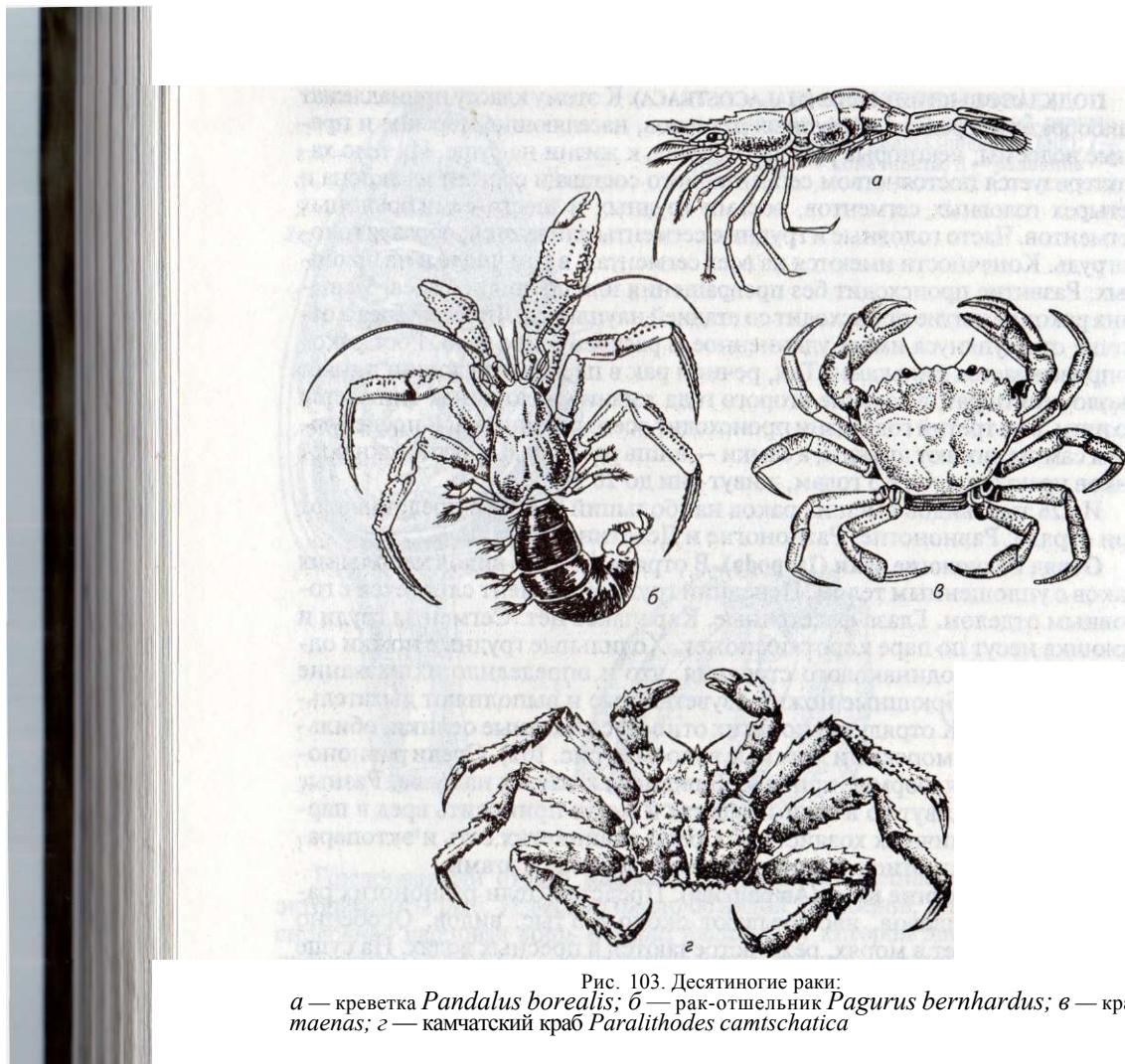


Рис. 103. Десятиногие раки:
а — креветка *Pandalus borealis*; б — рак-отшельник *Pagurus bernhardus*; в — краб *Carcinus maenas*; г — камчатский краб *Paralithodes camtschatica*

ния жаберный аппарат, поэтому сердце у бокоплавов находится в груди, тогда как у равноногих раков оно лежит в брюшном отделе.

Большинство бокоплавов — придонные ракообразные, которые питаются органическими остатками. Есть пелагические и планктонные формы. Они служат кормом для рыб. Во вновь созданные водохранилища и озера бокоплавов заселяют для укрепления кормовой базы. Встречаются среди них и паразиты: китовые вши (сем. Cyamidae) в массе живут на кожных покровах китов, вызывая их изъязвления. Китовые вши не плавают и заселяют китов, переползая с одного на другого.

Отряд Десятиногие раки (Decapoda). К ним относятся наиболее крупные и высокоорганизованные ракообразные (рис. 103), заселяю-

щие в основном моря и реже пресные воды, есть сухопутные формы. Насчитывают 8,5 тыс. видов, у которых сегменты головы и груди слиты в головогрудь, прикрытую с боков и сверху головогрудным хитиновым щитом — карапаксом. Хитин щита пропитан известью. Из восьми пар грудных конечностей три передние участвуют в захвате пищи; это так называемые ногочелюстии. Остальные пять пар — ходильные ноги, с их помощью раки передвигаются. Часто передняя пара ходильных ног заканчивается мощными клешнями. Брюшко представлено шестью сегментами, каждый из которых несет по паре ног.

Жабры располагаются не только на всех грудных ногах, но и на теле у основания ног. Карапакс закрывает тело с боков, образуя жаберные крышки. На брюшке имеется хвостовой плавник, образованный последним сегментом тела и широкими лопастями ножек предпоследнего сегмента. У некоторых десятиногих раков брюшко редуцировано. Развитие прямое или с метаморфозом.

Отряд десятиногих раков подразделяется на два подотряда: подотряд Плавающие раки (*Natantia*) и подотряд Ползающие раки (*Reptantia*). Зачастую десятиногих раков делят на три группы: длиннохвостые, мягкохвостые и короткохвостые.

Подотряд Плавающие раки (*Natantia*) включает наиболее примитивную группу десятиногих раков, ведущих плавающий образ жизни. Типичными представителями плавающих раков являются разнообразные креветки (*Pandalus*, *Crangon* и др.): тело их сплющено с боков, брюшко длинное и несет плавательные ножки. Грудные ножки тонкие и без клешней, с их помощью креветки плавают, дышат и захватывают пищу. Креветки являются объектом промысла. Сами они служат кормом для морских обитателей, особенно для рыб.

Подотряд Ползающие раки (*Reptantia*) — это прогрессивная группа десятиногих раков, в основном ведущих хищнический образ жизни, захватывая добычу клешнями. Брюшные ножки не выполняют плавательной функции и развиты слабо. У части раков брюшко существенно редуцировано (крабы). У представителей этого подотряда явно выраженная тенденция к передвижению ползанием, тогда как способность к плаванию существенно снижается. Подотряд подразделяется на отдельные: лангусты (*Palinura*), омары (*Astacura*), крабы (*Brachyura*) и отшельники, или крабоиды (*Anomura*). У омаров и лангустов брюшко мускулистое и длинное, они могут плавать. У отшельников брюшко асимметрично и недоразвито, раки прячут его в пустые раковины моллюсков или подгибают под себя. У крабов брюшко редуцировано. Отшельники и крабы плавать не могут.

Среди ползающих раков много ценных промысловых видов: лангуст (*Palinurus*), омар (*Homarus*), крабоид камчатский краб (*Paralithodes camtschatica*), крабы (*Cancer*, *Callinectes*), речной рак (*Astacus*). Промысел десятиногих раков широко развит: ежегодная мировая добыча приближается к 1 млн т.

Речные раки обитают в пресных водоемах с медленным течением и чистой водой. Ведут ночной и сумеречный образ жизни, питаясь дон-

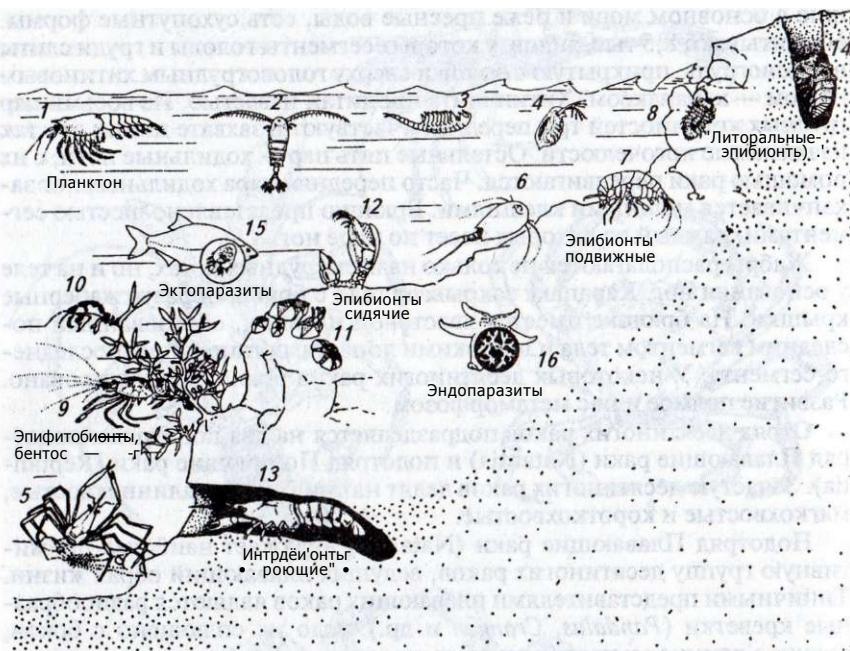


Рис. 104. Экологическая радиация ракообразных:
 1 — креветка; 2 — циклоп; 3 — жабромог; 4 — дафния; 5 — краб; 6 — щитень; 7 — бокоплав;
 8 — равноногий рак; 9 — креветка; 10 — равноногий рак; 11 — морские жемуди; 12 — морские
 уточки; 13 — десятиногий рак; 14 — роющая мокрица; 15 — карповая вошь; 16 — саккулина

ными животными и падалью. Самки вынашивают оплодотворенную икру, прикрепляя ее к ножкам брюшка. Речные раки и морские раки являются объектами промысла, а речного рака разводят в специализированных хозяйствах.

Филогения ракообразных. Ракообразные являются древней группой членистоногих животных. Предполагается, что каждый подкласс ракообразных имеет собственную линию развития от общих предков. Эти гипотетические предки обладали целым комплексом исходных признаков, которые проявляются у наиболее примитивных форм современных раков. Каждый из современных подклассов имеет эти признаки. У жаброногов Branchiopoda голова неслитная, брюшная нервная цепочка лестничного типа, сердце трубчатое. У представителей Maxillopoda примитивны двуветвистые головные конечности — антенны и челюсти, которыми они пользуются при передвижении. Только подкласс Ostracoda в своем эволюционном развитии утратил почти все примитивные признаки, и его современные представители практически лишиены их.

Таким образом, все подклассы ракообразных представляют потомков древней предковой группы Crustacea. Эволюция ракообразных

привела к образованию различных жизненных форм, занявших разнообразные экологические ниши (рис. 104). Исходным типом, видимо, были мелкие пелаго-бентосные формы, которые вели плавающий образ жизни. От этих форм специализация шла в нескольких направлениях: планктон, некton и бентос. Часть представителей приспособилась к паразитизму, а некоторые группы вышли на сушу, где положительные температуры и высокая влажность позволили им вести активный образ жизни без существенной перестройки органов дыхания.

ПОДТИП ХЕЛИЦЕРОВЫЕ (*Chelicerata*)

Хелицеровые — это особая ветвь членистоногих, по своим морфологическим характеристикам обособленная от других подтипов. Насчитывают около 63 тыс. видов современных хелицеровых — обитателей суши, представленных в основном паукообразными. Среди паукообразных встречаются вторичноводные виды, паразиты растений и животных. Для многих характерно выделение паутинных нитей из особых паутинных желез. Паутина помогает паукообразным в защите от врагов, в добывче пищи, расселении и т. п.

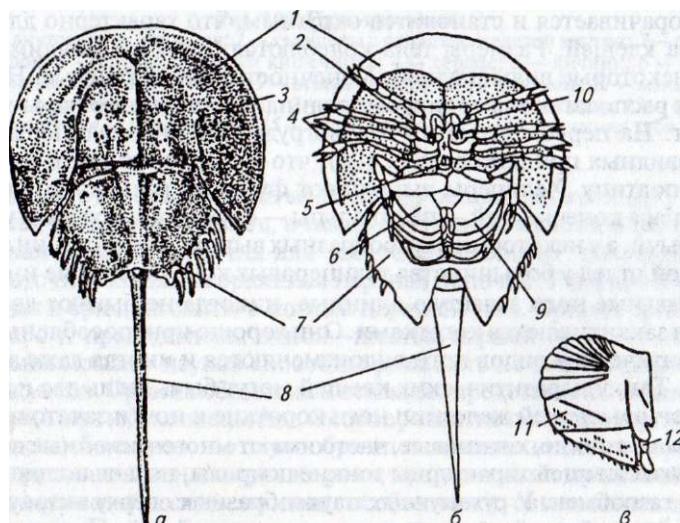


Рис. 105. Строение мечехвоста:
а — вид со спинной стороны; б — вид с брюшной стороны; в — жаберная ножка; 1 — головогрудной щит; 2 — рот; 3 — сложные глаза; 4 — ходильные ноги; 5 — хилярий; 6 — жаберная крышка; 7 — брюшной отдел; 8 — мечевидный отросток; 9 — жаброносные ножки; 10 — хелидеры; 11 — жаберный придаток; 12 — членистая ножка

Наиболее часто у хелицеровых сливаются сегменты головы и груди, образуя головогрудь. Это сильнее всего выражено у клещей. Антенные (усики) отсутствуют. Глаза простые, от одной до восьми пар. Функции усиков и челюстей выполняют первые две пары членистых пришатков: хелицыры и педипальпы. Имеется четыре пары ходильных ног. На брюшке обычно конечностей нет, у части сухопутных видов они видоизменены в половые пришатки, органы дыхания или в паутинные бородавки. Подтип Хелицеровые включает два класса: Мечехвосты (*Xiphosura*) и Паукообразные (*Arachnida*).

Мечехвосты представлены всего пятью ныне живущими видами (рис. 105), хотя в древности это была большая и широко распространенная группа водных животных. Мечехвосты имеют уплощенную головогрудь, закрытую панцирем, и широкое слитное брюшко, которое заканчивается мечевидным отростком. Ведут бентосный роющий образ жизни.

КЛАСС ПАУКООБРАЗНЫЕ (*Arachnida*)

Строение и жизненные отправления. Форма тела паукообразных весьма разнообразна. Тело состоит из головогруди и брюшка. Брюшко сегментированное, реже слитное, число сегментов достигает 12, и заканчивается брюшко тельсоном. У сильно расчлененных паукообразных (скорпионы, сольпуги) тело вытянутое, по мере слияния сегментов тело укорачивается и становится округлым, что характерно для большинства клещей. Размеры тела колеблются от долей миллиметра до 20 см (некоторые виды пауков). Конечности одноветвистые. На головогруди располагаются хелицыры, педипальпы и четыре пары ходильных ног. На первом сегменте головогруди имеются хелицыры в виде клешневидных пришатков (рис. 106), что и дало соответствующее название подтипу. Хелицыры выполняют функции размельчения пищи. Вторая пара конечностей — педипальпы — служат для захвата и удержания добычи, а у некоторых паукообразных выполняют функции антенн. Брюшной отдел у большинства хелицеровых конечностей не имеет.

Ходильные ноги зачастую длинные, никогда не бывают двуветвистыми и заканчиваются коготками. Они хорошо приспособлены к бегу. У паразитических видов ноги видоизменяются и иногда даже атрофируются. Так, у паразитических клещей могут быть лишь две передние пары ног, а у клещей железниц ноги короткие и почти зачаточные.

Покровы тонкие, хитиновые, часто имеют многочисленные волоски. Для мелких клещей характерны тонкие покровы, позволяющие осуществлять газообмен. У сухопутных паукообразных сверху экзокутикулы имеется тонкий слой эпикутикулы, в состав которой входят воскоподобные вещества, но нет хитина. Этот слой хорошо защищает тело от высыхания. В целом кутикула у паукообразных не бывает толстой. Лишь у вторичноводных представителей кутикула превращается в панцирь. У панцирных клещей кутикула также утолщается. К производным наруж-

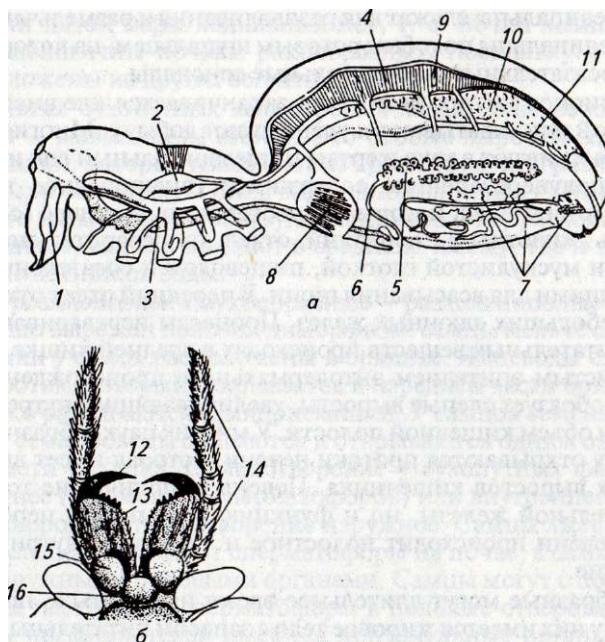


Рис. 106. Паук-крестовик:
а — схема внутреннего строения; 1 — хелицеры с ядовитой железой внутри; 2 — сосательный желудок; 3 — слепые отростки кишечника; 4 — сердце; 5 — яичник; 6 — яйцевод; 7 — паутинные железы; 8 — легкое; 9 — отверстие сердца; 10 — печень; 11 — мальпигиевые сосуды; б — хелицеры и педипальпы; 12 — когтевидный членник хелицер; 13 — основной членник хелицер; 14 — педипальпы; 15 — челюстная лопасть; 16 — нижняя губа

пого покрова относятся ядовитые железы у основания хелицер у пауков и ядовитая игла у скорпионов, а также паутинные железы пауков.

Нервная система типична для всех членистоногих: головной мозг, окологлоточное кольцо и брюшная нервная цепочка. У пауков и клещей узлы груди и брюшка слиты в единый нервный узел. Органы зрения развиты слабо и представлены одной—шестью парами простых глаз. Два центральных глаза у пауков способны различать не только форму, но и цвет предметов. Органы обоняния и осязания представлены отдельными сенсиллами или их скоплениями. Особое развитие получили органы осязания, так как большинство паукообразных ночные хищники и зрение не может играть в их жизни такую же роль, как у дневных представителей.

Пищеварительная система имеет особенности, связанные с характером питания. Паукообразные имеют две пары ротовых конечностей: хелицеры — челюсти и педипальпы — ногощупальца. Хелицеры состоит из основного членика и остального, загнутого крючка, который пауки вонзают в тело жертвы (см. рис. 106). Педипальпы состоят из основного членика, с которым соединен членистый щупик. У хищных пауков хе-

лидеры и педипальпы служат для раздавливания и размельчения пищи. У пауков педипальпы подобны ротовым щупальцам, на которых сосредоточены осязательные и обонятельные сенсиллы.

У скорпионов щупик педипальп заканчивается клешней, с помощью которой они схватывают и удерживают добычу. Многие паукообразные впрыскивают в тело жертвы пищеварительный сок и затем всасывают полупереваренное содержимое (внекишечное пищеварение). У представителей, которые питаются жидкой пищей (соки растений, кровь животных), передний отдел пищеварительного тракта представлен мускулистой глоткой, пищеводом и сосательным желудком, служащими для всасывания пищи. В передний отдел открываются протоки небольших слюнных желез. Процессы переваривания и всасывания питательных веществ происходят в средней кишке, выстланной железистым эпителием энтодермального происхождения. Средняя кишка образует слепые выросты, увеличивающие внутреннюю поверхность и объем кишечной полости. У многих паукообразных в среднюю кишку открываются протоки печени, которая имеет вид парных железистых выростов кишечника. Печень выполняет не только роль пищеварительной железы, но и функцию всасывания переваренной пищи. В печени происходит полостное и частично внутриклеточное пищеварение.

Паукообразные могут длительное время не питаться, поскольку в миксоцеле у них имеется жировое тело с запасом питательных веществ.

Кровеносная система представлена лежащим на спинной стороне тела мускулистым сердцем и отходящими от него сосудами, которые идут к различным органам. Обратный ток крови осуществляется по лакунам. У клещей часть кровеносных сосудов редуцирована, иногда отсутствует и сердце. У паукообразных, имеющих легочные мешки, кровеносная система развита лучше, чем у трахейнодышащих.

Органы дыхания. У одних паукообразных это легочные мешки, у других — трахеи, у третьих — легочные мешки и трахеи одновременно (большинство пауков). У водных хелицеровых органы дыхания представлены жабрами. У некоторых мелких форм газообмен осуществляется через покровы тела. Легочные мешки расположены в передней части брюшка и открываются наружу специальными отверстиями — дыхальцами. В легочных мешках кровь течет в параллельно расположенных тонких листовидных складках, через которые и происходит газообмен: в щелевидные пространства между складками проникает воздух, отдающий кислород в гемолимфу сосудов, которые пронизывают листовидные складки легочных мешков.

Трахеи — наиболее распространенные органы дыхания у паукообразных — начинаются отверстиями, или стигмами, в покровах брюшка; от стигм в глубь тела расходятся трахеи в виде ветвящихся трубочек. Трахеи и легкие паукообразных возникли независимо друг от друга. Легочные мешки более древние органы, чем трахеи.

Органы выделения представлены коксальными железами (почками), которые открываются выделительными отверстиями у основания

третьей или пятой пары ходильных ног, т. е. почки хелицеровых не вполне гомологичны почкам ракообразных, поскольку у последних они расположены на других сегментах тела.

Для многих сухопутных хелицеровых характерны особые органы выделения — *мальпигиевы сосуды*. Это особые выросты задней части средней кишki, которые извлекают из крови продукты распада и отводят их в среднюю кишку. Они в виде одной-двух пар слепых трубочек небольшого диаметра способствуют рациональному расходованию воды в организме, так как впадают в среднюю кишку, где и происходит всасывание излишков воды.

Органы размножения. Паукообразные — раздельнополые животные; у них хорошо выражен половой диморфизм: самцы мельче самок. Партеногенные яичники у самок расположены в брюшке. Яйцеводы сливаются в единый проток, который открывается в передней части брюшка. У самок нередко развиваются семяприемники. У самцов семенники лежат в брюшке, семяпроводы сливаются и открываются одним отверстием в нижней части брюшка. Оплодотворение у сухопутных форм наружновнутреннее (с помощью сперматофоров) или внутреннее. У водных форм хелицеровых оплодотворение наружное. Самцы лжескорпионов и многих клещей оставляют сперматофоры на почве, а самки захватывают их наружными половыми органами. Самцы могут с помощью хелицер сами переносить сперматофоры вальные отверстия самок. У некоторых представителей имеются копулятивные органы, и в этом случае сперматофоры у самцов не образуются.

Большинство паукообразных откладывают яйца, у некоторых наблюдается живорождение. Чаще всего развитие происходит без метаморфоза и сопровождается ростом и неоднократными линьками. У клещей иногда наблюдается партеногенетическое размножение, а развитие происходит со стадией личинки.

Класс паукообразных подразделяется на множество отрядов, из которых будут рассмотрены наиболее интересные для сельскохозяйственных работников отряды: Скорпионы (*Scorpiones*), Ложноскорпионы (*Pseudoscorpiones*), Сольпуги, или Фаланги (*Solifugae*), Сенокосцы (*Opiliones*), Пауко! (*Aranei*) и три отряда клещей (*Acariformes*, *Parasitiformes*, *Opiliocarina*).

Отряд Скорпионы (*Scorpiones*). Это наиболее древние по происхождению паукообразные. Длина тела у тропических видов может достигать 18 см. Для скорпионов характерна наибольшая расчлененность тела. За головогрудью следует 6-сегментное переднебрюшье и 6-сегментное заднебрюшье (рис. 107). Дышат скорпионы легочными мешками. Тельсон образует вздутие с ядовитой иглой, на вершине которой открываются протоки ядовитых желез. Крупные педипальпы вооружены клешнями. Жертву скорпион захватывает клешнями педипальп, перегибает брюшко через спину вперед и вонзает иглу в добычу. Четыре пары ходильных ног заканчиваются парой коготков.

Известно около 600 видов скорпионов, обитающих в странах с теплым климатом. Этоочные хищники, днем они скрываются в расщепах

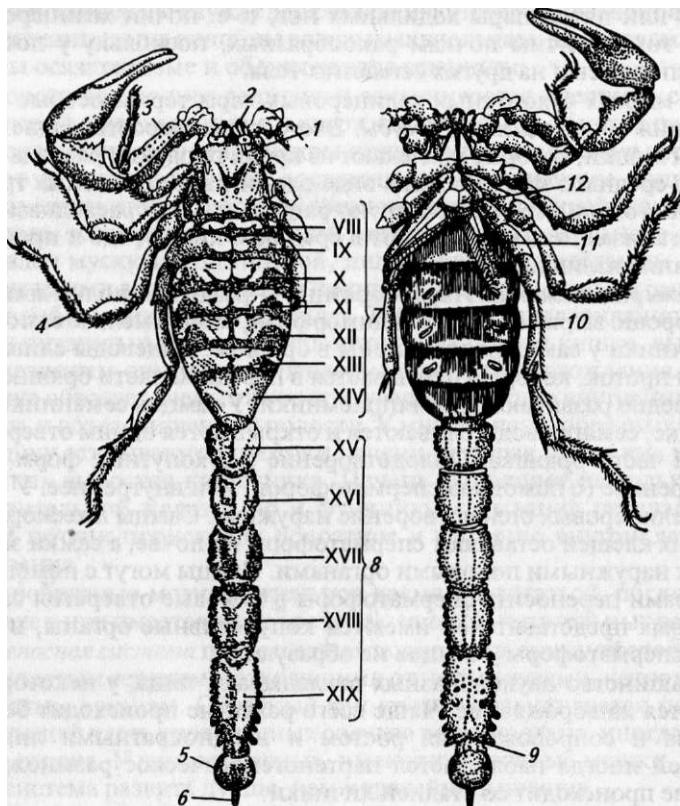


Рис. 107. Скорпион:
 а — вид с спинной стороны; 1 — головогрудь; 2 — хелицеры; 3 — педипальпы; 4 — ноги;
 5 — конечный членок брюшка; 6 — жало; 7 — передний отдел брюшка; 8 — задний отдел
 брюшка; (VIII—XIX — сегменты брюшка); б — вид с брюшной стороны; 9 — анальное от-
 верстие; 10 — легочная щель; 11 — гребенчатые органы; 12 — половые крылышки

линах и норках. Самки рождают детенышей и первое время носят их на спине. Укусы болезненны, но обычно не опасны для человека. В Закавказье обитает пестрый скорпион (*Buthus eureus*).

Отряд Ложноскорпионы (Pseudoscorpiones). Это мелкие (1—7 мм) паукообразные, имеющие клешневидные педипальпы и поэтому напоминающие скорпионов (рис. 108). Брюшко не разделено на передне- и заднебрюшье. Дышат ложноскорпионы с помощью трахей. На хелице-рах открываются протоки паутинных желез. Живут ложноскорпионы в лесной подстилке, под камнями и корой пней, в жилище человека. Питаются мелкими клещами. Самцы откладывают сперматофоры, которые захватываются самками в семяприемники. Яйца самка откладывает

ет в выводковую камеру, расположенную на брюшной стороне тела. Вышедшие из яиц личинки остаются подвешенными к камере и питаются желтком, который выделяет самка. После первой линьки молочные покидают мать.

Известно около 1,3 тыс. видов ложноскорпионов. В жилищах человека часто встречается книжный ложноскорпион (*Chelifer cancroides*), пытающийся мелкими насекомыми и клещами, которые вредят книгам.

Отряд Сольпуги, или Фаланги (Solifugae). Это крупные и сильно расчлененные обитатели степных и пустынных районов, насчитывающие около 600 видов. Головогрудь сольпуг неслитная и состоит из головного отдела и трех свободных сегментов, из которых последний недоразвит (см. рис. 108). Брюшко состоит из десяти сегментов. Педипальпы

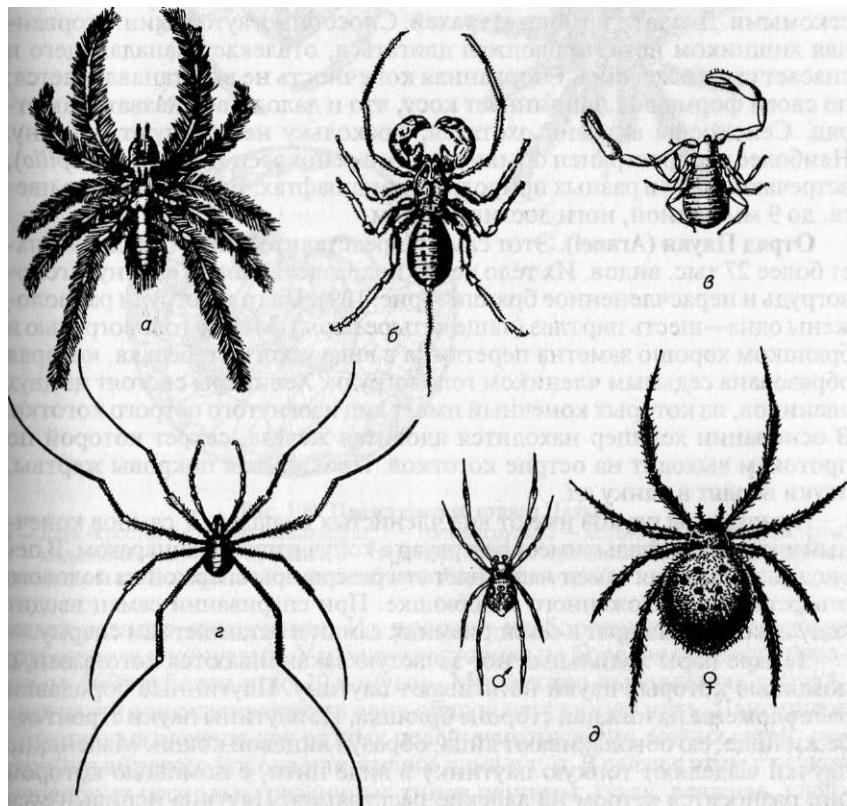


Рис. 108. Различные паукообразные:
а — сольпуга *Galeodes araneoides*; б — хвостатый телифон *Thelyphonus caudatus*; в — книжный лжескорпион *Chelifer cancroides*; г — обычный сенокося *Phalangium opilio*;
д — караукрут *Latrodectus tredecimguttatus*

похожи на ходильные ноги, участвуют в передвижении и выполняют чувствующие функции. Дышат с помощью трахей. Паутинных бородавок нет. Сольпуги питаются насекомыми, в том числе вредными (саранча), не ядовиты. Ведут ночной образ жизни. На Кавказе обитает фаланга *Galeodes araneoides*, имеющая в длину до 5 см. Самка откладывает яйца в норку и проявляет заботу о потомстве. Самец откладывает сперматофоры.

Отряд Сенокосцы (Opiliones) включает более 3 тыс. видов широко распространенных членистоногих, внешне похожих на пауков. Сенокосцы отличаются от пауков отсутствием перетяжки между головогрудью и брюшком, членистостью брюшного отдела, в котором десять сегментов, и клешневидными, а не крючковидными (как у пауков) хелицерами (см. рис. 108). Тело сенокосцев покоится на очень длинных и тонких ногах.

Сенокосцы повсеместно обитают на поверхности почвы, в трещинах коры деревьев, стенах строений и т. п., питаясь ночью мелкими насекомыми. Дышат с помощью трахей. Способны к аутотомии: оторванная хищником нога, продолжая двигаться, отвлекает нападающего и спасает своего хозяина. Оторванная конечность не восстанавливается; по своей форме она напоминает косу, что и дало повод назвать так отряд. Сенокосцы активно охотятся, поскольку не образуют паутину. Наиболее распространен обыкновенный сенокосец (*Phalangium opilio*), встречающийся в разных природных ландшафтах. Его тело бурого цвета, до 9 мм длиной, ноги достигают 5 см.

Отряд Пауки (Aranei). Этот самый представительный отряд включает более 27 тыс. видов. Их тело четко подразделяется на слитную головогрудь и нерасчлененное брюшко (рис. 109). На головогруди расположены одна—шесть пар глаз (чаще четыре пары). Между головогрудью и брюшком хорошо заметна перетяжка в виде узкого стебелька, которая образована седьмым члеником головогруди. Хелицеры состоят из двух члеников, из которых конечный имеет вид изогнутого острого коготка. В основании хелицер находится ядовитая железа, секрет которой по протокам выходит на острие коготков. Прокалывая покровы жертвы, пауки вводят в ранку яд.

Педипальпы пауков имеют вид членистых щупалец. У самцов конечный членик педипальп имеет резервуар с копулятивным аппаратом. В период размножения самец наполняет эти резервуары спермой из полового отверстия, расположенного на брюшке. При спаривании самец вводит копулятивный аппарат в семяприемник самки и оставляет там сперму.

Четыре пары ходильных ног зачастую заканчиваются коготками, с помощью которых пауки натягивают паутину. Паутинные бородавки расположены на нижней стороне брюшка. Из паутины пауки строят себе жилище, ею обволакивают яйца, образуя яйцевой кокон. Маленькие паучки выделяют тонкую паутинку в виде нити, с помощью которой они разносятся ветром на далекие расстояния. Паутинна используется для ловли добычи, с помощью паутинных гамаков самцы заполняют спермой свои семенные капсулы (резервуары) на педипальпах.

Паутинные железы выделяют клейкое тянущееся вещество, которое затвердевает на воздухе. У пауков имеется несколько типов паутинных

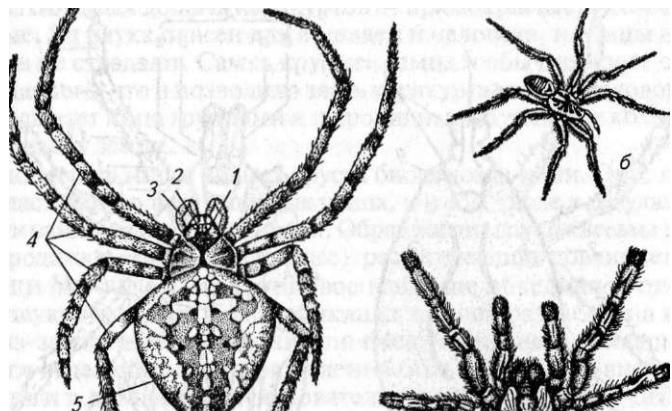


Рис. 109. Представители отряда Пауки:
 а — паук-крестовик *Araneus diadematus*; 1 — головогрудь; 2 — хелицеры; 3 — педипальпы;
 4 — ходильные ноги; 5 — брюшко; б — тарантул *Lycosa singoriensis*; в — каракурт (самка в
 гнезде у яйцевых коконов); г — паук-птицеед *Poecilotheria regalis*

желез разного назначения. Их протоки на бородавках открываются трубочками и конусами. У паука-крестовика на бородавках насчитывают до 560 трубочек и до 20 конусов. Множество выделяемых тончайших паутинок склеиваются в одну общую паутинную нить. Паутинные ииты предназначены для разных целей: изготовления ловчих сетей, постройки яйцевого кокона или жилого дома и т. п. В связи с этим у пауков существует несколько различных типов паутины: сухая, влажная, клейкая, гофрированная, для основания и поперечин ловчих сетей и т. п.

У большинства пауков имеется пара легких и пара трахей (подотряд Двулегочные), у некоторых тропических пауков есть только две пары легочных мешков. В нашей стране обитает около 1,5 тыс. видов пауков,

Рис. 110. Морфология клеша *Hyalomma* (самец) с дорсальной и вентральной сторон:
1 — средняя бороздка; 2 — глаза; 3 — латеральные бороздки; 4, //—лапка; 5 — присоски
и коготки; 6 — пальпы; 7 — футляр хелицер; 8 — хоботок; 9 — основание хоботка; 10 —
членики пальп (а, б, в, г); 12 — передняя лапка; 13 — голень; 14 — бедро; 15 — вертлуг;
16 — половое отверстие; 17 — половые бороздки; 18 — анальное отверстие

относящихся к двулегочным паукообразным. Наиболее часто встречаются пауки-крестовики (*Araneus*), которых легко узнать по крестообразному рисунку на верхней стороне брюшка (см. рис. 109). Они заселяют леса, кустарники, приусадебные участки и постройки. На ветвях пауки-крестовики сооружают большие радиальные сети, нити которых покрыты клейкой массой, захватывающей попавших в сети насекомых.

Домовые пауки (*Tegenaria*) селятся обычно в жилых и производственных помещениях, натягивая горизонтальные паутинные сети, с помощью которых они ловят мух и других насекомых.

Особую группу образуют пауки, преследующие свою жертву. Обычным представителем на юге страны в степных районах является тарантул (*Lycosa singoriensis*). Живет тарантул в отвесных норках, вырытых в почве и выстланных паутиной. Это самый крупный паук отечественной фауны — до 3 см длиной (см. рис. 109). Укус тарантула вызывает у человека болезненный отек, но не представляет серьезной опасности.

Для человека весьма опасен паук *каракурт* (*Latrodectus tredecimguttatus*), которого можно встретить в степных районах Кавказа и Поволжья. Это средних размеров (около 1,5 см) паук черного цвета с красными пятнышками на верхней поверхности брюшка (рис. 110). Каракуры обитают в норках, а на поверхности почвы расстилают ловчую

паутину. Обычная добыча каракуртов — прямокрылые, в том числе саранчовые. Яд паука опасен для лошадей и человека, но овцы и свиньи от укусов не страдают. Самка крупнее самца и обычно после спаривания съедает его, что и позволило звать каракурта «черной вдовой». Самка откладывает яйца группами в шаровидные паутинные коконы, расположенные у земли.

Пауков очень много во всех ярусах биоценозов суши. Они, как хищники, участвуют во всех пищевых цепях, и в том числе в регуляции численности самых разных насекомых. Образ жизни пауков весьма разнообразен: бродяжки, сидячие (тенетные), растягивающие ловчие сети и др.

КЛЕЩИ (ACARINA). Это групповое название объединяет три отряда класса паукообразных. Классификация клещей разработана недостаточно из-за скудности сведений по филогении, неравномерной изученности отдельных групп, появления большого числа вновь открытых видов и т. д. Многие исследователи не без оснований считают, что в классе Паукообразные следует выделить два подкласса: Пауки (Агапеа) и Клещи (Acari). От других подклассов клещи отличаются слабой выраженностью и полным отсутствием сегментации брюшного отдела тела. Постэмбриональное развитие клещей в отличие от других паукообразных проходит с метаморфозом: личинка имеет три пары ног.

Общая характеристика клещей. По приспособленности к различным местам обитания клещи приближаются к насекомым. Они живут во мхах и в лишайниках, лесной подстилке, в почвенном слое они могут составлять 95 % фауны членистоногих. Клещи живут во всех водоемах, даже в горячих источниках вулканов. Заселили они многие органы и ткани растений, разнообразны их отношения с растениями и животными. Многие растительноядные клещи наносят серьезный вред сельскохозяйственному производству. Потери урожая достигают 20—30 % в тепличных хозяйствах из-за паутинного клеша, вредящего овощным культурам. Повреждения, наносимые грушевым галловым клещом, приводят к потери до 90 % урожая. Земляничный клещ снижает сбор ягод на 40—70 %. Большой ущерб наносит зернопродуктам группа амбарных клещей. Часть клещей является паразитами животных и человека.

Покровы клещей, как и покровы всех членистоногих, состоят из кутикулы, гиподермы и подстилающей ее базальной перепонки. Тело некоторых клещей покрыто сверху мощным панциревидным щитом (панцирные клещи). У кровососущих иксодовых клещей покровы слабо склеротизированы, что позволяет самкам сильно растягиваться во время сосания крови. К производным наружных покровов относятся щетинки, щитки и железы.

Нервная система. Характерной особенностью клещей является то, что их центральная нервная система представляет собой единую цельную массу нервной ткани, окружающую пищевод плотным кольцом — **мозгом**. У клещей обнаружены чувствующие органы, связанные с механическим, химическим, гигротермическим чувствами и зрением. Основу органов чувств составляют сенсиллы, состоящие из щетинок,

пор и т. д. У клещей простые глаза, как правило, их две пары. Представители большого числа видов лишены глаз.

Пищеварительная система. Передняя кишечка подразделяется на мускулистую глотку и пищевод, который впадает в среднюю кишечку. У части клещей пищевод образует расширение — зоб. Средняя кишечка, часто называемая желудком, у многих видов связана со слепыми отростками, которые особенно хорошо развиты у кровососущих клещей. Их насчитываются до семи пар, и они значительно превышают размером саму среднюю кишечку. Небольшие отростки имеют сапрофаги.

Сложно устроена средняя кишечка у питающихся соком растений клещей, обладающих системой разделения пищи на фракции. Задний отдел кишечника состоит из тонкой и толстой кишечек, между которыми впадают в кишечник мальпигиевые сосуды. Очень короткая прямая кишечка выстлана хитином и заканчивается анальным отверстием. Для отдельных клещей характерно внекишечное пищеварение.

Ротовые части представлены хелицерами и педипальпами и могут образовывать два типа ротовых аппаратов: грызущий и колюще-сосущий. Ротовой аппарат грызущего типа характерен для клещей, питающихся твердой растительной пищей. С переходом к питанию жидккой пищей ротовые части становятся тонкими и теряют зубцы, образуя стилеты. Парные стилеты в виде двух желобков составляют трубку, которая выдвигается вперед и погружается в ткань растения.

Строение ног. Большинство клещей имеют четыре пары ног, а их личинки — три пары ног. Встречаются виды с редукцией части ног. Ноги состоят из шести члеников: тазика, вертлуга, бедра, колена, голени, лапки. Отдельные членики иногда сливаются с образованием четырех- или пятичлениковой конечности. Лапка может иметь различные приспособления для передвижения: пару коготков (у многих видов), присоски (у эмподий), расположенные между коготками, липкий секрет для удержания на субстрате, паутинные выделения и т. д. У водных клещей ноги превращаются в плавательные. Могут быть задние — прыгательные, могут быть приспособления для удержания на волосах, перьях и т. п. (см. рис. 110).

Кровеносная система лакунного типа. У большинства видов отсутствуют не только сосуды, но и сердце. Кровь бесцветная.

Органы дыхания. Крупные клещи дышат с помощью трахей. Мелкие формы дышат через покровы тела. Дыхальца — стигмы — парные, всего их от одной до четырех пар.

Органы выделения — мальпигиевые сосуды и коксальные железы.

Органы размножения. Все клещи раздельнополы. Часто наблюдается половой диморфизм по многим признакам, в том числе и по размерам тела. Половое отверстие обычно находится на уровне задней пары ног на брюшной стороне тела. У самки и самца оно зачастую прикрыто специальными щитками или клапанами, покрытыми сенсорными щетинками. Сперматофоры вводятся в половое отверстие самок с помощью хелицер самца. У некоторых клещей самцы прикрепляют сперматофоры к субстрату. У панцирных клещей имеется длинный яйцеклад.

Большинство клещей откладывают яйца, и только у немногих отмечено живорождение. Для ряда панцирных клещей характерно посмертное живорождение. В этом случае самка погибает, не отложив яиц, и они развиваются внутри ее тела. Вылупившиеся личинки, защищенные от неблагоприятных условий панцирем матери, питаются ее тканями, а затем выходят наружу. У клещей встречается партеногенез.

В течение онтогенеза клещи проходят четыре фазы: яйца, личинки, нимфы и взрослого клеша (имаго). По окончании эмбриогенеза личинка выходит из яйца. У некоторых видов еще в яйце происходит эмбриональная линька зародыша. Вышедшая личинка отличается от взрослого клеша меньшим числом щетинок, меньшими размерами, отсутствием полового отверстия и последней пары ног. Личинки хищных и паразитических видов не питаются и живут за счет запасов желтка. Закончившая развитие личинка впадает в состояние покоя, линяет и превращается в нимфу.

Нимфа имеет четыре пары ног, на ее теле появляются зачатки гениталий, увеличивается число щетинок. Фаза нимфы может включать от одного до трех возрастов: нимфа первого (протонимфа), нимфа второго (дейтонимфа) и третьего (тритонимфа) возраста. Переход из одного возраста в другой сопровождается периодом покоя и линькой.

При наступлении неблагоприятных условий нимфа первого возраста переходит в особую стадию *гипопуса*. Гипопусы значительно отличаются от обычных нимфальных фаз клещей: они лишены действующего ротового аппарата, у них сильно редуцируется пищеварительный аппарат и существуют они за счет резервов тела. Гипопусы бывают подвижные (расселительные) и покоящиеся (рис. 111).

Подвижные гипопусы сохраняют развитые ноги, покровы их уплотняются, тело уплощается. Передвигаются активно и пассивно, прикрепляясь к различным животным с помощью присосок или других приспособлений.

Для покоящихся гипопусов характерна сильная редукция ног (мучной клеш) или их полная утрата (волосатый домовый клеш). В состоянии покоя они могут находиться месяцами и устойчивы даже к действию фунгицидов. В благоприятных условиях гипопус линяет и превращается в нимфу, которая после стадии покоя и линьки превращается во взрослого клеша.

У многих клещей, у которых хорошо развит половой диморфизм, самцы развиваются быстрее самок. Поэтому к моменту последней линьки самки самцы уже достигают половой зрелости. Вскоре после сбрасывания самкой личиночной шкурки происходит спаривание, а через 2—3 сут самки начинают откладку яиц.

Жизненные циклы клещей разнообразны, зависят от многих факторов. Есть виды паутинных клещей, которые в течение года могут дать до 20 поколений. В северных ареалах на развитие одной генерации иксодовых клещей требуется до 4 лет.

В жизненном цикле клещей могут быть неблагоприятные периоды. Их это случае все процессы жизнедеятельности клещей резко замедля-

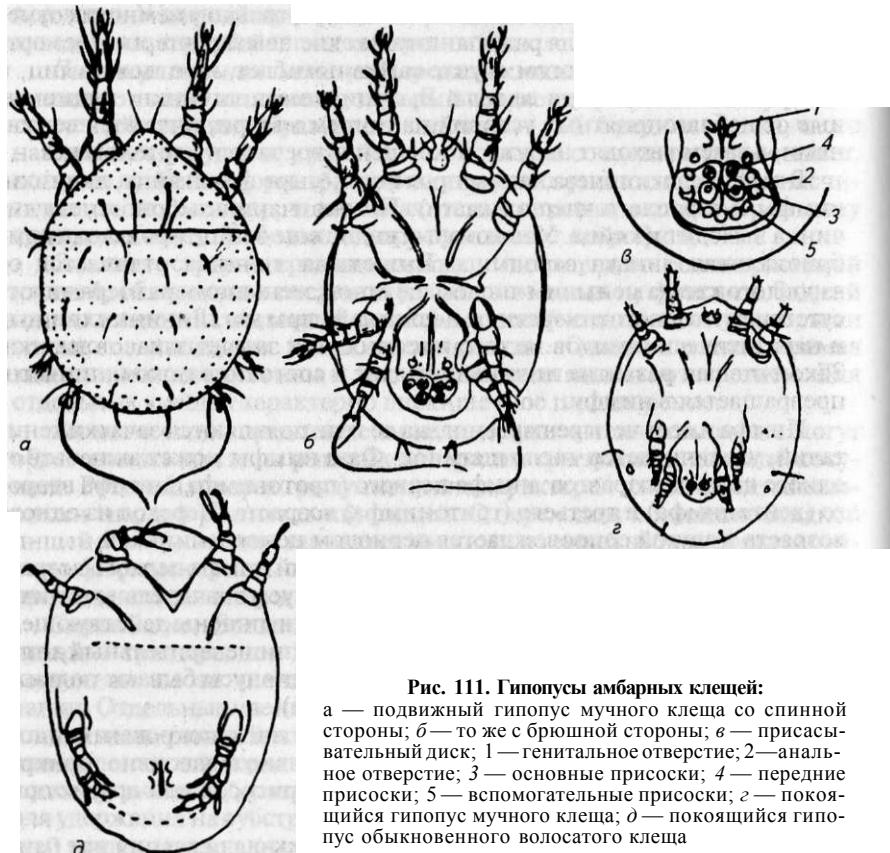


Рис. 111. Гипопусы амбарных клещей:
 а — подвижный гипопус мучного клеща со спинной стороны; б — то же с брюшной стороны; в — присасывательный диск; 1 — генитальное отверстие; 2 — анальное отверстие; 3 — основные присоски; 4 — передние присоски; 5 — вспомогательные присоски; г — покоящийся гипопус мучного клеща; д — покоящийся гипопус обыкновенного волосатого клеща

ются и они переходят в состояние диапаузы. В отличие от обычного покоя (оцепенения) диапауза находится в сложных взаимоотношениях с внешней средой, так как именно эти взаимоотношения обеспечивают синхронность развития организма клещей с фенологией питающих растений и климатическими условиями среды. Диапауза может быть зимней и летней.

Некоторые виды клещей ведут хищнический образ жизни, питаясь другими клещами и мелкими насекомыми. Поражая клещей-вредителей и паразитов сельскохозяйственных растений, хищные формы клещей приносят существенную пользу и могут быть использованы в биологической борьбе с вредными членистоногими. Многие почвенные клещи, являясь сапрофагами, участвуют в почвообразовательных процессах.

Отряд Клещи-сенокосцы (Opiliocarina) включает небольшую группу примитивных клещей с сохранившейся сегментацией тела: два последних сегмента головогруди свободные и брюшко из восьми сегментов. У

них две пары глаз, четыре пары стигм, хелицеры клешневидные, а между ними расположен теркообразный орган. Первая пара ног специализирована в основном как сенсорные органы. Размеры тела сенокосцев достигают 1 мм. Живут скрытно, обитая под камнями, в почвенном слое. Их хозяйственное значение неопределено.

Отряд Акариформные клещи (Acariformes). Это наиболее крупный отряд, объединяющий разнообразные по морфологическим и экологическим особенностям формы и насчитывающий более 15 тыс. видов. Наряду с микроскопическими мелкими паразитическими видами (менее 0,1 мм длины) встречаются свободноживущие хищники, достигающие в длину 10 мм. Примитивные формы дышат через покровы тела, а у эволюционно прогрессивных форм дыхание осуществляется с помощью трахей. Размножение сперматофорами. Развитие с анаморфозом. В состав отряда входят два подотряда: Краснотелковые (*Trombidiformes*) и Саркоптоидные (*Sarcoptiformes*), включающие множество семейств.

Подотряд Краснотелковые (*Trombidiformes*) включает много семейств свободноживущих сухопутных и водных клещей, имеются среди них и паразиты. Хелицеры колющие. Одна пара дыхальца расположена в передней части тела.

Из растительноядных клещей существенный вред растениям причиняют паутинные клещи из сем. *Tetranychidae*. Это мелкие клещики (0,3—0,5 мм) разнообразной окраски. Большинство видов выделяют паутину, что и дало название семейству. Под слоем паутины на нижней стороне листьев растений клещи образуют колонии. Из яиц выходят личинки, которые проходят стадии нимфы первого и второго возрастов и затем становятся взрослыми клещами. Весь цикл развития длится 12—28 сут. Клещи питаются соком растений, который высасывают из листьев. Особенно большой вред приносят в тепличных и парниковых хозяйствах (рис. 112). Серьезными вредителями являются обыкновенный паутинный (*Tetranychus urticae* Koch), садовый паутинный (*Schizotetranychus* Ond.) и другие.

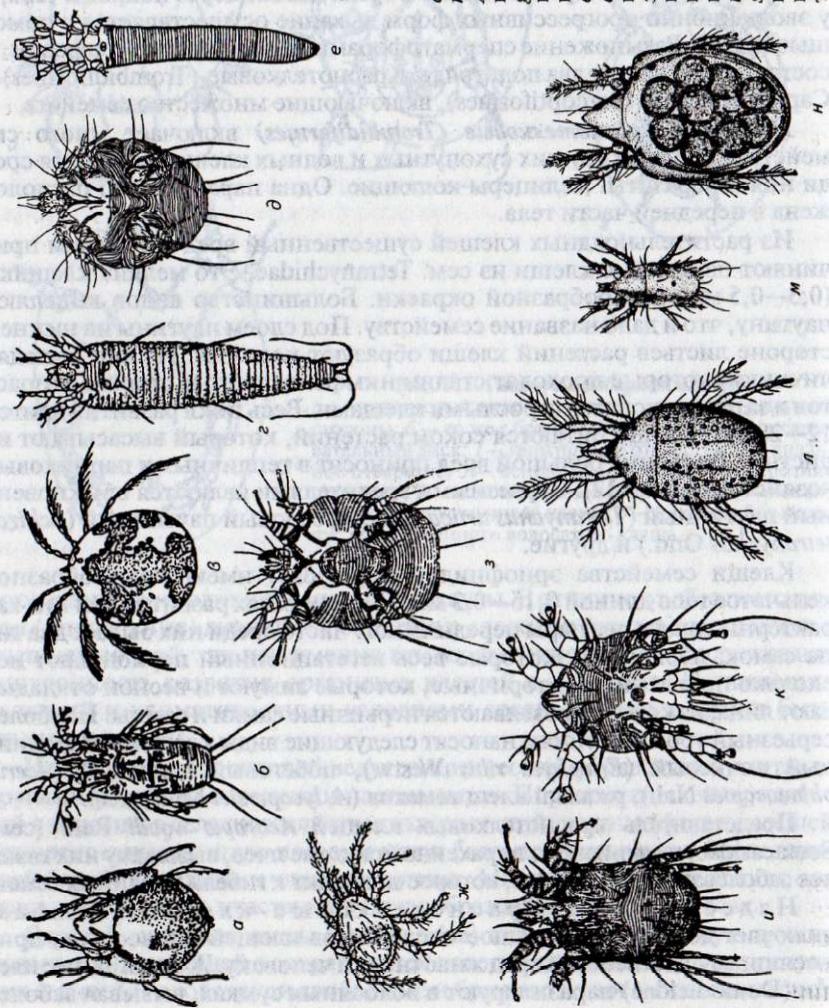
Клещи семейства эриофиид (*Eriophyidae*) имеют червеобразное кольчатое тело длиной 0,15—0,3 мм. На всех фазах развития для них характерны только две пары передних ног. Часто среди них бывает два типа самок: первичные, которые весь вегетационный период дают несколько поколений, и вторичные, которые зимуют и весной откладывают яйца, из которых развиваются первичные самки и самцы. Наиболее серьезный урон растениям наносят следующие виды клещей: смородинный почковый (*Eriophyes ribis* Westw.), побеговый слиновый (*Aceria phloeocoptes* Nal.), ржавый клещ томатов (*A. lycopersici* Mas.) и др.

Представитель краснотелковых клещей *Acarapis woodi* Ren. (сем. *Scutacaridae*) паразитирует в трахейной системе пчел, вызывая у них тяжелое заболевание акарапидоз, которое приводит к гибели пчелиных семей.

Надсемейство Краснотелковые клещи (Trombea) включает достаточно большое число видов клещей-кровососов, приносящих существенный вред животным и человеку. Железничные клещи (*Demodicidae*) паразитируют в волосяных сумках, вызывая заболе-

Рис. 112. Клещи:

a — панцирный клещ *Galumna muscicola*; *b* — первоый клещ *Aneloposis passerinus*; *c* — волчаной клещ *Hydrachna geographica*; *d* — четырехногий клещ *Eriophyes*; *e* — чесоточный зудень *Sarcopis scadiei*; *f* — железнина угря *Demodex felicis*; *g* — группный клещ *Poecilococcus necrotrophi*; *h* — зудневый клещ; *i* — на кожниковый клещ; *k* — клещ коксед; *l* — обыкновенный паутинный клещ *Tetranychus cinnabarinus*; *m* — мучной клещ *Acarus siro*; *n* — панцирный (орбатидный) клещ, внутри которого находится цистицеркоиды ленточного черва мониезии



вание — железницу у животных и человека (см. рис. 112). При сильном поражении на 1 см² кожи овец насчитывали до 25 тыс. особей клещей. В пораженных местах шерсть выпадает, в дерме кожи образуются пузырьки.

Подотряд Саркоптоидные (Sarcoptiformes) включает клещей, имеющих грызущий ротовой аппарат. Среди большого числа семейств следует выделить панцирных клещей (*Oribatei*) — участников почвообразовательных процессов, среди которых есть виды, являющиеся промежуточными хозяевами паразитических ленточных червей.

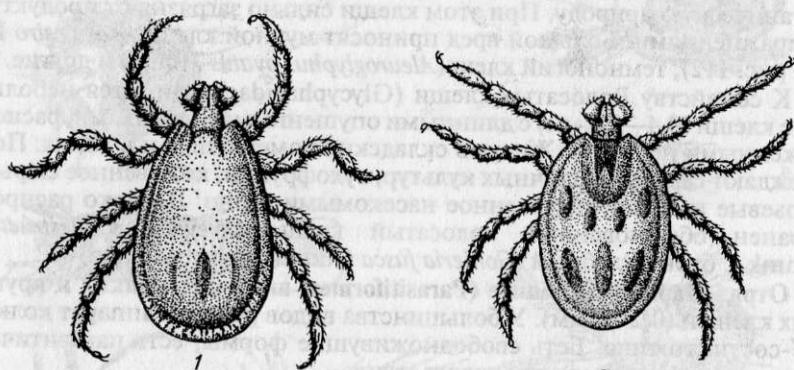
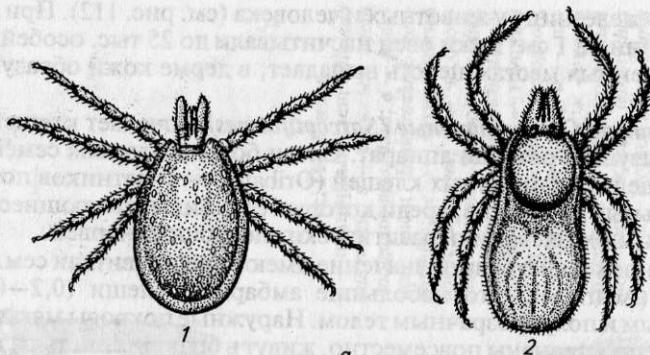
Важное хозяйственное значение имеют представители сем. Мучные клещи (Acaridae). Это небольшие амбарные клещи (0,2—0,8 мм) с овальным и полупрозрачным телом. Наружные покровы мягкие и гладкие. Распространены повсеместно, живут в опавших листьях, в скирдах сена и соломы, в зернохранилищах, складах, гнездах птиц и т. д., где повреждают зернофураж, муку, сухофрукты и многое другое, имеющее органическую природу. При этом клещи сильно загрязняют продукты испражнениями. Большой вред приносят мучной клещ (*Acarus siro* L, см. рис. 112), темноногий клещ (*Aleuroglyphus ovatus* Troup.) и другие.

К семейству Волосатые клещи (Glycyphagidae) относятся небольшие клещи (0,4—0,6 мм) с длинными опущенными щетинками, расположеннымми на спине. Живут в складских помещениях и в домах. Повреждают семена масличных культур, сухофрукты, кожевенное сырье, перьевые изделия, пораженное насекомыми зерно. Широко распространен обыкновенный волосатый клещ (*Glycyphagus destructor* Sehrnk.), бурый хлебный (*Gohieria fusca* Oud.) и другие клещи.

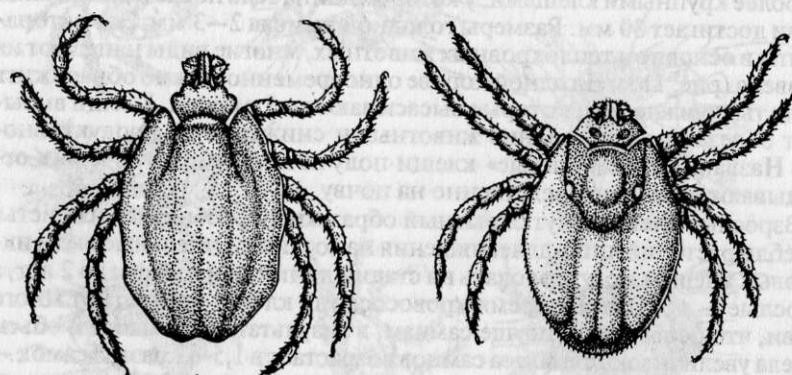
Отряд Паразитiformные (Parasitiformes) включает мелких и крупных клещей (0,2—7 мм). У большинства видов ротовой аппарат колюще-сосущего типа. Есть свободноживущие формы, есть паразитические виды и вредители продовольствия.

Семейство Иксодовые, или Пастбищные клещи (Ixodidae), относящиеся к подотряду Заднедыхальцевые (Metastigmata), представлено наиболее крупными клещами, у которых длина тела после насасывания крови достигает 30 мм. Размеры голодной нимфи 2—3 мм. Это эктопаразиты в основном теплокровных животных, многие виды нападают на человека (рис. 113). На одной корове одновременно можно обнаружить около тысячи клещей, которые высасывают до 5 л крови. Клещи вызывают болезненное состояние животных и снижение их продуктивности. Название «пастбищные» клещи получили потому, что самки откладывают яйца непосредственно на почву

Взрослые клещи ведут скрытный образ жизни, взбираясь на листья и стебли растений лишь для нападения на хозяина. Отдельные виды иксодовых клещей могут голодать на стадии личинки и нимфи до 2 лет, а взрослые — до года. Во время кровососания клещи поглощают много крови, что особенно присуще самкам, в результате чего масса и объем их тела увеличиваются: масса самцов возрастает в 1,5—2 раза, а самок — в 100 раз. Большинство видов в период развития и во взрослом состоянии имеют разных хозяев. По биологическому признаку различают од-



6



3

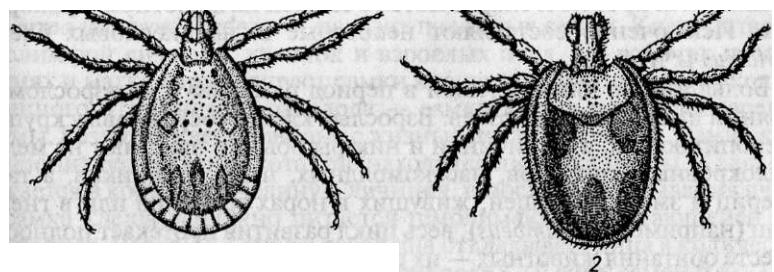


Рис. 113. Представители клещей подотряда Ixodidae:
a — Ixodes ricinus; б — Haemaphysalis punctata; в — Boophilus calcaratus; г — Dermacentor dagestanicus; д — Rhipicephalus tyrranicus. 1 — самец; 2 — самка

но-, двух- и треххозяиных клещей. Однохозяиные клещи все стадии своего развития проходят на одном хозяине. Так, личинки *Boophilus calcaratus* нападают на крупных животных и после насыщения кровью линяют, оставаясь на этом же хозяине, и превращаются в нимфу. Нимфы питаются и превращаются на этом же хозяине в имаго. Лишь в фазе срытых половозрелых особей клещи самопроизвольно отпадают со своего единственного хозяина и откладывают в почве яйца.

У двуххозяиных клещей на теплокровное животное нападают личинки, а с него отпадают уже срытые нимфы. Нимфы в почве линяют, превращаются в имаго, которые нападают на второго хозяина и после насыщения кровью становятся половозрелыми клещами.

Треххозяиный тип питания (большинство *Ixodes*) заключается в том, что клещи нападают на животных в каждой фазе развития, каждый раз отпадая и линяя во внешней среде. Половозрелый клещ должен насосаться крови, так как лишь после этого у него созревают половые продукты. Насасывание кровью сопровождается у самок быстрым развитием яичников, что позволяет обеспечивать высокую плодовитость. После этого клещи спариваются и самки откладывают яйца, число ко-

торых достигает нескольких тысяч. У большинства видов клещей самцы питаются кровью и спариваются с самкой на животном-прокормителе. Исключение составляют некоторые виды иксодовых клещей (*I. ricinus* L. и др.).

Большинство видов клещей в период развития и во взрослом состоянии имеют разных хозяев. Взрослые клещи сосут кровь у крупных млекопитающих, а их личинки и нимфы обычно нападают на мелких теплокровных: грызунов, насекомоядных, птиц, хищников, а также ящериц и змей. У клещей, живущих в норах грызунов или в гнездах птиц (например, *I. plumbeus*), весь цикл развития протекает полностью в месте обитания животных — их прокормителей. Паразитический образ жизни обуславливает тесную биологическую связь клещей с различными животными. Снижение численности животных-прокормителей угрожает выживанию самих клещей.

Иксодовые клещи в основном являются теплолюбивыми паукообразными, поэтому по мере продвижения на север число их видов снижается. В холодных регионах уживаются виды родов *Ixodes* и *Dermacentor*. Иксодовые клещи (*Ixodes persulcatus* Sch. и *I. ricinus* L.) передают вирус весенне-летнего (клещевого) энцефалита, возбудителей туляремии и прочих опасных заболеваний, другие виды клещей могут передавать гемоспоридиоз крупного рогатого скота, пироплазмоз, бруцеллез, сыпной тиф и другие опасные заболевания домашних и диких животных и человека (см. рис. 113).

Аргасовые клещи (сем. Argasidae, п/отр. Metastigmata) — облигатные кровососы. Весь жизненный цикл их протекает в норах, гнездах, пещерах и других укрытиях позвоночных животных. Кладка яиц и линька происходят вне хозяина. Самки откладывают до сотни яиц многократно, после каждого питания (за всю жизнь около тысячи штук). Метаморфоз длится до года. Живут клещи 15—25 лет. Могут длительное время не питаться — до 10 лет. У них более резко по сравнению с иксодовыми клещами выражен временный характер связи с хозяином. Многие из видов пребывают на хозяине от нескольких минут до 2 ч, а личинки — до нескольких суток (3—11). Клещи ведут ночной образ жизни, скрываясь днем недалеко от своих хозяев: в щелях полов и стен зданий, норах, гнездах птиц и т. п. Птичий, или персидский, клещ (*A. persicus* Oken.) вызывает сильное истощение и даже гибель домашних птиц, особенно молодняка. Этот клещ может быть хранителем и переносчиком заболеваний гусей, уток и кур, например спирохетоза птиц.

Мелкие гамазовые клещи имеют короткий (12—15 сут) цикл развития и поэтому могут давать десятки поколений. Эти влаголюбивые клещи распространены повсеместно. Наиболее богатый видовой состав характерен для лесной зоны. Клещ *Dermahyssus gallinae* обитает в птичниках, гнездах голубей и даже в клетках декоративных птиц. Это гнездовой паразит, облигатный кровосос, может длительно голодать. Самки питаются кровью на курах ночью. После каждого насыщения они заползают в трещины помещений и откладывают по 12—20 яиц. Раз-

множается клещ весной и летом, нанося вред птицеводству: снижается яичная продуктивность и могут быть летальные исходы.

Varroa jacobsoni Oudem. — паразит пчелиных семей. Клещ питается гемолимфой личинок, куколок и взрослых пчел. На рабочих пчелах, трутнях и матке паразитируют самки клеща, а на личинках и куколках пчелиного и трутневого расплода — самки клеща и неполовозрелые формы. Развитие клеща связано с жизнью пчелиной семьи. Самки клеща проникают в ячейку сотова, где находится личинка пчелы. Клещ погружается в корм под пчелинью личинку, и после запечатывания ячейки самка клеща усиленно питается гемолимфой развивающейся личинки пчел. Через 2—3 сут самка клеща откладывает яйца в количестве 5—6 штук. Цикл развития клеща составляет 8—9 сут у самок и 6—7 сут у самцов. Каждая самка за свою жизнь может сделать до трех яйцекладок. Самцы после оплодотворения самок погибают.

Самки клеща, отродившиеся весной и летом, живут до 3 мес, а самки осеннего выплода — до 7 мес. Вне пчелиного гнезда самки клеща могут выживать не более недели, на трупах пчел — не более 11 сут. Сильная зараженность клещом может приводить к гибели пчелиных семей.

Гамазовые перьевые клещи (надсемейство *Analgoidea*) — широко распространенные паразиты птиц. Насчитывают около 1,2 тыс. видов высокоспециализированных паразитов, живущих на перьях и коже птиц. На одном виде птиц может паразитировать несколько видов перьевых клещей, обитающих на разных участках оперения. Питаются клещи отмершими частичками эпидермиса кожи и перьев, но главная их пища — жировая смазка оперения птиц. Особенности перьевых клещей в том, что самец спаривается с телеонимфой, которая затем линяет и превращается в оплодотворенную самку. Яйца приклеиваются к бородкам пера. Вред от таких клещей невелик, но некоторые виды поражают кожу. Так, кожный зудень (*Knemidocoptes mutans*) живет под чешуями неоперенной части ног кур, вызывая заболевание, которое называют «известковые ноги».

Саркоптоидные зудневые (чесоточные) клещи (надсемейство *Sarcoptoidae*) живут в коже домашних и диких млекопитающих, а также человека, вызывая заболевание — чесотку. Это очень мелкие клещи с короткими ножками. Самки чесоточного зудня питаются кожей, прогрызая в ее роговом слое извитые ходы до 15 мм длиной. В толще кожи самка откладывает до 50 яиц. Размещает она их в ходах, над которыми выгрызает вентиляционные отверстия. Личинки и протонимфы живут в этих ходах, питаясь остатками поврежденной самкой кожи и тканевой жидкостью. Развитие длится около двух недель. Протонимфы превращаются в телеонимф и выходят на поверхность кожи. Здесь некоторые из них превращаются в самцов и спариваются с женскими телеонимфами — будущими самками. Оплодотворенные телеонимфы вгрызаются в кожу и превращаются в самок, а самцы выгрызают в коже небольшие ходы и прячутся там. Во внешней среде зудни не размножаются и сохраняют подвижность не более двух недель.

Клещи вызывают воспаление кожи, которая утолщается, и на ней образуются струпья и роговые корки. Под корками скапливается гнойный экссудат. Волосы на этих участках выпадают. Клещи рода *Sarcoptes* являются возбудителями саркоптоза у лошадей, мулов и ослов (*S. equi*), свиней (*S. suis*), овец (*S. ovis*), крупного рогатого скота (*S. bovis*), кроликов (*B. cuniculi*), собак (*S. canis*) и лисиц (*S. vulpis*). По морфологическим признакам эти виды клещей сходны между собой. При переходе на неспецифического хозяина зудни могут размножаться и вызывать кратковременное заболевание — псевдочесотку.

Клещи рода *Notoeders* — возбудители нотоэдроза кошек, собак, лисиц, крыс (*N. cati*) и кроликов (*N. cuniculi*). Эти клещи могут переходить и на человека.

Накожные клещи (сем. *Psoroptidae*, род *Psoroptes* Gerv.) паразитируют на эпидермальном слое кожи овец и других домашних животных. Это более крупные по сравнению с саркоптoidными клещами научно-образные. Длина их тела достигает 1 мм. Впиваясь в кожу хозяина с помощью хоботка, клещи сосут кровь, вызывая воспалительные процессы и выпадение шерсти. Сильное заселение может привести к истощению и даже гибели овец.

Развитие псороптидов проходит по фазам: яйцо, личинка, протонимфа, телеонимфа, имаго. Эти накожники адаптировались к паразитированию у овец (*P. ovis*), у крупного рогатого скота (*P. bovis*), у лошадей (*P. equi*) и у кроликов (*P. cuniculi*). На неспецифических хозяевах эти клещи не размножаются. Они постоянные паразиты животных; выпадая во внешнюю среду, они быстро гибнут. Самки откладывают яйца на поверхность кожи, прикрепляя их маточным секретом. Продолжительность жизни самок около 2 мес.

Клещи рода *Chorioptes* — кожеды, питаются отслоившимися клетками эпителия у лошадей, рогатого скота, кроликов. Самки откладывают яйца на волосы животных. Клещи рода *Otodectes* в морфологическом отношении сходны с клещами *Chorioptes*, но у *Otodectes* не развита последняя пара ног. Клещ *O. cynotis* является возбудителем ушной чесотки у собак, кошек и пушных зверей.

Клещи — регуляторы численности вредных беспозвоночных. Многие свободноживущие представители подотряда краснотелок (отр. Acariformes) являются хищниками и поэтому широко используются для борьбы с вредными беспозвоночными животными. Клеши из семейства Стигмеиды (Stigmeidae) питаются клещами и мелкими насекомыми, в том числе бродяжками щитовок, паутинным клещом и красным плодовым клещом.

Интересен обычный хищный клещ *Cheyletus eruditus* Schrnk., который регулирует численность амбарных клещей в зернохранилищах. Довольно крупные клещи из сем. Краснотелки (Trombidiidae) окрашены в красный цвет. Их нимфы и взрослые клещи обычно покрыты многочисленными щетинками, которые придают им бархатистый вид. Среди краснотелок имеются виды, личинки которых паразитируют на насекомых, а нимфы и взрослые фазы питаются яйцами и личинками

этих насекомых. Так, *Eutrombidium trigonum* Herm. является врагом саранчовых, *Allotrombium fuliginosum* Негш. уничтожает яйца стеблевого мотылька и питается тлями и медяницами, а *A. filiginosum* Herm. уничтожает яйца колорадского жука.

Наиболее широко распространенный *Hemisarcoptes malus* Schim. (п/отр. Sarcoptiformes, сем. Hemisarcopidae) питается яйцами и бородяжками щитовок. Этот вид клеща был интродуцирован в Канаду, и в некоторых районах он успешно регулирует размножение яблонной щитовки.

Среди представителей фитосейидов (сем. Phytoseiidae, п/отр. Mesostigmata) много мелких (0,2—0,6 мм) хищных клещей, которые питаются клещами — вредителями растений, особенно в садах, не подвергающихся частой обработке пестицидами. В нашу страну были интродуцированы: *Metaseiulus occidentalis* Nesb.; устойчивый к инсектоакарицидам и испытываемый на плодовых культурах); канадский клещ фитосейилюс (*Phytoseiulus persimilis* Ath.—Henr.) для борьбы с паутинными клещами в теплицах. Этого клеша успешно разводят в ряде хозяйств.

ФИЛОГЕНИЯ ХЕЛИЦЕРОВЫХ

Хелицеровые, несмотря на существенные различия в морфологии между отрядами, имеют много общих признаков: две пары ротовых конечностей, мальпигиевые сосуды, коксальные железы, четыре пары ходильных ног и т. д. Считается, что паукообразные произошли от водных членистоногих, относившихся к ракоскорпионам, которые в процессе эволюции постепенно перешли к жизни на суше.

Наиболее древнюю группу хелицеровых представляют водные хелицеровые с жаберным дыханием: мечехвосты и ракоскорпионы, из которых сохранились лишь несколько видов мечехвостов. На суше, видимо, вышли представители ракоскорпионов. Выход на суше и формирование современных отрядов происходили независимо. Вместе с тем у всех отрядов паукообразных происходили сходные эволюционные процессы морфогенеза в связи с приспособлением к жизни на суше: формирование непроницаемой кутикулы, смена жаберного дыхания на трахейное и легочное, образование мальпигиевых сосудов вместо почек, специализация ротовых аппаратов и т. д.

В настоящее время ученые не располагают доказательствами единого происхождения отрядов паукообразных, и поэтому этот класс можно считать объединяющим несколько самостоятельных эволюционных линий развития сухопутных хелицеровых. Филогенетические отношения современных таксонов отражены на рис. 114.

Среди первичноводных хелицеровых выделялись группы крупных хищных бентосных и бенто-pelагических форм с мощным головогрудным панцирем (мечехвосты, ракоскорпионы). Они характеризовались жаберным дыханием, наружным оплодотворением и развитием с метаморфозом. Более мелкие формы выходили на суше, и у них формировалась система приспособлений к жизни в воздушной среде. Затем эко-



Рис. 114. Экологическая радиация хелицеровых:

1 — первичные водные предки хелицеровых; 2 — ископаемый водяной скорпион; 3 — сухопутные ночные хищники; 4 — норные хищники; 5 — пауки в паутинных трубках; 6 — фитобионты; 7 — дневные сухопутные хищники; 8 — мелкие геобионты; 9 — обитатели гниющих субстратов; 10 — эпифионты; 11 — геобионты (скважники); 12 — эктопаразиты; 13 — эндопаразиты; 14 — аэропланктон; 15 — амфибионты; 16 — форезия эпифионтов на насекомых; 17 — эндозообионты; 18 — кровососущие эктопаразиты; 19 — вторичные гидробионты

логическая радиация в освоении суши пошла в нескольких направлениях: освоение почвы (клещи), лесной подстилки и разлагающихся растений (паукообразные). Оторвавшись от почвы смогли клещи и пауки, заняв растительный ярус и перейдя вторично в воду. Клещи заняли все среды обитания, что привело к возникновению несвойственных для хелицеровых трофических групп: фитофаги, сапрофаги и паразиты.

ПОДТИП ТРАХЕЙНОДЫШАЩИЕ (*Tracheata*)

Трахейнодышащие, или трахейные, — сухопутные членистоногие животные, к которым относятся многоножки и насекомые. Органы дыхания у них представлены трахеями, хотя есть представители, дышащие через покровы тела. Среди трахейных встречаются и вторичноводные виды, которые сохранили трахеи.

Тело трахейных подразделяется на голову и многочлениковое туловище (многоножки) или на голову, грудь и сегментированное брюшко

(насекомые). Конечности одноветвистые. На голове одна пара усиков. Кутикула непроницаема, что обусловлено появлением водонепроницаемого слоя — эпикутикулы, состоящей из воскоподобных веществ. У некоторых видов сохранилась полупроницаемая кутикула.

Для экономии влаги у трахейных функционируют мальпигиевые сосуды, имеющие морфофункциональное сходство с мальпигиевыми сосудами у паукообразных. Но это пример конвергентного сходства, так как у трахейных мальпигиевые сосуды имеют эктодермальное происхождение (у паукообразных они энтодермального происхождения).

Подтип Трахейнодышащие подразделяется на два надкласса: Многоножки (*Myriapoda*) и Шестиногие (*Hexapoda*). Ранее эти надклассы рассматривались как классы, но сейчас выяснилось, что каждый из них объединяет группу родственных, но одновременно филогенетически различных классов, характеризующихся существенными различиями в организации.

НАДКЛАСС МНОГОНОЖКИ (*MYRIAPODA*)

К многоножкам относится более 15 тыс. видов трахейнодышащих членистоногих, тело которых состоит из головы и туловища, разделенного на множество сегментов. Почти все сегменты туловища несут по 1-2 пары ножек. Это самая древняя группа трахейных, которые произошли от древних примитивных групп ракообразных, перешедших к обитанию на суше. Многоножки были одними из первых сухопутных бес позвоночных: у них сохранились остатки кожно-мускульного мешка, гомономность сегментации туловищного отдела, производные целомодуктов. Многоножки ведут скрытный образ жизни, прячась под камнями, в щелях, под корой пней, под листьями и т. п., предпочитая влажные места обитания. Развитие у многоножек и бессмяжковых насекомых развитие происходит с изменениями, связанными с увеличением числа сегментов тела и члеников. Такое развитие получило название *анаморфоз*.

К надклассу *Myriapoda* относят четыре класса: класс Симфилии (*Sympyla*), класс Пауроподы (*Pauropoda*), класс Двупарногие, или Кивсяки (*Diplopoda*) и класс Губоногие (*Chilopoda*).

Представители класса Симфилии (*Sympyla*) обитают в почве под растительными остатками. Это мелкие многоножки (рис. 115), длина тела которых составляет несколько миллиметров, питающиеся растительными остатками. Насчитывают около 150 видов симифил. Глаза у них отсутствуют. Туловище состоит из 15—22 сегментов, но ходильных ног всегда 12 пар. Оплодотворение сперматофорное.

Многоножки класса Пауроподы (*Pauropoda*) еще мельче — длина тела до 1,5 мм. Обитают они в лесной подстилке. Известно около 350 видов. Усики ветвистые. Глаза отсутствуют. Туловище состоит из 10 сегментов и имеет 9 пар ног. Дышат через покровы тела. Кровеносная система отсутствует. Развитие с анаморфозом (личинки выходят из

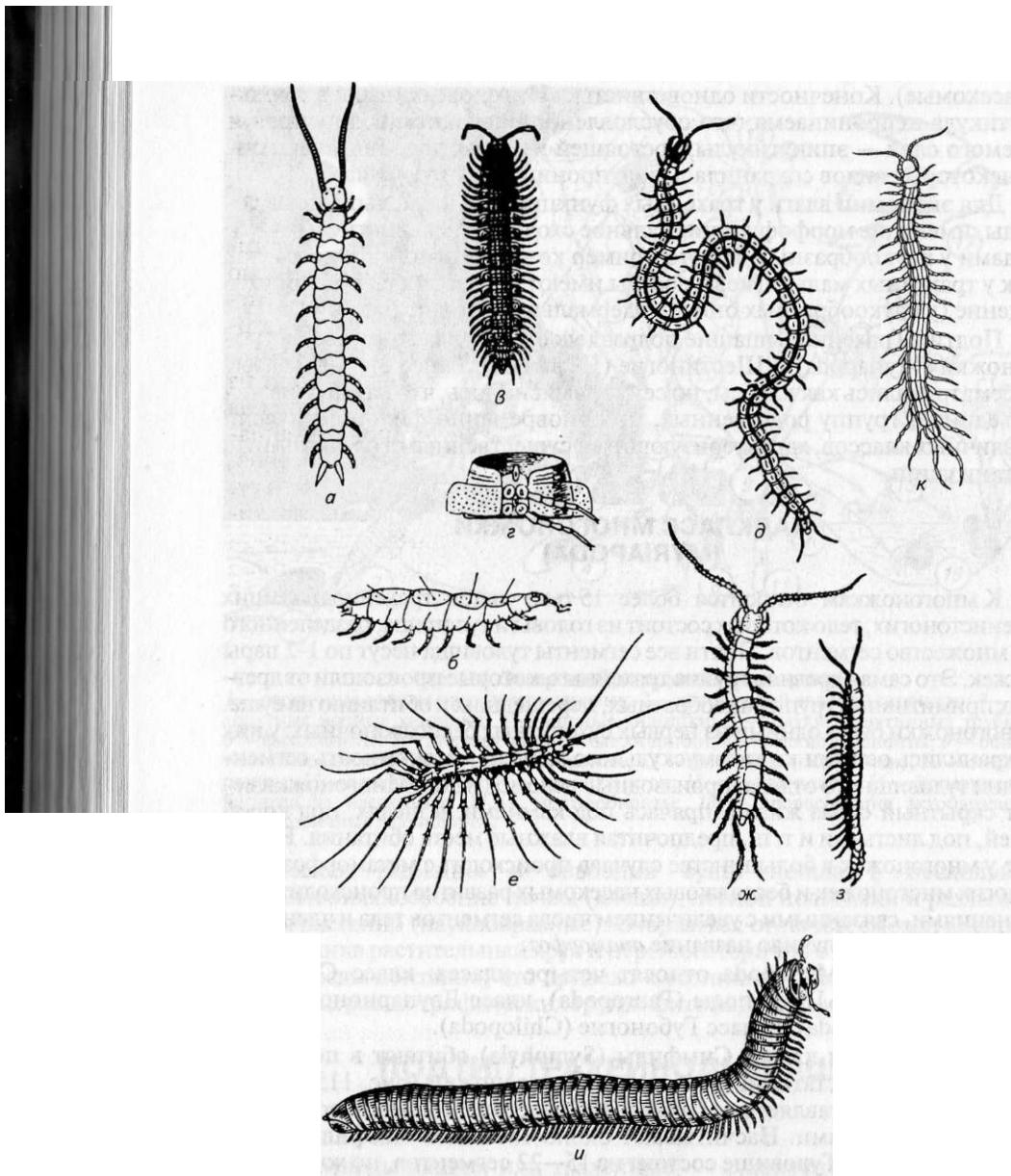


Рис. 115. Различные многоножки:
а — симфила *Scolopendrella immaculata*; **б** — пауропода *Decapauropus cuenoti*; **в** — *Polydesmus complanatus*; **г** — двойной сегмент многоножки *Pheacobius*; **д** — *Pachymerium ferrugineum*; **е** — мухоловка *Scutigera*; **ж** — костянка *Lithobius fojicatus*; **з** — *Cupifer amphieuris*; **и** — кивсяк *Julus memorensis*; **к** — сколопендра *Scolopendra inermipes*

яйца с тремя парами ног). Пауropоды питаются разжиженной в результате деятельности микроорганизмов растительной пищей, участвуя тем самым в почвообразовательном процессе.

КЛАСС ДВУПАРНОНОГИЕ, ИЛИ КИВСЯКИ (*Diplopoda*)

Представители этого класса получили название двупарногих, потому что каждый сегмент туловища несет по две пары ног. Это обусловлено попарным слиянием грудных сегментов и образованием сдвоенных сегментов. Известно около 12 тыс. видов диплопод, большинство из которых питаются растительными остатками, реже встречаются фитофаги.

Длина тела у кивсяков может достигать 10—20 см. Тело состоит из головы, свободного шейного сегмента и сегментированного туловища (см. рис. 115). На голове у них расположена пара усиков; ротовой аппарат состоит из пары мандибул и пластинки, представляющей собой, по-видимому, максиллы. Сверху ротовые части прикрыты верхней губой. По бокам головы расположены глаза, но некоторые формы глаз лишены. Число ног может достигать более 130 пар. На каждом сдвоенном сегменте имеются паучьи железы, секрет которых отпугивает врагов.

Нервная система представлена головным мозгом и удлиненной брюшной нервной цепочкой, которая в каждом сдвоенном сегменте имеет по два парных ганглия. Органы чувств развиты слабо.

Пищеварительная система имеет три пары слюнных желез мезодермального происхождения. Задняя кишечка длинная и состоит из нескольких отделов, что связано с бродильными процессами, происходящими в кишечнике при питании растительными остатками.

Кровеносная система незамкнутая. Сердце длинное и многокамерное. Число камер соответствует числу сегментов. Каждая камера сердца открывается двумя парами отверстий — ости. Кровь из отходящих от сердца артерий попадает в лакуны миксоцеля.

Органы дыхания. На каждом сдвоенном сегменте находится по две пары стигм. Трахейная система примитивна: от каждой стигмы отходит изолированный от других пучок трахей.

Выделительная система в виде мальпигиевых сосудов эктодермального происхождения. Функцию почек выполняет жировое тело.

Органы размножения. Кивсяки раздельнополы. При спаривании самец переносит сперматофоры в половые пути самки. Развитие происходит с анаморфозом: личинка выходит с неполным числом сегментов [туловища и с тремя парами ног на первых трех туловищных сегментах]. Они напоминают личинок насекомых. По мере роста личинки линяют и увеличивают число сегментов и ног.

Из современных отрядов наиболее часто встречаются представители основного отряда кивсяков (*Juliformia*). В нашей стране обитает около 150 видов кивсяков. В лесной зоне обычен серый кивсяк (*Rossiulus kessleri*) и песчаный кивсяк (*Schizophyllum sabulosum*), питающиеся растительными остатками и участвующие в почвообразовательных процессах. В лесных Ююсах степной зоны на 1 м² почвы насчитывают до 200 особей кивсяков.

КЛАСС ГУБОНОГИЕ (Chilopoda)

В отличие от прочих многоножек губоногие являются активными хищниками. У основания ногочелюстей у них имеются ядовитые железы. Губоногих насчитывают около 2,8 тыс. видов, длина тела некоторых достигает 20 см. Тело уплощено, сегментация туловища относительно гомономна. На голове длинные усики, скопления простых глаз и три пары челюстей: мандибулы и две пары максилл. Туловищные сегменты имеют сходное строение и несут по паре ходильных ног. Могут передвигаться вперед и назад.

Губоногим свойственно внекишечное пищеварение: в ранку на теле жертвы они вводят слюну и высасывают полупереваренную пищу. Трахейная система в значительной степени усложнена. Размножение сперматофорное. Самец откладывает сперматофор на специально сотканный им паутину, самка сперматофор захватывает. Развитие может быть прямым (у сколопендр), но чаще наблюдается аноморфоз.

Наиболее часто встречаются в средней полосе и на юге представители отрядов костянок (*Lithobiomorpha*) и геофилов (*Geophilomorpha*), обитающих под укрытиями и способных проникать глубоко в почву. У геофилов длинное (3—4 см) тело с 31—177 парами ног. Питаются мелкими беспозвоночными, могут нападать на дождевых червей, высасывая из них кровь. Костянки короче, и число пар ног у них равно 15. Это ночные хищники, охотящиеся на насекомых.

Наиболее крупные представители входят в отряд сколопендр (*Scolopendromorpha*). Это теплолюбивые многоножки, нападающие даже на мелких позвоночных животных. Укус сколопендр для их жертв смертелен, но не опасен для человека. На юге нашей страны обычна кольчатая сколопендра (*Scolopendra eugulata*) — ночной хищник, размножающийся партеногенетически.

НАДКЛАСС ШЕСТИНОГИЕ (Hexapoda)

Шестиногие относятся к подтипу трахейных, а следовательно, характеризуются трахейным дыханием, наличием на голове, состоящей из акрона и четырех сегментов, пары антенн и трех пар челюстей. Туловище состоит из трех сегментов, несущих три пары ходильных ног. Брюшко лишено развитых конечностей. Надкласс шестиногих представлен двумя классами: Скрыточелюстные насекомые (*Insecta* — *Entognatha*) и Открыточелюстные насекомые (*Insecta* — *Ectognatha*).

КЛАСС НАСЕКОМЫЕ СКРЫТОЧЕЛЮСТНЫЕ (Insecta — Entognatha)

К этому классу относятся примитивные бескрылые шестиногие, у которых грудной отдел слабо обособлен от брюшного. В брюшке 10—11 сегментов, ауногохвостоких. Крыльев нет. Скрыточелюстными они названы потому, что боковые стенки их ротовой полости срос-

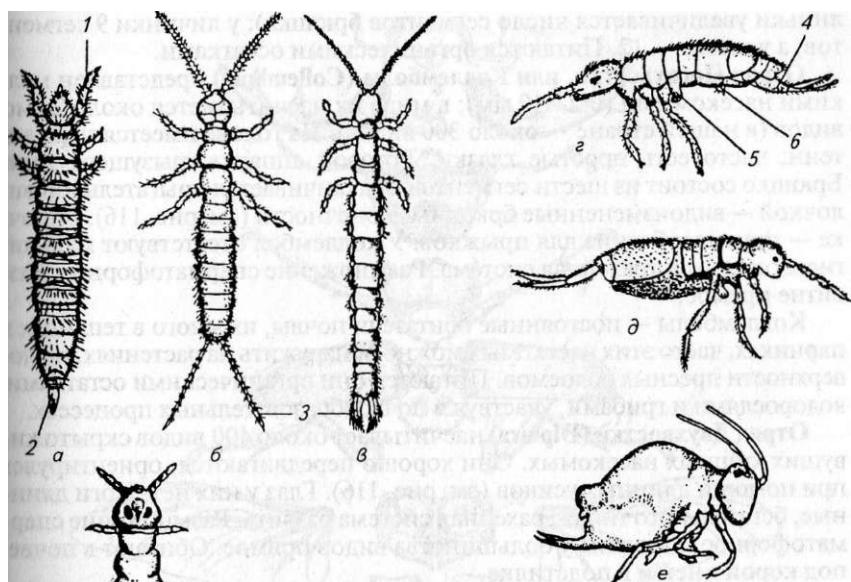


Рис. 116. Скрыточелостные насекомые:

а — протура *Acerentomon*; б — двухвостка *Campodea plusiochela*; в — двухвостка-япикс *Jyruх*; г, д, е, же — ногохвостки *Collembola*; 1 — передняя нога; 2 — задний конец тела; 3 — церки; 4 — вилка; 5 — трубка; 6 — придаток сегмента

лись с нижней губой, образуя головную капсулу. В этой капсule функционируют мандибулы и максиллы, у которых наружу выдаются только их концы. У большинства на голове имеется по паре усиков, могут быть простые глазки; дышат с помощью трахей или через покровы тела. Это мелкие насекомые, обитающие в основном в почве, подстилке и реже в травостое. Ротовой аппарат грызущего типа или сосущий. Развитие без метаморфоза; у бессяжковых наблюдается анаморфоз.

К скрыточелостным насекомым относят три отряда: Бессяжковые (IVotura), Ногохвостки (Collembola) и Двухвостки (Diplura).

Отряд Бессяжковые, или Протуры (Protura) включает мелких насекомых (0,5—2 мм), живущих в почве. Голова у них лишена усиков и глаз. Первая пара ходильных ног длиннее других и выполняет функции интенз (рис. 116). Большинство протур дышат через покровы тела. Размножение сперматофорное, развитие с анаморфозом (после каждой

линьки увеличивается число сегментов брюшка): у личинки 9 сегментов, а у имаго — 12. Питаются органическими остатками.

Отряд Ногохвостки, или Коллемболы (Collembola) представлен мелкими насекомыми (0,2—10 мм); в мире их насчитывается около 2 тыс. видов (в нашей стране — около 300 видов). На голове имеется пара антенн, часто есть простые глазки. Ротовой аппарат грызущего типа. Брюшко состоит из шести сегментов и заканчивается прыгательной вилочкой — видоизмененные брюшные конечности (см. рис. 116). Вилочка — приспособление для прыжков. У коллемболов отсутствуют мальпигиевые сосуды и трахейная система. Размножение сперматофорное. Развитие прямое.

Коллемболовы — постоянные обитатели почвы, их много в теплицах и парниках, часто этих насекомых можно обнаружить на растениях, на поверхности пресных водоемов. Питаются они органическими остатками, водорослями и грибами, участвуя в почвообразовательных процессах.

Отряд Двухвостки (Diplura) насчитывает около 400 видов скрыто живущих хищных насекомых. Они хорошо передвигаются, ориентируясь при помощи длинных усиков (см. рис. 116). Глаз у них нет. Ноги длинные, бегательного типа. Трахейная система развита. Размножение сперматофорное. Развитие у большинства видов прямое. Обитают в почве, под корой пней и в подстилке.

КЛАСС НАСЕКОМЫЕ ОТКРЫТОЧЕЛЮСТНЫЕ (Insecta — Ectognatha)

Общая характеристика. Это главный класс шестиногих, насчитывающий около миллиона видов. Их тело подразделяется на голову с усиками и тремя парами ротовых придатков, грудь с тремя парами ног и брюшко, которое состоит из 6—12 сегментов и лишено развитых конечностей. Дышат открыточелюстные с помощью трахей. У большинства представителей второй и третий сегменты груди несут по паре крыльев (рис. 117). Насекомые заселили все уголки нашей планеты. Роль их в природе и хозяйственной деятельности человека трудно переоценить.

Строение и жизненные отправления. Размеры тела, строение придатков отделов тела и их окраска чрезвычайно разнообразны. Ротовые части представлены парой верхних челюстей (мантибулами), парой нижних челюстей (максиллами) и нижней губой, которая образована в результате срастания второй пары нижних челюстей. Строение ротового аппарата у насекомых различных систематических групп существенно варьирует, что обусловлено разнообразием в способах питания. Можно выделить пять основных типов ротовых аппаратов: грызущий, грызуще-лижущий, колюще-сосущий, сосущий и лижущий (рис. 118).

Первичным типом ротового аппарата у насекомых можно считать *грызущий*, который свойствен представителям, питающимся твердой пищей (жуки, тараканы, прямокрылые). Верхние челюсти — жвалы — имеют вид хитиновых пластинок с зазубренными краями. Нижние челюсти — максиллы — состоят из члеников, несущих щупики и парные

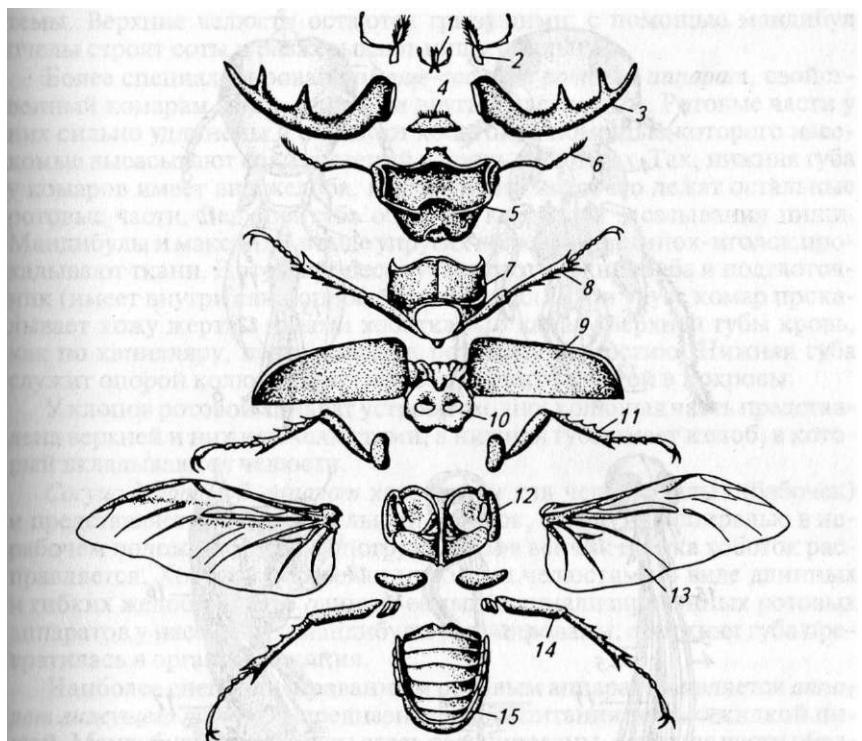


Рис. 117. Расчлененный жук-олень *Lucanus servus*:
1 — нижняя губа; 2 — нижняя челюсть; 3 — мандибула; 4 — верхняя губа; 5 — голова; 6 — антenna; 7 — переднегрудь; 8, I, 14 — грудные конечности; 9 — передняя пара крыльев (надкрылья); 10 — среднегрудь; 12 — заднегрудь; 13 — задние крылья; 15 — брюшко

попасти, усаженные щетинками. Жвалы и максиллы служат для отделения и размельчения пищи. По бокам нижней губы расположены щупики с осязательными волосками. На ротовых щупиках имеются ямки органов вкуса и осознания. Рот и его части снизу прикрыты нижней губой, а сверху он ограничен хитиновой складкой — верхней губой.

Остальные типы ротовых аппаратов насекомых, видимо, являются производным от аппарата грызущего типа, что было обусловлено переходом к питанию жидким пищей.

По сравнению с аппаратом грызущего типа наименее изменен ротовой аппарат *грызуще-лижущего* (лакающего) типа, который свойствен пчелам и шмелям. Для высыпывания нектара эти насекомые используют хоботок, состоящий из вытянутых максилл и нижней губы, а для изъятия нектара — язычок, который образован двумя внутренними выростами нижней губы. К язычку прилегают нижнегубные щупики, всасывание нектара в хоботок происходит по типу капиллярной сис-

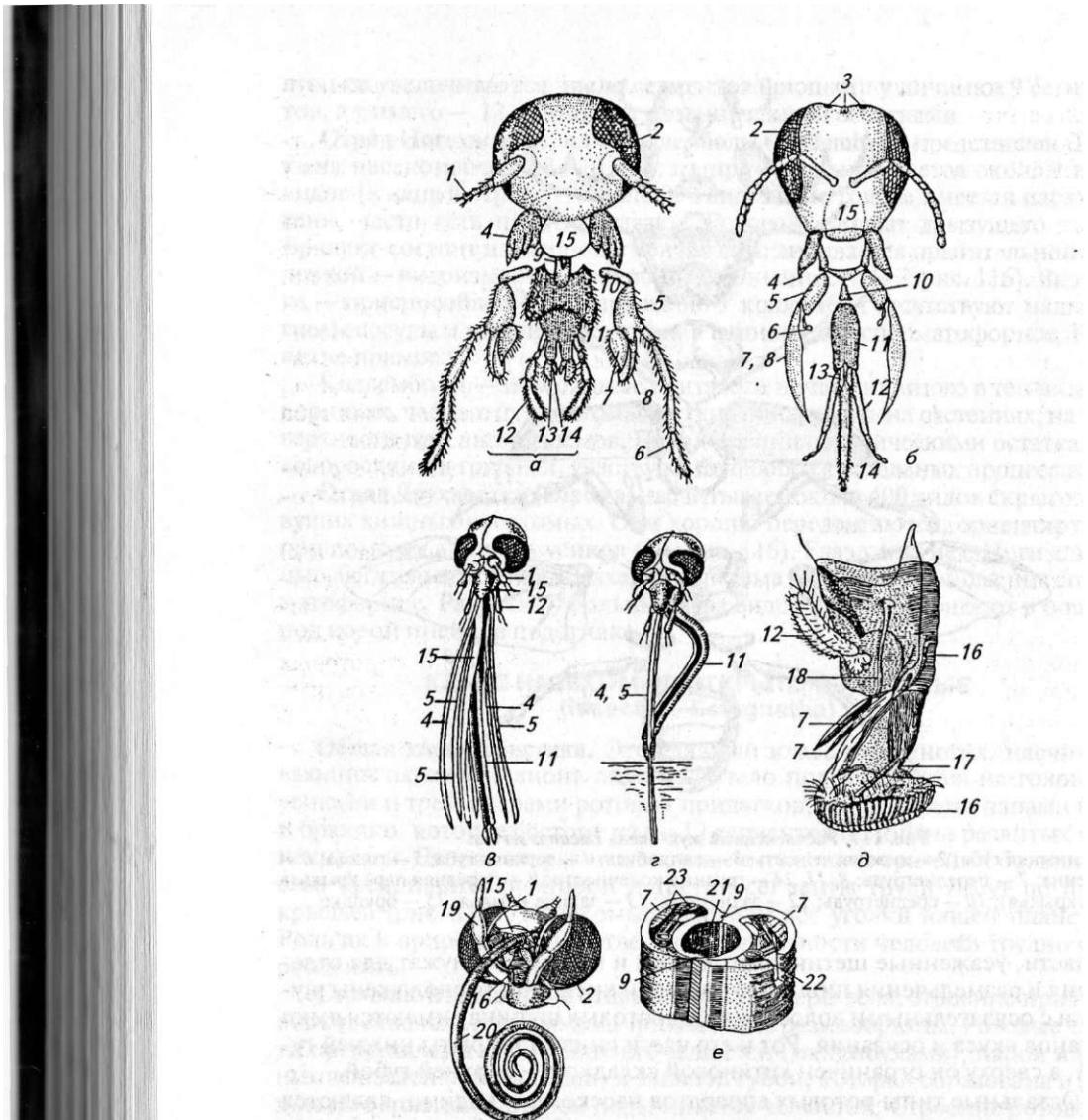


Рис. 118. Ротовые аппараты насекомых:
 а — грызущий у таракана; б — грызуще-лижущий у пчелы; в, г — колюще-сосущий у комара; д — лижущий у мухи; е — сосущий у бабочки (слева — голова с хоботком; справа — участок хоботка на поперечном разрезе); 1 — антенны; 2 — глаза; 3 — простые глазки; 4 — мандибулы; 5 — стволик максилл; 6 — челюстной шупик; 7, 8 — нижние челюсти, жевательные лопасти; 9 — сочленение; 10 — подбородок; 11 — подбородок нижней губы; 12 — губной шупик; 13, 14 — язычок; 15 — верхняя губа; 16 — нижняя губа; 17 — сочленение с лопастями; 18 — слюнные протоки; 19 — нижнегубной шупик; 20 — хоботок — нижние челюсти; 21 — полость хоботка; 22 — трахея; 23 — мускулатура хоботка

темы. Верхние челюсти остаются грызущими: с помощью мандибул пчелы строят соты и разжевывают пищу (пыльцу).

Более специализирован *колоюще-сосущий ротовой аппарат*, свойственный комарам, тлям, клопам и другим насекомым. Ротовые части у них сильно удлинены и образуют хоботок, с помощью которого насекомые высасывают соки растений и кровь животных. Так, нижняя губа у комаров имеет вид желоба, в углублении которого лежат остальные ротовые части. Верхняя губа образует канал для засасывания пищи. Мандибулы и максиллы в виде упругих колюющих щетинок-иголок прокалывают ткани. В этом процессе участвуют верхняя губа и подглоточник (имеет внутри слюнапроводящий канал). При укусе комар прокалывает кожу жертвы иглами хоботка. По каналу верхней губы кровь, как по капилляру, поднимается к ротовому отверстию. Нижняя губа служит опорой колющей части хоботка, погруженной в покровы.

У клопов ротовой аппарат устроен сходно: колющая часть представлена верхней и нижней челюстями, а нижняя губа имеет желоб, в который вкладываются челюсти.

Сосущий ротовой аппарат характерен для чешуекрылых (бабочек) и представляет собой сосательный хоботок, свернутый спиралью в не-рабочем положении. Перед погружением в венчик цветка хоботок расправляется. Хоботок образован нижними челюстями в виде длинных и гибких желобков. Это один из самых специализированных ротовых аппаратов у насекомых: мандибулы редуцированы, а нижняя губа превратилась в органы обоняния.

Наиболее специализированным ротовым аппаратом является *аппарат лижущего типа*. Он предназначен для питания только жидкой пищей. Мандибулы и максиллы здесь редуцированы, а прочие части образуют хоботок, снабженный на конце губкоподобными лопастями. Мухи [югружают хоботок в жидкую пищу, которая поступает в пищевой канал по тонким капиллярным трубочкам лопастей на конце хоботка. У хищных и кровососущих мух имеются также режущие челюсти.

У некоторых насекомых ротовой аппарат редуцирован (поденки). Питаются поденки только на личиночной стадии.

Перечисленные типы ротовых аппаратов свидетельствуют о широкой экологической радиации насекомых. От ротового аппарата грызущего типа прослеживается переход к другим более специализированным типам. При этом у насекомых, обладающих специализированными ротовыми аппаратами (бабочки, мухи), личинки имеют ротовые части грызущего типа.

Грудь насекомых состоит из передне-, средне- и заднегруди. К каждому сегменту груди прикрепляются по паре ног — всего шесть ножек. Каждый сегмент грудного отдела построен из склеритов, чередующихся с участками мембранных; такая конструкция обеспечивает подвижность и прочность наружного скелета. Помимо этого в грудном отделе имеются внутренние втячивания покровов — эндоскелет, к которому прикрепляются мышцы ног и крыльев. Различают спинные склериты (тергиты), брюшные (стерниты) и боковые (плевриты).

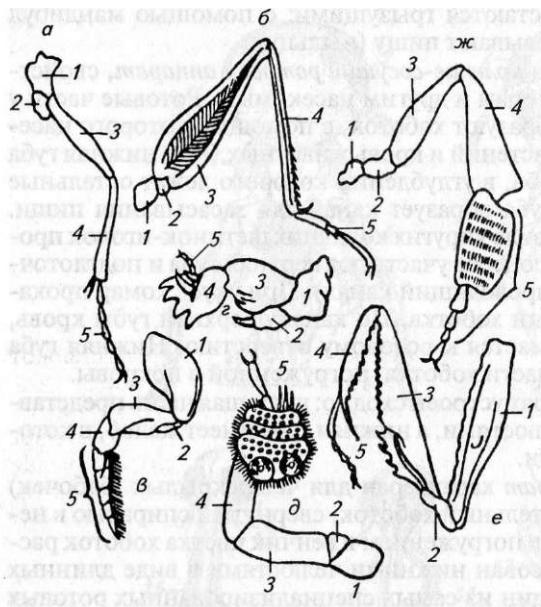
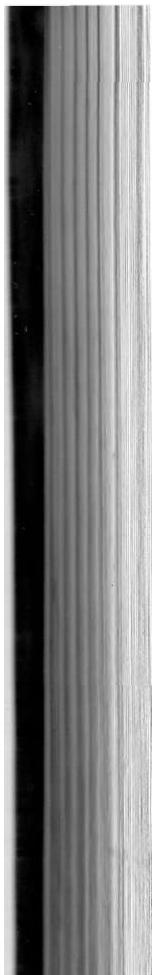


Рис. 119. Конечности насекомых:
а — бегательная (жука);
б — прыгательная (саранчи);
в — плавательная (плавунца);
г — роющая (медведки);
д — присасывательная (плавунца);
е — хватательная (богомола);
ж — собирает льна (медоносной пчелы); 1 — тазик;
2 — вертлуг; 3 — бедро;
4 — голень; 5 — лапка

Ноги насекомых чрезвычайно разнообразны по своему строению, что зависит от образа жизни и способа передвижения (рис. 119). Различают несколько типов ног, которые соответствуют характеру движения насекомых: бегательные, прыгательные, собирательные, копательные, плавательные, присасывательные и хватательные. В зависимости от выполняемой функции каждый тип конечностей характеризуется особенностями морфологии. Многочлениковые конечности позволяют насекомым успешно передвигаться в различных условиях обитания. Ходильные и бегательные ноги были исходным типом ног у насекомых. От них происходил переход к специализированным типам.

Ноги состоят из членников, соединенных друг с другом с помощью суставов. Основной членник — тазик, за которым расположен небольших размеров вертлуг, облегчающий движение конечности. За вертлугом идут два удлиненных членника — бедро и голень, в которых лежат хорошо развитые мышцы. Нога заканчивается многочлениковой лапкой с коготками на конце. Образование лапки, состоящей из трех-четырех членников, является достижением эволюционного развития. Функционально лапка соответствует стопе сухопутных позвоночных животных. Особые приспособления (присоски, волоски, клейкое вещество) позволяют насекомым передвигаться спиной вниз даже по гладкой поверхности.

Основная масса насекомых на средне- и заднегруди имеют по паре крыльев. Комары, мухи и близкие к ним формы сохранили только переднюю пару крыльев, задняя пара крыльев у них редуцирована и превратилась в своеобразные органы — жужжалыца. У некоторых насекомых крыльев вообще нет. У примитивных форм отсутствие крыльев яв-

ляется первичным признаком, тогда как у высокоразвитых форм это вторичный признак, возникший в процессе эволюции. Редукция крыльев свойственна паразитическим насекомым — вшам, пухоедам, Глохам и другим, а также некоторым почвенным и пещерным насекомым, обитателям гнезд термитов и муравьев. У жуков передние крылья называются надкрыльями. Они толстые и жесткие и служат не только для полета, но и для защиты тонких перепончатых крыльев.

У высших насекомых из отрядов перепончатокрылых и чешуекрылых передние и задние крылья скрепляются между собой в рабочем положении специальными приспособлениями и образуют как бы два функционирующих крыла.

Покровы насекомых образованы однослойным эпителием — гиподермой и выделяемой ею хитиновой кутикулой. Крылья представляют собой тонкие и плоские складки покровов сегментов груди, образованные двумя слоями кутикулы и подстилающей ее гиподермой. Иначе говоря, они представляют собой складки стенки тела и состоят из двух слоев, покрытых кутикулой, и узкой полости тела между ними. Крылья имеют жилки — трубчатые утолщения хитинового покрова крыла (в виде каналов), что придает ему прочность. По жилкам в крыло проходят трахеи, нервы, лакуны миксоцеля.

Крылья закладываются в виде мягких складок на стадии куколки. При выходе насекомого из куколки в крылья по жилкам нагнетается гемолимфа, а по трахеям — воздух. Крылья расправляются и через некоторое время их покровы затвердевают. Крыло приводится в движение специальной крыловой мускулатурой. По характеру жилкования крыла высоких насекомых относят к той или иной систематической группе (рис. 120).

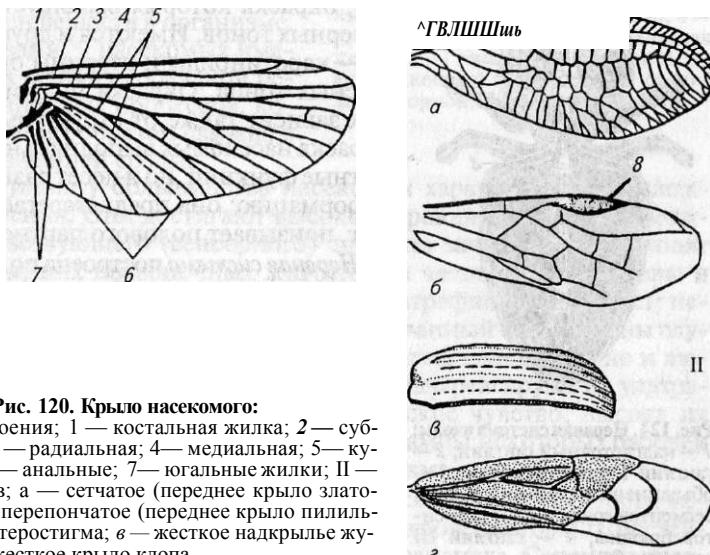


Рис. 120. Крыло насекомого:
I - схема строения; 1 — костальная жилка; 2 — субкостальная; 3 — радиальная; 4 — медиальная; 5 — кутикулярные; 6 — анальные; 7 — югальные жилки; II — виды крыльев; а — сетчатое (переднее крыло златоглазки); б — перепончатое (переднее крыло пилильщика); в — жесткое надкрылье жука; г — полужесткое крыло клопа

Полет насекомых чрезвычайно разнообразен: машущий, парящий, порхающий и т. п. Работа крыльев может быть весьма интенсивной: при полете пчела совершаet более 200 взмахов в 1 с, а комары-звонцы — до 1000. Пчела летит со скоростью 3 м/с, а бражники — 15 м/с.

Брюшко насекомых состоит из 6—10 сегментов. У взрослых насекомых конечностей на брюшке нет, но у некоторых представителей в личиночных стадиях на сегментах брюшка расположено несколько пар нечленистых ложноножек. У одних видов на конце брюшка самки находится яйцеклад, с помощью которого они откладывают яйца на субстрат или в тело будущего хозяина личинки (паразитические наездники). У других на конце брюшка расположено жало, открывающееся протоком ядовитой железы. Имеются и другие придатки брюшка (грифельки, церки).

Покровы представлены однослойной гиподермой, выделяющей снаружи кутикулу. Кутикула состоит из трех слоев: эпи-, экзо- и эндокутикула. Последние два слоя значительно толще эпикутикулы и в основном построены из хитина, белков и других веществ. Кутикула с внутренней стороны имеет выросты, к которым крепятся мускулатура и внутренние органы. Снаружи насекомые могут быть покрыты чешуйками, волосками, щетинками, шипами и т. п. Все это — производные покровов.

В гиподерме насекомых находится много разнообразных желез: восковые (пчелы), паучие (клопы), ядовитые (гусеницы). Слюнные и паутинные железы также имеют кожное происхождение.

В гиподерме и кутикуле насекомых могут содержаться различные пигменты (меланины), окраска которых варьирует от желтых до черных тонов. Имеются и другие пигменты — каротиноиды и птерины оранжевых и красных тонов. Окраска покровов насекомых зависит также от структуры кутикулы. Окраска насекомых выполняет не только защитные функции, но и несет разнообразную информацию: она предостерегает, обманывает, призывает полового партнера и т. п.

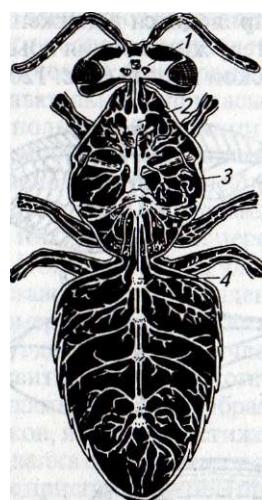


Рис. 121. Нервная система пчелы: нервирует усики, а тритоцеребрум — верхнюю губу. На протоцеребруме развиты ассоциативные центры — грибовидные тела, к которым подходит нервы от глаз. Особенно хорошо развиты головной мозг и грибовидные тела

у насекомых со сложным поведением (пчелы, муравьи).

Головной мозг связан окологлоточными нервными тяжами с подглоточным ганглием, иннервирующим ротовой аппарат и передний отдел кишечника. Брюшная нервная цепочка иннервирует грудь и брюшко. В ее состав входит три грудных и до восьми—десяти брюшных пар ганглиев. У некоторых насекомых число брюшных пар ганглиев уменьшается, а у многих мух грудные и брюшные ганглии цепочки сливаются в один грудной узел.

С центральной нервной системой связаны симпатические нервы, иннервирующие внутренние органы. Функции эндокринных желез у насекомых выполняют нейросекреторные клетки мозга, кардиальные, прилежащие тела и переднегрудные железы (рис. 122). Нейросекреты поступают в гемолимфу и управляют деятельностью эндокринных желез, вырабатывающих гормоны, которые регулируют рост, линьку, размножение и обмен веществ в организме. Иными словами, у насекомых имеется система нейрогуморальной регуляции жизненных процессов, которая теснейшим образом связана с факторами внешней среды.

Органы чувств у большинства насекомых характеризуются сложностью строения. Они достигают высокого уровня развития: возможности их чувствующего (сенсорного) аппарата зачастую превосходят таковые у высших позвоночных животных и человека. Так, пчелы и шмели воспринимают поляризованный и ультрафиолетовый свет; некоторые мухи также воспринимают поляризованный свет. Органы слуха насекомых воспринимают не только звуковые колебания, но и любые колебания среды. Доказано, что насекомые реагируют на ультразвук, у них есть сейсмическое и магнитическое чувство, высока их чувствительность к химическим веществам: самцы некоторых бабочек по запаху находят самок на расстоянии до 11 км.

Органы чувств у насекомых подразделяются на фоторецепторы, механорецепторы, хеморецепторы, термо- и гигиорецепторы. Простейшим элементом всех органов чувств является сенсилла. Сенсилла пред-

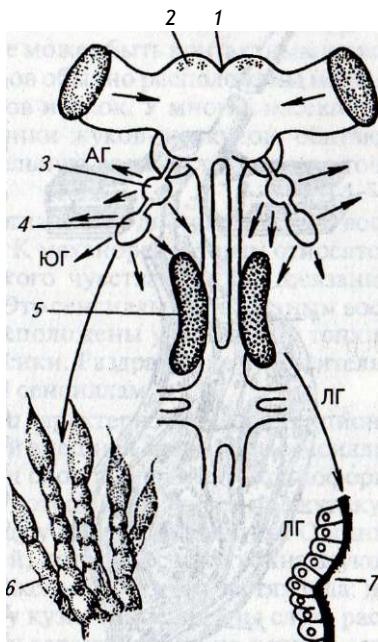
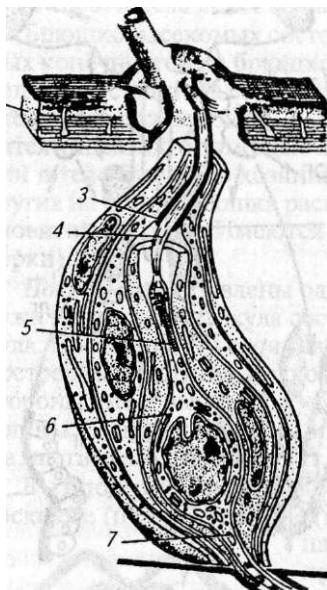


Рис. 122. Нейро-гуморальная система насекомых:

1 — головной мозг; 2 — нейросекреторные клетки; 3 — кардиальное тело; 4 — прилежащее тело; 5 — проторакальная железа; 6 — гонада; 7 — линька; АГ — активационный гормон; ЮГ — ювенильный гормон; ЛГ — гормон линьки

Рис. 123. Строение механорецепторной сенсильы насекомых:



1 — чувствительный волосок; 2 — кутикула; 3 — кутикулярная трубочка; 4 — видоизмененный жгутик; 5 — периферический отросток чувствительной клетки; 6 — чувствительная клетка; 7 — центральный отросток чувствительной клетки

ставляет собой одну или несколько сенсорных клеток, которые имеют чувствительные периферические отростки, за канчивающиеся во внешней среде чувствительным волоском. Раздражения, воспринимаемые чувствительным волоском, передаются по видоизмененному жгутику и периферическому отростку в сенсорную клетку и оттуда по центральному отростку клетки направляются в центральную нервную систему (рис. 123). Сенсиллы насекомых разнообразны по строению и функциям.

По бокам головы расположены сложные глаза и простые глазки. Сложные глаза насчитывают до тысячи и более омматидиев. Каждый омматидий состоит из двух главных компонентов: оптического и сенсорного. Сенсорный компонент образован чувствительны

ми, или ретикулярными, клетками, окружеными пигментными клетками. У дневных насекомых пигментные клетки, переполненные пигментом, изолируют соседние омматидии. У насекомых, ведущих сумеречный образ жизни, в пигментных клетках мало пигмента, что способствует проникновению световых лучей в соседние омматидии. В дневное время у сумеречных насекомых пигмент в клетках перераспределяется, что создает оптическую изоляцию омматидиев. При сумеречном зрении нарушается эффект мозаичности зрения.

Для многих насекомых свойственно восприятие цвета (бабочки, пчелы и муравьи). Пчелы, например, различают четыре цвета. Первый цвет, который воспринимают пчелы, соответствует нашему восприятию красного, желтого и зеленого. Второй цвет — сине-зеленый, третий — сине-фиолетовый и четвертый — ультрафиолетовый.

Простые глазки имеются не у всех насекомых. У пчел и муравьев их три и они размещены на темени. Они лишены хрустального конуса и иннервируются от переднего отдела мозга, тогда как сложные глаза иннервируются от грибовидных тел. Простые глазки дополняют зрительную функцию сложных глаз, информируя о степени освещенности. Это определяет суточную активность насекомых, имеющих только простые глазки.

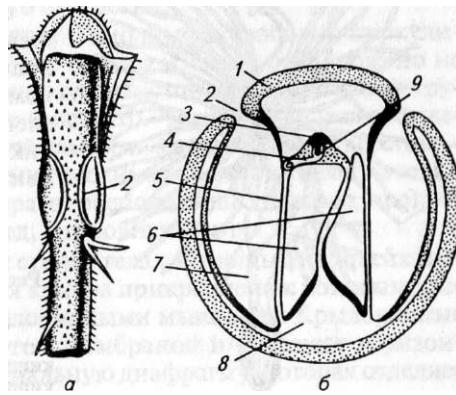
Термо- и гидрорецепторы воспринимают изменения температуры и влажности внешней среды. Восприятие может быть контактным и дистанционным. Сенсиллы этих рецепторов обычно расположены на усиках и щупиках; они имеют вид бугорков и ямок. У многих насекомых эти рецепторы хорошо развиты: личинки жуков-шелкунов, обитающие в почве, могут различать относительную влажность воздуха с точностью до 0,5 %.

Механорецепторы обладают контактным или дистанционным восприятием механических раздражений. К механорецепторам относятся органы осязания, слуха и сейсмического чувства. Органы осязания многочисленны на усиках насекомых. Это сенсиллы с контактным восприятием. Осязательные сенсиллы расположены у основания тонких подвижных волосков, покрывающих усики. Раздражение чувствительного волоска передается осязательным сенсиллам.

Органы слуха и сейсмического чувства характеризуются дистанционным восприятием различных колебаний внешней среды. Эти сенсиллы узкоспециализированы и называются сколпофорами. Сколпофоры входят в состав органов слуха, часть из которых находится под кутикулой или на мембранных, натянутых между участками кутикулы. Органы слуха хорошо развиты у представителей, которым свойственна звуковая сигнализация. Эти органы расположены на разных частях тела: на брюшке, усиках, ногах, крыльях. Так, у кузнецов органы слуха расположены на голенях передних ног, а у саранчовых — на первом сегменте брюшка (рис. 124). Самцы кузнецов и сверчков издают звуки с помощью передней пары крыльев. На левом крыле есть выступ в виде зубчатой жилки, а на правом — резонатор. При трении жилки (смычка) резонатор кузнецики издают звуки.

Хеморецепторы контактно или дистанционно воспринимают химические раздражения. К хеморецепторам относят органы обоняния и вкуса. Органы обоняния могут располагаться на разных частях тела, но особенно их много на усиках: на усиках пчелы насчитывают более 10 тыс. обонятельных сенсилл. По запаху насекомые отыскивают пищу, места для откладки яиц, половых партнеров. Многие насекомые выделяют особые веще-

Ил. 124. Тимпанальный орган (орган слуха) кузнецика:
п — передняя голень спереди, видны два отверстия тимпанального органа;
ft — поперечный разрез через голень в области тимпанального органа; 1 — кутикула голени; 2 — слуховая щель; 3 — шапочковая клетка; 4 — сколпофор; 5 — барабанная перепонка; ft — трахея; 7 — барабанная полость; V — полость ноги; 9 — сколопоидное гельце



ства — *аттрактанты*, к которым чрезвычайно чувствительны обонятельные сенсиллы: самцы бабочек находят самок на значительных расстояниях (3—9 км).

К контактным хеморецепторам относятся *органы вкуса*, которые расположены у мух и бабочек на лапках передних ног, у жуков — на челюстных и губных щупиках. При этом сенсиллы вкуса включают значительное число чувствительных клеток, специализированных на восприятие разного вкуса.

Мускулатура насекомых дифференцирована на скелетные мышцы (в основном поперечно-полосатые), приводящие в движение участки тела и его придатки, и мускулатуру внутренних органов. Интенсивность работы и относительная сила мышц насекомых обусловлены высокой скоростью обменных процессов в них, обеспеченных трахейным дыханием.

Пищеварительная система. Передний отдел кишечника эктодермального происхождения и представлен ротовой полостью, в которую открываются протоки одной-двух пар слюнных желез кожного происхождения, глоткой и пищеводом. В конце пищевода зачастую образуется расширение — зоб, где накапливается пища (рис. 125). У многих насекомых передний отдел кишечника заканчивается мускульным желудком, в котором пища перетирается. У растительноядных насекомых мускульный желудок имеет внутри хитиновые зубцы, у некоторых хищных форм желудок снабжен цедильным аппаратом из волосков. Первая пара слюнных желез вырабатывает слюну, богатую пищеварительными ферментами. Вторая пара слюнных желез может видоизменяться в паутинные или шелкоотделительные железы (гусеницы бабочек).

Строение переднего отдела кишечника может сильно варьировать в зависимости от типа питания насекомых. Например, у пчел имеется слепой вырост зоба — медовый зобик, в котором пчела накапливает нектар или мед.

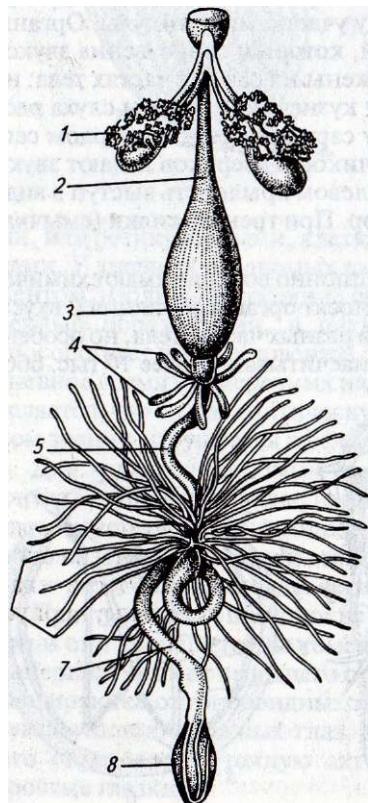
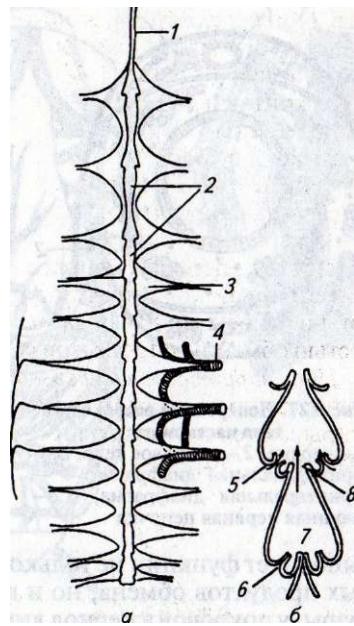


Рис. 125. Пищеварительная система черного тарантула:

1 — слюнные железы; 2 — пищевод; 3 — зоб; 4 — пищеварительные придатки; 5 — средняя кишка; 6 — мальпигиевые сосуды; 7 — задняя кишка; 8 — прямая кишка

Рис. 126. Сердце таракана (а) и схема работы сердца насекомого (б):

1 — аорта; 2 — камеры сердца; 3 — крыловидные мышцы; 4 — трахея; 5 — ости камеры сердца открыты; 6 — ости камеры сердца закрыты; 7 — клапан между камерами закрыт; 8 — клапан между камерами открыт



В энтодермальном среднем отделе кишечника происходит переваривание и всасывание питательных веществ, поступивших с пищей. Для увеличения поверхности кишечника в нем могут быть слепые выросты — пилорические придатки, выполняющие также функции пищеварительных желез. В зависимости от типа питания спектр пищеварительных ферментов у разных насекомых может меняться: у хищных форм преобладают протеолитические ферменты, а у фитофагов — амилазы. Клетчатка расщепляется под действием целлюлаз, вырабатываемых симбиотическими бактериями и простейшими. Например, у термитов, питающихся древесиной, таких симбионтов более десятка видов. Без этих симбионтов термиты гибнут.

На границе среднего и заднего отделов кишечника в него впадают от 2 до 150 мальпигиевых сосудов; эти сосуды энтодермального происхождения. Задняя кишка может быть дифференцирована на тонкую, толстую и прямую кишки. В стенках прямой кишки расположены ректальные железы, выполняющие функции всасывания воды из экскрементов перед выходом их из анального отверстия.

Кровеносная система незамкнутая. Гемолимфа, свободно циркулирующая в полости тела, омывает все органы. Сердце расположено на спинной стороне тела и имеет вид мускулистой многокамерной трубочки, замкнутой на одном конце (рис. 126). Число камер может колебаться от одной до восьми. В каждой камере имеется по паре отверстий — ости с клапанами, пропускающими гемолимфу только из полости тела в сердце. Между отдельными камерами расположены клапаны, пропускающие гемолимфу только вперед, в головную аорту.

Сердце подвешено к спинной стенке тела с помощью коротких мышечных тяжей. Каждая сердечная камера прикреплена к боковым частям тергитов специальными крыловидными мышцами. Крыловидные мышцы соединяются друг с другом мембранный и образуют горизонтальную перегородку — перикардиальную диафрагму, которая отделяет

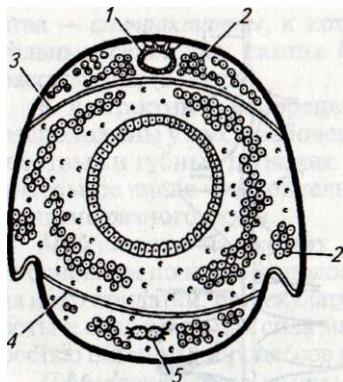


Рис. 127. Поперечный разрез через тело насекомого:

1 — сердце; 2 — жировое тело; 3 — перикардиальная диафрагма; 4 — периневральная диафрагма; 5 — брюшная нервная цепочка

выполняет функции не только переноса питательных веществ и конечных продуктов обмена, но и поддержания тургора тела, способствует разрыву покровов в период линьки и расправлению крыльев у молодых насекомых. У некоторых (жуки-нарывники, божьи коровки) кровь обладает ядовитыми свойствами: при опасности эти насекомые выбрасывают гемолимфу из сочленений сегментов и ног (произвольное кроповпускание, или аутогеморрагия).

Органы дыхания в основном представлены глубокими впячиваниями покровов — трахеями, в которые воздух поступает через специальные отверстия по бокам тела — дыхальца, или стигмы. Обычно число дыхалец колеблется от одной до десяти пар (рис. 128). Нагнетание и удаление воздуха из трахейной системы происходит за счет изменения объема брюшка с помощью специальных мышц. Внутри трахеи выстиланы тонкой хитиновой пленкой со спиральным утолщением. Это придает трахеям эластичность и упругость.

Трахеи опутывают и пронизывают все внутренние органы. Их концевые ветви заканчиваются звездчатой трахейной клеткой, от которой отходят тончайшие трубочки — трахеолы, проникающие даже внутрь клеток тканей. Иногда трахеи могут образовывать воздушные мешки, снижающие массу насекомых и улучшающие вентиляцию трахейной системы. У хорошо летающих насекомых (жуки, мухи, пчелы) воздушные мешки заполняют большую часть полости тела.

Личинки некоторых насекомых, живущих в воде (стрекозы, поденки), дышат трахейными жабрами, представляющими собой выросты брюшка с сетью трахей; у части личинок, живущих в воде, могут быть жабры, не имеющие трахей. В этом случае газообмен происходит через покровы жабр.

околосердечную полость (перикардиальную полость) от остальной полости тела (рис. 127). У некоторых насекомых на брюшной стороне тела образуется вторая (периневральная) диафрагма с мышцами. Под действием мышц обе диафрагмы могут совершать пульсирующие движения и тем самым способствовать циркуляции гемолимфы.

Гемолимфа состоит из плазмы и гемоцитов, или клеточных элементов, к числу которых относятся фагоциты, лейкоциты, амебоциты. Гемолимфа не выполняет функции переноса кислорода и диоксида углерода, так как у насекомых имеется трахейная система. Однако в гемолимфе личинок комаров *Chironomus* содержится красный дыхательный пигмент, близкий по своей структуре к гемоглобину. Гемолимфа

Рис. 128. Трахейная система черного таракана (а) и участок трахеи (б):

1 — трахейные стволы; 2 — стигмы; 3 — зоб; 4 — пилорические придатки; 5 — средняя кишка; 6 — задняя кишка; 7 — тенидии

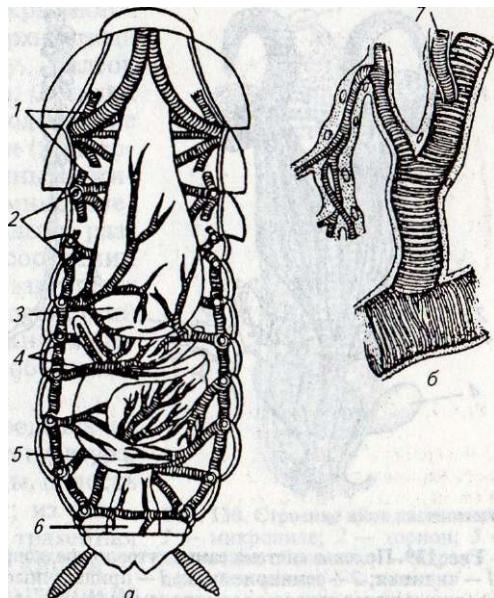
Органами выделения служат мальпигиевые сосуды, транспортирующие в заднюю кишку в основном кристаллы мочевой кислоты. У водных насекомых, а также у обитателей влажной среды, мальпигиевых сосудов много, так как они несут дополнительные функции осморегуляции. В засушливых условиях мальпигиевых сосудов у насекомых мало — всего одна-две пары; часто они слепыми концами прирастают к прямой кишке, • по способствует более полному всасыванию воды из них в гемолимфу.

Дополнительно функции органов выделения выполняют почки накопления: клетки жирового тела, перикардиальные клетки и кутикула. Жировое тело в основном выполняет функции накопления резервов питательных веществ (жиров, гликогена и белков) и обменной воды. За счет этого резерва происходит рост, осуществляется метаморфоз, насекомые могут длительное время переживать неблагоприятные условия. Одновременно в клетках жирового тела накапливаются соли мочевой кислоты и другие конечные продукты обмена (почка накопления).

Органы размножения. Насекомые раздельнополы, и часто у них хорошо выражен половой диморфизм (по окраске, дополнительным выростам, наличию крыльев и т. д.). Мужская половая система состоит из парных семенников, расположенных в брюшке, отходящих от них семяпроводов и непарного семязвергательного канала (рис. 129). Последний заканчивается совокупительным органом разнообразного строения. К мужской половой системе относятся придаточные железы, секреты которых разбавляют сперму и участвуют в образовании оболочек сперматофоров.

У самок яичники имеют вид двух пучков трубочек; их может быть от 1 до 100 пар, и открываются они в парные яйцеводы, впадающие в непарное влагалище, куда впадают протоки придаточных желез.

Самец вводит сперму в совокупительную сумку самки. У одних насекомых сперма из совокупительной сумки по специальному каналу



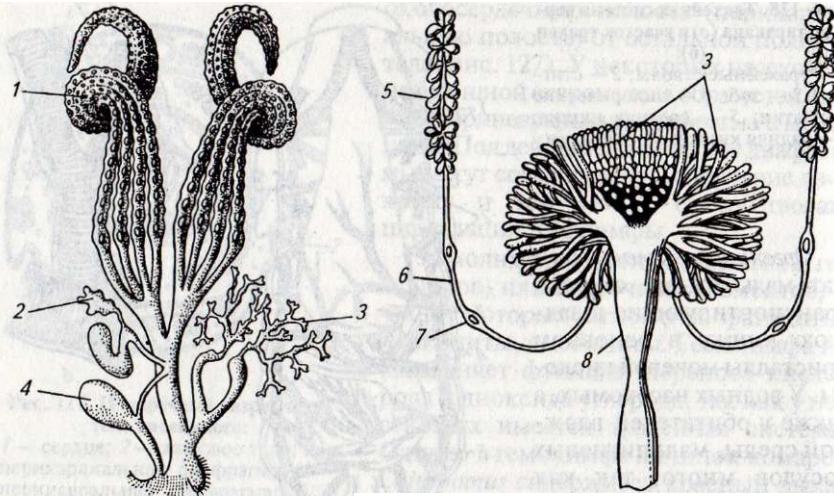


Рис. 129. Половая система самки тутового шелкопряда (а) и самца черного таракана (б):
1 — яичники; 2 — семяприемник; 3 — придаточные железы; 4 — совокупительная сумка, соединяющаяся каналом с семяприемником; 5 — семенник; 6 — семепровод; 7 — его раздвоение; 8 — семязвергательный канал

поступает во влагалище, где и происходит оплодотворение яиц. У других из совокупительной сумки сперма попадает в семяприемник, а затем порциями по протоку — во влагалище, где и происходит оплодотворение яиц. В семяприемнике спермии могут длительное время сохранять жизнеспособность. Например, матка пчел после спаривания с трутнями в течение 4—6 лет производит десятки тысяч оплодотвроренных яиц без повторного оплодотворения. Иногда у полового отверстия самок располагается яйцеклад (у прямокрылых он имеет саблевидную форму).

Развитие насекомых. Онтогенез (индивидуальное развитие) насекомых складывается из эмбрионального и постэмбрионального развития. Эмбриональный период начинается после оплодотворения яйцеклетки спермием. Яйца насекомых чрезвычайно разнообразны по форме, которая зачастую обусловлена средой, в которой они развиваются. Насекомые откладывают яйца группами (колорадский жук) и поодиноке. Кладки яиц могут быть открытыми (белянка) и закрытыми (саранчовые). Некоторые насекомые откладывают яйца в яйцевые капсулы — оотеки (тараканы).

Яйца насекомых покрыты оболочкой — хорионом, который защищает их от потери влаги. Оболочка яйца имеет небольшое отверстие — микропиле с пробочкой и каналец для проникновения спермия (рис. 130). Под хорионом располагается тонкая желточная оболочка, а под ней — плотный слой цитоплазмы с желтком. Дробление поверхностное: ядро многократно делится, и дочерние ядра с участками цитоплазмы мигриру-

ШШ*

30<0.01
0.01
0.001

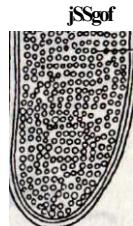


Рис. 130. Строение яйца насекомого:
1 — микропиле; 2 — хорион; 3 — желточная оболочка; 4 — ядро; 5 — полярное тельце; 6 — желток

ют к периферии яйца, где покрываются мембраной и образуют поверхностный слой клеток — бластодерму. Желток остается в центре зародыша. Складки бластодермы образуют зародышевые оболочки — серозу и амнион (конвергенция с высшими позвоночными животными). Образованная амниотическая полость позволяет зародышу развиваться во взвешенном состоянии внутри яйца, что надежно защищает его от механических воздействий (рис. 131), а жидкость в амниотической полости участвует в обменных процессах зародыша.

В процессе развития зародыша из мезодермы образуются мускулатура, сердце, жировое депо и гонады, полость тела становится смешанной; из эктодермы образуются нервная и трахейная системы, а из стенок задней кишки — мальпигиевы сосуды. Перед выходом из яйца личинка заглатывает жидкость из амниотической полости, что увеличивает тургор тела. Хорион личинка прорывает головой.

Постэмбриональное развитие. В период постэмбрионального развития увеличиваются масса и размеры насекомого, что сопровождается последовательными линьками и прохождением качественно различных фаз. Число линек колеблется от 3—4 до 30, но в среднем составляет 5—6. Промежуток между двумя последовательными линьками называется стадией, а состояние развития — возрастом.

Морфологические изменения в развитии от личинки до взрослого насекомого носят название *метаморфоз*. У всех насекомых (кроме низших бескрылых форм), достигших взрослого состояния, рост и линьки прекращаются.

Выделяют три основных типа постэмбрионального развития насекомых: прямое развитие без метаморфоза, развитие с неполным превращением и развитие с полным превращением.

Прямое развитие (аметаморфоз) происходит лишь у первичнобескрылых насекомых (двуухвостки, чешуйницы, коллемболы). При прямом развитии из яйца выходит личинка, внешне

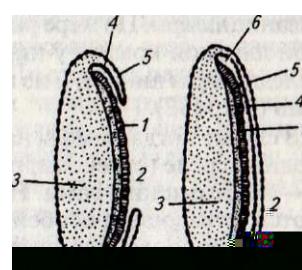


Рис. 131. Схема развития зародышевых оболочек у насекомых:
1 — эктодерма; 2 — впячивание, образующее общий зародок энтодермы и мезодермы; 3 — желток; 4 — амнион; 5 — серозная оболочка; 6 — амниотическая полость

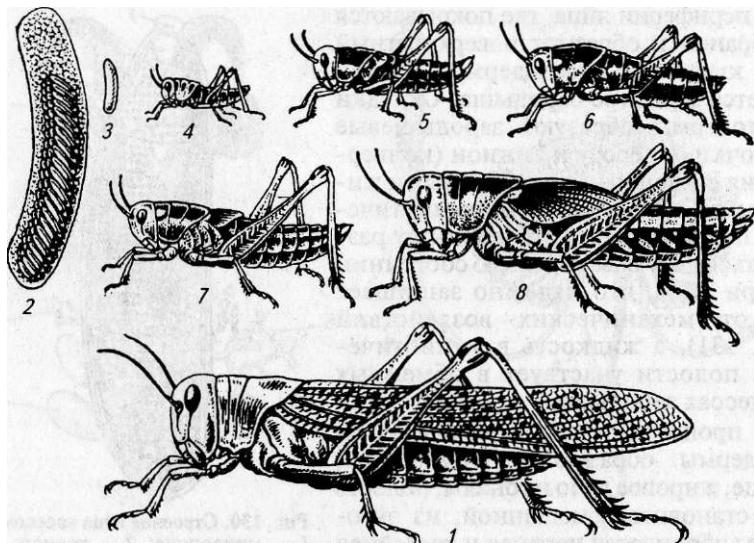


Рис. 132. Развитие насекомых (азиатская саранча) с неполным превращением:
1 — взрослая крылатая саранча (имаго); 2 — кубышка; 3 — яйцо; 4, 5, 6, 7, 8 — пять возрастов пешей саранчи (личинки разных возрастных стадий)

похожая на взрослое насекомое, но меньших размеров, с другими пропорциями тела и недоразвитыми половыми органами. У насекомых с прямым развитием линьки продолжаются и у взрослых особей.

Неполное превращение (гемиметаморфоз) часто называют развитием с постепенным метаморфозом. Такое развитие характерно для многих крылатых насекомых (ткараканы, прямокрылые, цикады, клопы и др.). При неполном метаморфозе из яйца выходит личинка, внешне похожая на взрослое насекомое, но меньшей величины, с зачаточными крыльями и недоразвитыми половыми органами. Такие личинки носят название *нимфы*. По мере развития нимфы несколько раз линяют, и с каждой линькой крылья у них увеличиваются. Нимфа старшего возраста последний раз линяет, и из нее выходит крылатое взрослое насекомое — *имаго* (рис. 132).

В случае когда нимфы насекомых с неполным превращением живут в водной среде (стрекозы, поденки, веснянки), они называются *наядами* — водными нимфами. Наяды по внешнему виду существенно отличаются от взрослых особей, в частности, у них имеются особые личиночные приспособления: трахейные жабры, видоизмененная нижняя губа, например маска у наяды стрекозы. У вторичнобескрылых (клопы, вши, пухоеды, власоеды) и у ведущих скрытный образ жизни в почве (прямокрылые) наблюдается слабо выраженный метаморфоз, когда взрослую нимфу трудно отличить от имаго.

Развитие с полным превращением (голометаморфоз) протекает по следующей схеме: яйцо — личинка — куколка — имаго (рис. 133). Такое

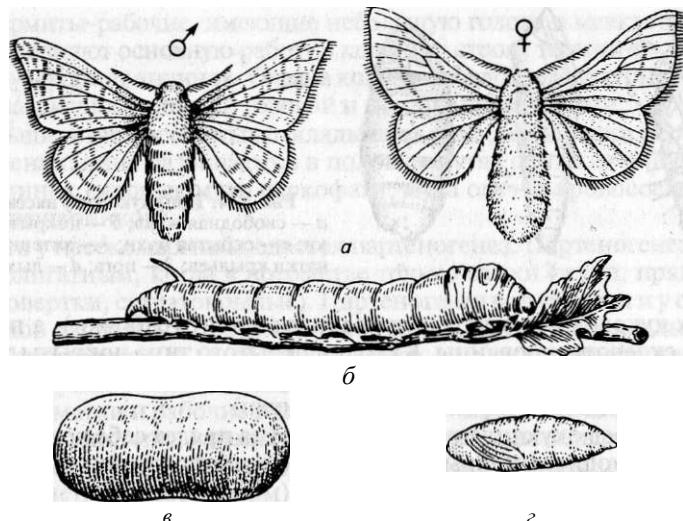


Рис. 133. Развитие насекомых (тутовый шелкопряд) с полным превращением:
а — бабочка; б — гусеница; в — кокон; г — куколка, извлеченная из кокона

развитие свойственно жукам, перепончатокрылым, двукрылым, ручейникам и сетчатокрылым. Личинки у насекомых с полным превращением внешне не схожи со взрослыми особями. Зачастую личинки не только отличаются от имаго типом питания, но и ведут иной образ жизни (личинки многих мух и майских жуков живут в почве). После нескольких линек личинки превращаются в куколку. В куколке происходит разрушение личиночных органов (гистолиз) и формирование органов и тканей взрослого организма (гистогенез). Из куколки выходит крылатое насекомое (имаго).

Личинки насекомых с полным превращением характеризуются грызущим типом ротового аппарата, отсутствием глаз и зачатков крыльев, наличием коротких усиков и ног. По степени развития конечностей личинки могут быть разделены на четыре группы. Безногие (аподные) личинки у двукрылых и у части жуков, бабочек. Личинки с зачатками конечностей (протоподные) характерны для пчел и ос. Эти личинки развиваются в сотах и поэтому малоподвижны. Хорошо развитые три пары ходильных ног (олигоподные) наиболее типичны для личинок крылатых насекомых (жуки, сетчатокрылые). Гусеницы (полиподные личинки) помимо хорошо развитых трех пар ходильных ног обладают еще несколькими парами ложножек на брюшке. Такие личинки свойственны бабочкам и пилильщикам.

Куколки могут быть свободные, покрытые и скрытые (рис. 134). У свободных куколок покровы тонкие и мягкие, у них хорошо обозначены и отделены от тела зачатки крыльев и конечностей (жуки). У по-

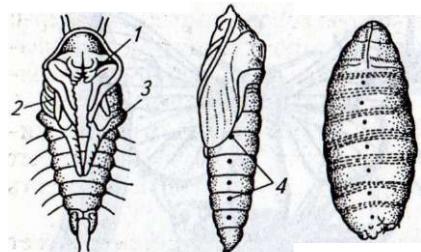


Рис. 134. Типы куколок у насекомых:
а — свободная жука; б — покрытая бабочки;
в — скрытая мухи; 1 — антenna; 2 — за-
чатки крыльев; 3 — нога; 4 — дыхальца

крытых куколок все зачатки прирастают к телу (бабочки), а покровы сильно склеротизированы. Куколки скрытого типа покрыты твердой сброшенной личиночной шкуркой, формирующей ложный кокон, в котором находится открытая куколка (мухи).

Часто перед оккулированием личинка сама прядет себе кокон, внутри которого находится покрытая куколка (шелкопряд) или куколка свободного типа (муравьи).

Большинству насекомых свойственно половое размножение. При этом у многих насекомых наблюдается хорошо выраженный половой диморфизм (самцы жуков-носорогов имеют на голове рог). У общественных насекомых наблюдается полиморфизм, т.е. особи одного и того же вида имеют различное строение в соответствии с их ролью в жизни колонии. Так, в колониях многих термитов матка — родоначальница семьи отличается огромным брюшком, которое не дает возможности ей передвигаться, и поэтому ее кормят особая группа термитов — рабочие (рис. 135). Каждый день матка откладывает несколько тысяч яиц. Жизнь самцов коротка: их роль сводится лишь к оплодотворению самки. Основу колонии состав-

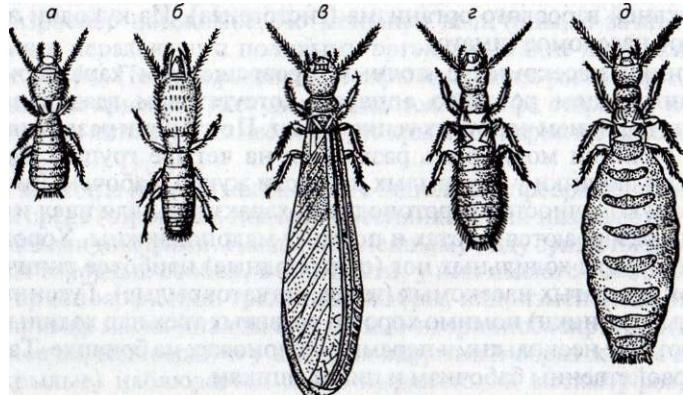


Рис. 135. Полиморфизм у термитов:
а — рабочий; б — солдат; в — крылатый самец; г — молодая самка, сбросившая крылья по-
сле оплодотворения; д — взрослая самка

ляют термиты-рабочие, имеющие небольшую голову и мелкие размеры. Они выполняют основную работу в колонии: строят термитник, добывают пищу и кормят личинок. Охрана колонии возложена на термитов-солдат, обладающих огромной головой и мощными челюстями.

Большинство насекомых откладывают яйца, но нередки случаи живорождения: яйца развиваются в половых путях самки, и она рождает уже личинок (мясные муhi саркофаги, муха овечья кровососка, часть жуков-пещерников).

Часто у насекомых наблюдается партеногенез. Партеногенез может быть облигатным, когда в потомстве только самки (жуки, прямокрылые, уховертки, сетчатокрылые). Партеногенез встречается и у обоеполых видов насекомых. В этом случае только часть яиц откладываются неоплодотворенными, а основная масса яиц оплодотворенные. Например, трутни у медоносных пчел развиваются из неоплодотворенных яиц, а матки и рабочие пчелы (самки) — из оплодотворенных. Подобный партеногенез характерен для муравьев, пилильщиков, термитов, части клопов и жуков. У тлей происходит смена поколений: партеногенетического (летом) и обоеполого (осенью). Иногда партеногенез может проявляться лишь при неблагоприятных условиях.

У некоторых насекомых наблюдается один из способов увеличения численности — бесполое размножение зародышей (полиэмбриония). Так, например, наездники откладывают яйца в личинок своих жертв — насекомых, где и происходит размножение эмбрионов.

В отличие от жизненного цикла, продолжительность которого у некоторых насекомых может превышать 10 лет, сезонные циклы характеризуют развитие вида в течение сезонов 1 года (от зимы до зимы). Например, продолжительность жизненного цикла майского жука может составлять до 5 лет (от яйца до взрослой стадии), поэтому в любой сезон года можно встретить личинок жуков разного возраста. Некоторые насекомые в течение года могут дать несколько поколений — поливольтинные виды (комнатная муха), но большинство насекомых дают одно поколение в год — моновольтинные виды. Важной особенностью сезонных циклов насекомых являются их активная жизнедеятельность и времененная задержка вследствие неблагоприятных условий среды.

Экология насекомых. Основная масса насекомых обитает на суше, заселяя разнообразные места обитания. Многие ведут подземный образ жизни, питаясь подземными частями растений, гниющими остатками и почвенными животными. Много насекомых заселяют лесную подстилку и травяной покров, где находят пищу и убежища. Обитатели кустарников и деревьев поедают листву, побеги, семена и плоды, грызут древесину или питаются соком растений. Многие насекомые длительное время находятся в полете, подымаясь с воздушными потоками на значительные высоты. Водные насекомые предпочитают стоячие водоемы и медленно текущие реки, тогда как в соленых водах их почти нет.

По времени активности насекомых делят на дневных, сумеречных и ночных.

Насекомые не способны поддерживать постоянную температуру тела, поэтому наиболее разнообразна и многочисленна фауна тропических стран. Энтомофауна зон умеренного и холодного климата значительно беднее, и почти все насекомые этих зон впадают в длительную зимнюю спячку. Даже в теплое время года климатические и погодные условия оказывают огромное влияние на жизнедеятельность насекомых. Так, бабочка совка-гамма на севере европейской части нашей страны за теплый период года дает лишь одну генерацию, в средней полосе — две, а на юге — до трех генераций.

Естественно, что погодные условия являются одним из главных факторов, влияющих на численность насекомых в разные годы, и это важно учитывать при организации борьбы с вредными насекомыми. Колебания численности насекомых обусловливаются также наличием запасов пищи, естественных врагов и миграциями животных.

В зависимости от численности насекомого область его распространения может быть разделена на зоны:

- зона постоянного обилия данного вида (деятельность вредителей постоянна и активна);
- зона постоянного обитания, но с периодическим обилием вида в благоприятные годы (деятельность вредителей по годам выражена в разной степени);
- зона с редкой встречаемостью вида (вредоносность может быть в виде вспышек численности в некоторые годы);
- зона, где данный вид насекомого не встречается, хотя может появляться в результате миграции в годы массового размножения в других зонах.

По образу жизни насекомые могут быть отнесены к следующим группам:

- свободноживущие виды, которые питаются растениями, животными, трупами, гниющими остатками и т. п.;
- паразиты растений:
 - а) эктопаразиты, обитающие на поверхности растений и питающиеся их соками (тли, червецы);
 - б) эндопаразиты, живущие в растениях и питающиеся их тканями (личинки короедов и дровосеков);
- паразиты животных:
 - а) эктопаразиты (вши, блохи, пухоеды);
 - б) эндопаразиты (личинки желудочных и кожных оводов, наездников, яйцеедов).

По характеру питания насекомые могут быть отнесены к следующим типам:

- фитофаги — растительноядные виды (соки растений, листья и побеги, семена, плоды, корни, корнеплоды и т. п.);
- зоофаги — питаются животной пищей;
- копрофаги — питаются навозом и экскрементами животных;
- некрофаги — питаются трупами;
- сапрофаги — питаются гниющими остатками растений;

- пантофаги — всеядные насекомые.

Насекомые характеризуются сложной нервной деятельностью. В основе их поведения лежат инстинкты — совокупность совершенных безусловных рефлексов (реакции организма на раздражители, поступающие из внешней среды; эти реакции сложились в течение длительного времени и стали врожденными — наследственными). Часто инстинкты отличаются сложностью, а многие из них до сих пор остаются загадкой для человека. Например, песчаная оса аммофила выкармливает своих личинок гусеницами бабочек, которых оса не убивает, а парализует. Парализованную гусеницу оса затаскивает в норку и откладывает в нее свое яйцо. Вышедшая личинка питается живыми тканями парализованной гусеницы.

Наиболее сложны инстинкты у общественных насекомых (муравьи, терmites, пчелы). В их колониях каждая группа насекомых (матка, трутни и рабочие пчелы) выполняет свои обязанности, которые могут меняться с возрастом насекомых.

Хозяйственное значение насекомых. Значение насекомых в природе и хозяйственной деятельности человека трудно переоценить. Шестиногие — самый многочисленный и разнообразный надкласс на планете. Особая роль принадлежит насекомым на суше, так как насекомые в основном ее обитатели. Они активно участвуют в круговоротах биогенных элементов, потому что среди насекомых есть и консументы первого порядка — фитофаги, и консументы второго и третьего порядков — хищники и паразиты, и редуценты — разрушители органических остатков. Хищные насекомые уничтожают множество вредителей сельскохозяйственных и дикорастущих растений (рис. 136). Насекомые-сaproфаги являются участниками почвообразовательного процесса.

Насекомые способны питаться практически любым органическим материалом, который встречается на нашей планете, но одновременно они являются пищей для представителей других групп животных: амфибий, рептилий, птиц и млекопитающих. Даже зерноядные птицы свое потомство выкармливают в основном насекомыми. Велика роль насекомых в опылении многих культурных и дикорастущих растений. Насекомые используются для получения продуктов питания (пчелы), сырья (тутовый шелкопряд), лекарственных препаратов, а также для борьбы с вредными представителями своего класса (энтомофаги) и растениями-сорняками (фитофаги). Гусениц тутового шелкопряда разводят в специальных хозяйствах, где из коконов этой бабочки производят естественные шелковые ткани. Широкое распространение получает разведение насекомых нектарофагов для опыления растений (пчелы-листорезы, шмели, пчелы-осмишидр.). Насекомых сапрофагов культивируют для переработки органических отходов, в том числе навоза, в целях получения ценного органического удобрения (комposta) и белкового корма для животных. Разводят насекомых и в кормовых целях (саранчовые, сверчковые, личинки мух и др.). Так, потомство только одной пары навозных мух за сезон может дать биомассу в сотни тонн. Одна самка комнатной мухи может произвести до 200 яиц и более.



Рис. 136. Полезные насекомые:
 а — муха- журчалка; б — ее хищная личинка; в — златоглазка; г — ее хищная личинка;
 д, е — жук-красотел и его личинка; ж — божья коровка; з — их личинки; и — жужелица;
 к — стафилинус

Насекомыми кормятрыб, певчих птиц и другихживотных, содержащихся в неволе. Разводят для содержания в неволе и самих насекомых: бабочек, жуков, палочников, богомолов, сверчков и др. Насекомые давно служат науке. На муке дрозофиле проводились классические опыты по генетике.

Большое число видов насекомых являются вредителями лесных насаждений, сельскохозяйственных растений, запасов сырья и продуктов, паразитами животных и человека. Одни вредители уничтожают

культурные растения и снижают их урожайность, другие наносят вред хлебным и фуражным запасам. Насекомые-паразиты снижают продуктивность сельскохозяйственных животных, вызывают тяжелые заболевания. Кровососущие насекомые могут быть переносчиками таких заболеваний, как бешенство, энцефалит, туляремия и др. Есть среди насекомых и промежуточные хозяева паразитических червей. С вредными видами насекомых человек ведет постоянную борьбу. Наряду с механическими, химическими и агротехническими методами борьбы все шире используются биологические методы, так как они безопасны для окружающей среды. Применяют в биологической борьбе энтомофагов, патогенные организмы, используют генетические методы. Насекомые, яйцееды и другие перепончатокрылые откладывают яйца в тело или яйца вредных насекомых, что приводит к их гибели.

СИСТЕМАТИЧЕСКИЙ ОБЗОР КЛАССА НАСЕКОМЫЕ (*Insecta*—*Ectognatha*)

В основу классификации насекомых положены особенности строения крыльев, ротового аппарата, тип постэмбрионального развития и другие признаки. Но приоритет отдается характеру жилкования крыльев, типу ротового аппарата, строению конечностей и половой системы.

Класс насекомых подразделяют на два подкласса: Первичнобескрылые (*Apterygota*) и Крылатые (*Pterygota*).

ПОДКЛАСС ПЕРВИЧНОБЕСКРЫЛЫЕ НАСЕКОМЫЕ (APTERYGOTA). Первичнобескрылые — это насекомые с примитивными чертами организации, у которых первично отсутствуют крылья (их предки тоже не имели крыльев). Ротовой аппарат грызущего типа, но слабоспециализированный. Их челюсти погружены в капсулу (скрыточелюстные). Глаза просые, реже сложные, у части видов отсутствуют. Развитие без превращений: личинки отличаются от имаго только размерами и пропорциями тела. Линяют и во взрослом состоянии. Населяют почву, сырье скрытые места, живут под камнями, под корой пней, во мху и т. п. К первичнобескрылым относятся два отряда, из которых наиболее распространены представители отряда Щетинохвостки (*Thysanura*). Это небольшие (8—20 мм) насекомые с тремя хвостовыми нитями. В почве многочисленны мелкие бесцветные прыгающие ногохвостки, участвующие в почвообразовательном процессе (рис. 137). В жилых помещениях встречается сахарная чешуйница, повреждающая бумагу и продукты.

ПОДКЛАСС КРЫЛАТЫЕ НАСЕКОМЫЕ (PTERYGOTA). У представителей крылатых насекомых имеются крылья, а у нелетающих — ихrudименты; однако это вторичная бескрыльость, обычно связанная с паразитическим образом жизни. Крылатые насекомые — наиболее высокоорганизованная и самая многочисленная группа насекомых. Челюсти у них расположены свободно (открыточелюстные). Ротовые аппараты разных типов. Глаза простые и сложные. Развитие происходит с полным или неполным метаморфозом. Представители важнейших отрядов крылатых насекомых показаны на рис. 138.

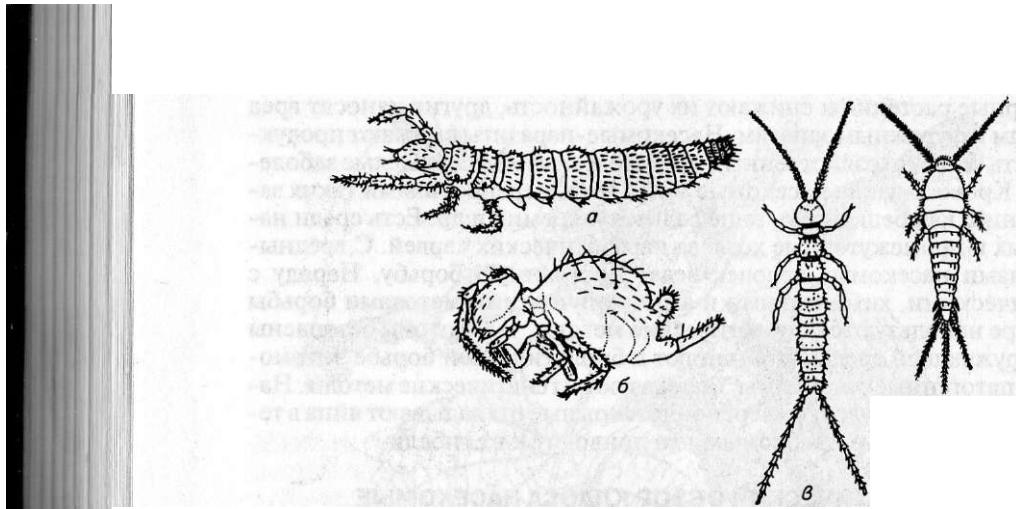


Рис. 137. Первично бескрылые насекомые:
а — бессяжковые; б — ногохвостки; в — двухвостки (чешуйница)

Среди крылатых насекомых выделяют два инфракласса: Древнекрылые (*Palaeoptera*) и Новокрылые (*Neoptera*).

Инфракласс Древнекрылые включает наиболее древних представителей крылатых насекомых: отряд Стрекозы и отряд Поденки. У этих насекомых крылья не могут складываться на спине и имеют примитивное сетчатое жилкование. Ротовой аппарат грызущего типа. Развитие с неполным превращением. Личинки развиваются в воде (наяды) и имеют трахейные жабры.

Инфракласс Новокрылые объединяет более высокоорганизованных насекомых. Их крылья складываются на спине, что позволило этим насекомым занять самые разные экологические ниши. Ротовые аппараты разнообразного строения.

Отряды, входящие в инфракласс Новокрылые, могут быть объединены по особенностям развития в два отдела: насекомые с неполным и насекомые с полным превращением. Ниже дана характеристика в основном тех отрядов, представители которых имеют значение для сельского и лесного хозяйства.

ИНФРАКЛАСС ДРЕВНЕКРЫЛЫЕ (PALAEOPTERA). Офяд Стрекозы (Odonata). Известно около 4,5 тыс. видов стрекоз, из них в России встречается около 160 видов. Голова стрекоз несет пару огромных сложных глаз (рис. 139). Брюшко тонкое и вытянутое. Крылья две пары сходного строения и с густой сетью жилок. У большинства стрекоз крылья не складываются на спине, но есть виды, у которых они складываются вертикально над спиной. Современных стрекоз делят на подотряды равнокрылых и разнокрылых. Равнокрылые могут складывать крылья. Ротовые органы у них грызущие. Усики короткие.

Самки стрекоз откладывают яйца на водные растения или в воду. Вышедшие из яиц наяды — хищные личинки — имеют специальный орган захвата добычи — маску (видоизмененная нижняя губа; рис. 139).

8

10

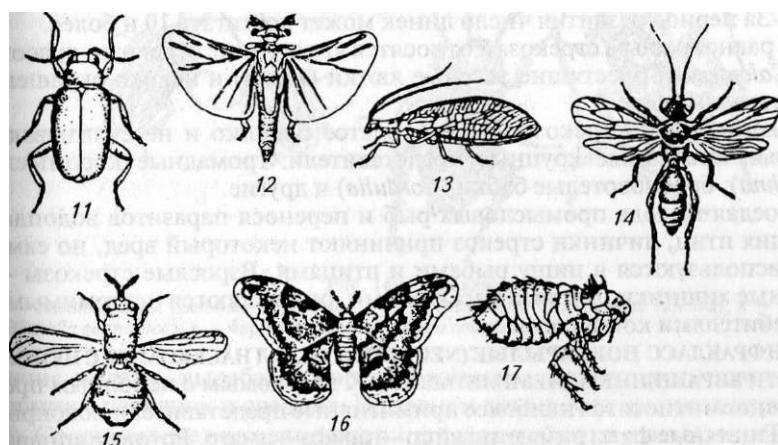


Рис. 138. Представители отрядов крылатых насекомых:
 1 — Поденки (Ephemeroptera); 2 — Стрекозы (Odonata); 3 — Тараканы (Blattodea); 4 —
 Веснянки (Plecoptera); 5 — Уховертки (Dermaptera); 6 — Прямокрылые (Orthoptera); 7 —
 Равнокрылые хоботные (Homoptera); 8 — Клопы (Hemiptera); 9 — Вши (Anoplura); 10 —
 Трипсы (Thysanoptera); 11 — Жуки (Coleoptera); 12 — Веерокрылки (Strepsiptera); 13 —
 Сетчатокрылые (Neuroptera); 14 — Перепончатокрылые (Hymenoptera); 15 — Двукрылые
 (Diptera); 16 — Бабочки (Lepidoptera); 17 — Блохи (Siphonaptera)

Рис. 139. Стрекоза-коромысло:
a — имаго; *б* — личинка; *в* — голова личинки с вытянутой маской

Наяды питаются личинками комаров, поденок и т. п. Личинки последнего возраста вылезают из воды и после линьки превращаются в стрекозу. За период развития число линек может достигать 10 и более.

К равнокрылым стрекозам относятся ярко-синие стрекозы-красотки (*Calopteryx*), блестящие зеленые лютки (*Lestes*) и неярко окрашенные стрелки (*Agriorni*).

Разнокрылые стрекозы имеют толстое брюшко и не складывают крылья. Это самые крупные представители: громадные коромысла (*Aeschna*), бронзовотельные бабки (*Cordulia*) и другие.

Поедая молодь промысловых рыб и перенося паразитов водопла-вающих птиц, личинки стрекоз причиняют некоторый вред, но сами они используются в пищу рыбами и птицами. Взрослые стрекозы — дневные хищники, питаясь насекомыми, они являются неутомимыми истребителями комаров.

ИНФРАКЛАСС НОВОКРЫЛЫЕ (NEOPTERA). ОТДЕЛ НАСЕКОМЫЕ С НЕПОЛНЫМ ПРЕВРАЩЕНИЕМ (НЕМИМЕТАБОЛА). К насекомым с неполным пре-вращением относятся наиболее примитивные представители новокры-лых. Типичные фазы развития: яйцо—нимфа—имаго. Ротовые аппара-ты в основном грызущие или колюще-сосущие.

Отряд Таракановые (Blattodea). Тараканы — крупных и средних раз-меров насекомые с уплощенным телом. Покровы мягкие и на ощупь маслянистые. Голова подогнута под переднегрудь и имеет тонкие и длинные усики. Ротовой аппарат грызущий (рис. 140). Кожистые над-крылья и сложенные веером задние крылья нередко укорочены или полностью редуцированы (чаще у самок). Обычно тараканы имеют па-

Рис. 140. Тараканы:
а — лапландский (*Ectobius lapponicus*); б — реликтовый (*Cryptocercus relictus*); в — прусак (*Blattellagermanica*); г — черный таракан (*Blatta orientalis*)

хучие железы, вырабатывающие феромоны. Задние ноги ненамного длиннее передних и средних. На заднем конце тела имеются церки.

Современные виды лишены яйцеклада и обычно откладывают яйца и своеобразном яйцевом коконе — оотеке, каждая из которых может содержать несколько десятков яиц. Есть живородящие виды тараканов. К партеногенетическим формам относится суринамский таракан (*Rycnoscelus surinamensis*). Яйца в оотеках могут длительное время переживать неблагоприятные условия среды.

Насчитывают около 2,5 тыс. видов тараканов, в основном обитающих в тропиках. Некоторые синантропные виды живут повсюду в жи-

лицах человека. Эти насекомые теплолюбивы и влаголюбивы, отличаются светобоязнью, весьма неприхотливы в выборе пищи. Развитие протекает от 2 мес до 5 лет, за это время проходит пять—девять линек. Имаго живут до 7 лет. У многих форм отмечена специфическая фауна кишечных симбионтов, помогающих усваивать питательные вещества.

В нашей стране обитает около 50 видов тараканов, предпочитающих южные регионы страны. В естественных условиях они являются сапрофагами и живут в лесной подстилке, гниющей древесине и почве. В лесах Европы обычен лапландский таракан (см. рис. 140), который внешне похож на рыжего домового таракана.

В жилищах человека живут крупный черный таракан (*Blatta orientalis*), который был завезен в Европу из тропических стран около 300 лет назад, и мелкий рыжий таракан-prusак (*Blattella germanica*). У самок черного таракана крылья недоразвиты, а у рыжего таракана они развиты и у самок, и у самцов. Развитие у прусака завершается через 5—6 мес. В Америке распространен синантропный очень крупный таракан американский (*Periplaneta americana*). Синантропные виды тараканов при нарушении санитарно-гигиенических норм могут загрязнять продукты питания, разносить возбудителей кишечных инфекций и яйца гельминтов.

Отряд Богомоловые (Mantodea). Богомолы — крупные дневные хищники, скрывающиеся среди растений. Они ловят добычу с помощью хватательных передних ног (рис. 141), у которых удлиненная голова входит своим зазубренным краем в желобок длинных бедер, что позволяет удерживать жертву. Простирая свои передние ноги, богомолы застаивают в ожидании жертвы, медленно поводя из стороны в сторону маленькой треугольной головой с выпуклыми глазами и хорошо развитыми усиками. Несмотря на свою флегматичность, богомолы способны на молниеносные броски. Крупные тропические богомолы могут захватить даже мелких птиц. Крылья у богомолов листовидные. Ротовой аппарат грызущего типа. Богомолам присущ каннибализм: широко распространено поедание самкой самца после спаривания.

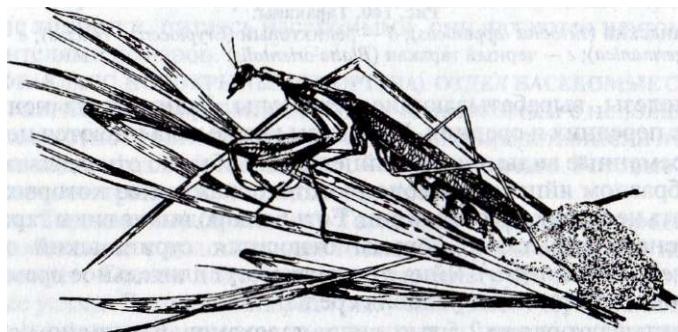


Рис. 141. Богомол обыкновенный (*Mantis religiosa*)

Самки откладывают яйца в оотеках, прикрепляя их к стеблям растений. Вылупившиеся нимфы мало похожи на богомола. В течение года после 7—8 линек молодые богомолы достигают половой зрелости.

Отряд представлен примерно 2 тыс. видов, из которых лишь немногие обитают за пределами субтропиков. В южных регионах нашей страны распространен обыкновенный богомол (*Mantis religiosa*), в Крыму, Закавказье и средней Азии обитает грациозный богомол — эмпуза пестистоусая (*Empusa pennicornis*).

Отряд Прямокрылые (Orthoptera). Типичные прямокрылые — крупные (до 8 см) насекомые с сильными бедрами задних ног, мощными жвалами и двумя парами крыльев. Ротовой аппарат грызущий. На голове имеются нитевидные усики. Передние крылья, плотные и узкие, покрывают перепончатые задние крылья, которые при полете расправляются веером. Задние ноги часто прыгательного типа, отличаются от двух передних пар ног большой длиной за счет удлиненных бедер и голеней; благодаря этому насекомые способны совершать большие прыжки: длина прыжка у азиатской саранчи до 5 м, у кобылки — до 70 см. Обладая хорошо развитыми органами зрения и антеннами, прямокрылые обитают в кустарниках и траве. Охотясь за другими насекомыми, они совершают прыжки и короткие перелеты; кроме того, они обгрызывают растения. Призывая друг друга, прямокрылые громко стрекочут. Звуки возникают в результате трения друг о друга некоторых частей тела: кузнечики и сверчки трут крыло о крыло, а саранча и кобылки — бедро задних ног о края надкрыльев. У многих форм имеются органы слуха. Для каждого вида специфично свое стрекотание, которое является половым сигналом.

По разнообразию жизненных форм и числу видов (более 20 тыс.) прямокрылые конкурируют с наиболее совершенными насекомыми. Среди них есть сапрофаги и фитофаги, но нет паразитов и переносчиков болезней. В России обитает около 700 видов. К прямокрылым относится несколько семейств, из которых наиболее многочисленны: семейство Кузнечики (Tettigoniidae), семейство Сверчки (Gryllidae), семейство Саранчовые (Acrididae) и семейство Медведки (Gryllotalpidae). Многие прямокрылые являются вредителями растений (рис. 142).

Саранчовые — наиболее обширное семейство прямокрылых: насчитывает около 10 тыс. видов, из которых в нашей стране обитает около 500 видов. Саранча всегда воспринималась как символ бедствий, опустошений и голода. Это довольно крупные насекомые (до 10 см длиной), похожие на кузнечиков, но отличающиеся от них короткими усиками (до 2 см). Они фитофаги, и поэтому среди них много опасных вредителей сельского хозяйства. Звуки издают только самцы. Органы слуха у саранчи расположены на первом сегменте брюшка.

Самки откладывают яйца в норки, вырытые в почве коротким яйцекладом. Яйца выделяются вместе с пенистым секретом особых желез. Этот секрет затвердевает, скрепляя частицы почвы вокруг яиц и заключая яйца в колбасовидную кубышку (рис. 142), имеющую форму мешочка с земляными стенками. В кубышке может быть от 10 до 115 яиц.

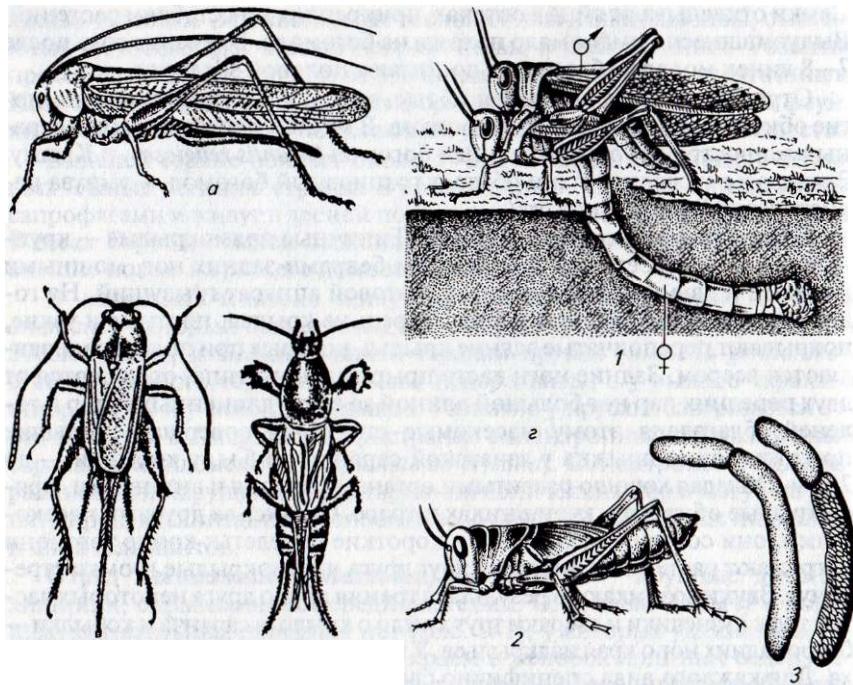


Рис. 142. Прямокрылые:
а — кузнецик обыкновенный (*Tettigonia viridis*); б — сверчок степной (*Gryllus desertus*);
в — медведка обыкновенная (*Gryllotalpa gryllotalpa*); г — саранча перелетная; 1 — имаго;
2 — пешая саранча; 3 — кубышки

Из яиц выходят личинки, которые похожи на взрослых насекомых, но имеют зачаточные крылья. Личинки саранчовых обычно вылупляются из яиц в конце весны. Пройдя четыре—семь линек, личинки за 3-4 мес достигают половой зрелости. В это же время среди личинок происходит дифференцировка на стадные и одиночные формы. Стai (кулиги) личинок именуют пешей саранчой. Эти кулиги могут совершать большие кочевки, двигаясь только днем и уничтожая по пути посевы (за световой день кулига может пройти до 70 км). Насекомые постоянно движутся, выбирая съедобные для себя растения. Отсутствие таких растений заставляет переходить на другие виды — культурные злаки и др. Экскременты саранчовых являются основным субстратом для развития почвенных микроорганизмов. Оптимальная численность саранчовых способствует поддержанию плодородия почв.

Взрослая саранча (крылатая) хорошо летает и ее стаи, насчитывающие огромное число особей, совершают далекие перелеты, нанося более разрушительные повреждения посевам. Так, азиатская саранча летит со скоростью 50 км/ч, преодолевая без посадки до 2 тыс. км. Из

многих видов саранчи, обитающих в России, наибольший вред приносят перелетная, или азиатская, саранча (*Locusta migratoria*), распространенная в Африке и Евразии; пустынная саранча-шистоцерка (*Schistocerca gregaria*), обитающая в Индии, Африке, Передней Азии, и итальянская кобылка (*Calliptamus italicus*). Наиболее эффективна борьба с саранчой в местах ее размножения и с пешей саранчой.

Саранчу едят почти все виды животных. В теле саранчовых много жира, белка и витаминов. По вкусу саранча не уступает креветкам и ракам. Африканцы едят саранчу в любом виде. Саранча может стать объектом разведения для производства комбикормов.

Кобылки близки к саранче, но отличаются более мелкими размерами и обычно яркой окраской задней пары крыльев. Многие виды кобылок наносят вред сельскохозяйственным растениям, травостою лугов и пастбищ. Вместе с тем они являются пищей для многих млекопитающих и птиц.

Кузнецики в отечественной фауне представлены многими видами. Их легко узнать по очень длинным тонким усикам (длиннее тела) и хорошо выраженному яйцекладу. У самцов на надкрыльях имеются органы стрекотания. Органы слуха расположены на передних голенях. Вредоносная деятельность кузнециков не столь велика, а сами они служат пищей для многих животных и даже человека. Кузнецики питаются растительной пищей, но среди них есть и хищные формы, поедающие вредных насекомых. Самки кузнециков откладывают яйца в стебли растений. Поскольку кузнецики обитают в луговой растительности, они обычно имеют зеленую окраску. Наиболее часто в средней полосе встречается обыкновенный кузнецик (*Tettigonia viridis*), который ведет хищный образ жизни.

Сверчки отличаются от кузнециков трехчлениковыми конечностями (у кузнециков они четырехчлениковые) и более темной окраской. Они фитофаги. Самки откладывают яйца в стебли растений или в почву. Наибольший ущерб приносят в нашей стране степной сверчок (*Gryllus desertus*), в жилищах человека встречается домовый сверчок (*Acheta domestica*). Последний питается пищевыми остатками.

Медведки — роющие прямокрылые, имеющие мощные, укороченные и широкие копательные передние ноги (рис. 142). Они роют в почве ходы и питаются корнями и корнеклубнеплодами растений, нанося существенный вред, особенно в южных регионах страны. Наиболее часто встречается медведка обыкновенная (*Gryllotalpa gryllotalpa*).

Отряд Вши (Anoplura). Это накожные паразиты многих млекопитающих и человека (рис. 143). Вши — один из самых молодых отрядов насекомых. Они известны около 50 млн лет. Эти паразиты-кровососы, по-видимому, ведут свое происхождение от предков, которые сначала эпизодически, а затем систематически питались кровью древних млекопитающих. Среди паразитических насекомых нет другой такой группы, существование которой было бы столь неразрывно связано с человеком, как это характерно именно для вшей.

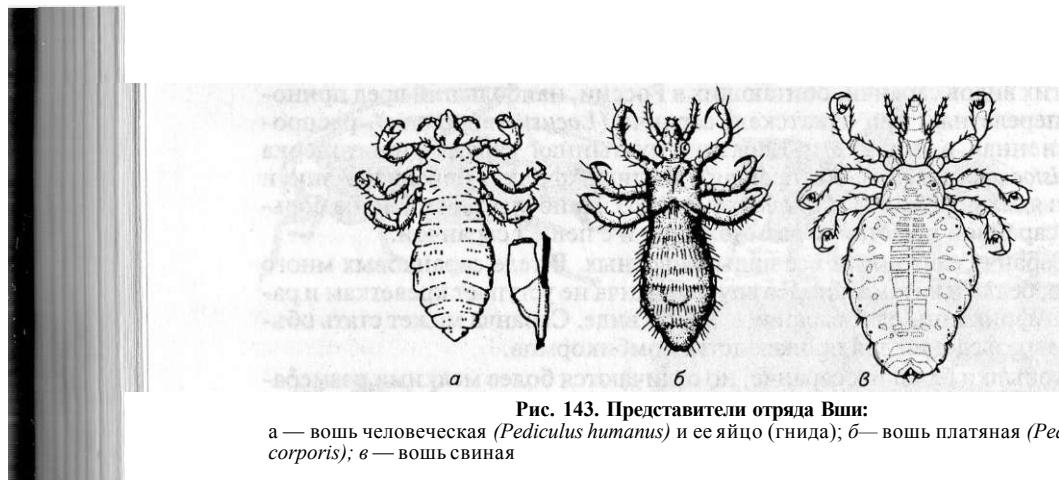


Рис. 143. Представители отряда Вши:
а — вошь человеческая (*Pediculus humanus*) и ее яйцо (гнида); б — вошь платяная (*Pediculus corporis*); в — вошь свиная

Вши — вторичнонеспрытые насекомые, у которых небольшое уплощенное тело. Глаза не развиты. Лапки цепкие и несут загнутые коготки, которые загибаются и вкладываются в промежуток между двумя шпорами голени, образуя кольцо, охватывающее волос хозяина. Это позволяет вшам прочно держаться на волосах. Ротовой аппарат колюще-сосущего типа. В спокойном состоянии колющая часть втянута в рот. Вши сосут кровь, прокалывая кожу хозяина.

Известно около 150 видов вшей — паразитов млекопитающих. На крупном рогатом скоте паразитирует бычья вось, на оленях — оленя, на тюленях — тюленя (в ноздрях). На человека паразитируют три вида вшей: головная, или детская, вось (*Pediculus capitis*), платяная вось (*P. corporis*) и плошица, или лобковая, вось (*Phthirus pubis*). Первые два вида морфологически настолько похожи друг на друга, что есть сомнения в правильности их разделения. Между участками тела человека для вшей существуют непреодолимые границы, которые предопределяются характером волоса — среди обитания плошиц. Плошицы могут держаться за волос только определенной тощины — на лобке, бровях, ресницах, бороде и подмышках.

Самки вшей откладывают яйца (гниды), приклеивая их к волосам или внутренней стороне одежды, обычно в местах пересечения волос (нитей). Человечья вось является распространителем среди людей сыпного и возвратного тифа.

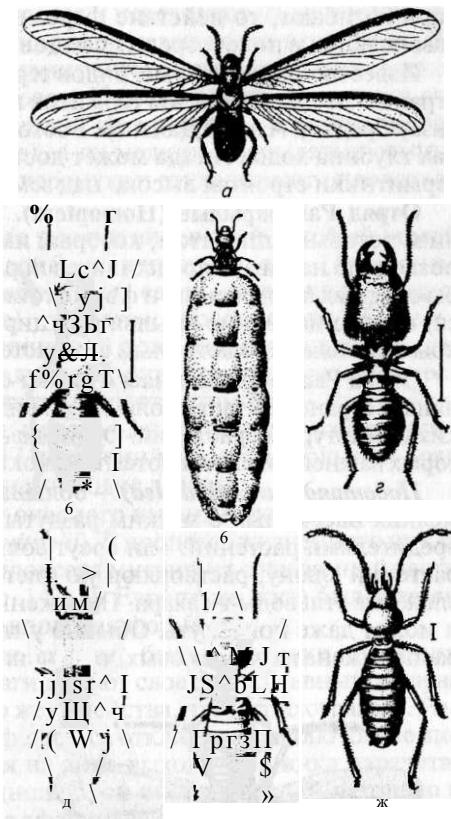
Отряд Пухоеды и Власоеды (Mallophaga). Это мелкие насекомые, похожие на вшей, но отличающиеся от них грызущим роговым аппаратом и

очень большой головой, ширина которой превышает ширину груди (рис. 144). Различные виды паразитируют в перьевом покрове птиц (пухоеды) и в волосяном покрове млекопитающих (власоеды). Встречаются почти на всех домашних животных, а также на диких животных. Питаются пухом, шерстью и роговым слоем эпидермиса кожи хозяев, вызывая кожные заболевания, разрушая перьевый "волосянной" покровы.

Рис. 144. Пухоед собачий



Рис. 145. Термиты:
а — молодая крылатая самка; б — самка с брюшком, наполненным яйцами; в — самец, сбросивший крылья; г, д, е — солдаты; ж — рабочий термит



Отряд Термиты (Isoptera).
Термиты подобно муравьям являются общественными насекомыми, их часто называют «белыми муравьями», так как они строят гнезда — терmitники. У них наблюдается полиморфизм особей (рис. 145): самки (царицы), крылатые самцы, бескрылые неполовозрелые рабочие и солдаты мужского и женского пола. Термиты в отличие от муравьев развиваются с неполным превращением, ведут скрытный образ жизни и питаются в основном древесиной и плесневыми грибами (муравьи развиваются с полным метаморфозом и питаются обычно животной пищей). Ротовой аппарат грызущего типа. Крылатые самки и самцы имеют по две пары одинаковых крыльев. Покровы тела непигментированные и слабо склеротизированы.

Начало новой семьи дают крылатые самки и самец, которые после спаривания отламывают крылья и роют в земле ходы, где самка откладывает яйца. Из яиц выходит первое поколение личинок, которых выкармливают родители. Из этого поколения формируются рабочие особи, на них и ложатся все последующие заботы о семье термитов. Рабочие особи заботятся о потомстве, строят гнездо, разводят культуру грибов, являющихся пищей термитов. Самка увеличивается в размерах и становится неподвижной, она теперь выполняет лишь одну функцию — откладывание яиц (суточная плодовитость может достигать 3 тыс. яиц). В большой камере терmitника находятся самка-царица и самец-царь, уход за которыми лежит на рабочих термитах. Охрану терmitника выполняют солдаты, имеющие большую голову с мощными челюстями.

Самки постоянно выделяют особые вещества, содержащие феромоны, которые слизывают развивающиеся нимфы. Феромоны тормозят развитие нимф, и они превращаются в рабочих особей. Если царица и

царь погибают, то действие феромонов прекращается и часть нимф развиваются в половозрелых самцов и самок.

Известно около 2,5 тыс. видов термитов, обитающих в тропических странах. Всего пять видов термитов встречается на юге Украины, Кавказе, Средней Азии и Дальнем Востоке. В засушливых и жарких регионах глубина ходов гнезда может достигать Юм. Во влажных тропиках термитники строятся высоко над землей из цементированной глины.

Отряд Равнокрылые (Homoptera). Это насекомые с колюще-сосущим ротовым аппаратом, который имеет вид членистого хоботка, что позволило называть представителей равнокрылых хоботными. Две пары сходных по строению и форме тонких крыльев в покое складываются, но иногда задние крылья редуцированы. Основной пищей равнокрылых является клеточный сок растений.

Отряд Равнокрылые насчитывает более 30 тыс. видов, из которых в нашей стране обитает около 4 тыс. видов, из них много вредителей диких и культурных растений. Отряд делится на шесть подотрядов, из которых важнейшими являются тли, кокциды, цикадовые и листоблошки.

Подотряд Тли (Aphidodea) — большая группа равнокрылых хоботных мелких насекомых с мягким раздутым телом (рис. 146), являющаяся вредителями растений. Тли сосут соки растений, вводя внутрь тканей растений слону, растворяющую клеточные стенки и превращающую сложные углеводы в сахара. Пораженное тлей растение отстает в росте и может даже погибнуть. Обычно у тлей на пятом брюшном сегменте расположена пара восковых трубочек. Бороться с тлей сложно, ибо их тело покрыто восковым пушком, оберегающим от ядохимикатов. Тли плодовиты: одна самка за год дает до 17 • 10 потомков.

У многих тлей наблюдаются сложные циклы развития, когда бескрылые партеногенетические поколения сменяются крылатыми половыми. Так, на ветках черемухи зимуют оплодотворенные яйца черемухо-овсяной тли, из которых развивается бескрылое поколение партеногенетических самок-основательниц. Они дают второе поколение партеноге-

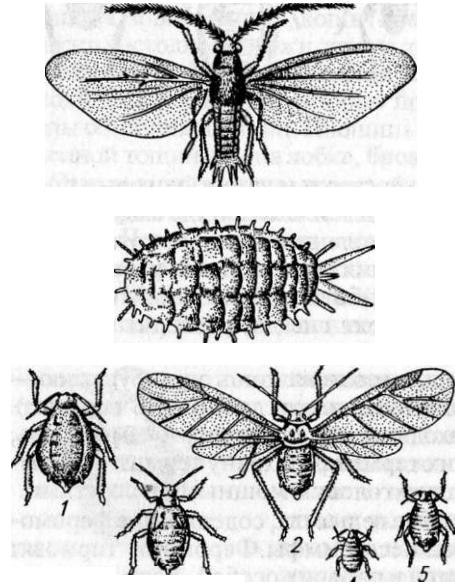


Рис. 146. Представители отряда Равнокрылые:
а — самец красного гигантского червеца;
б — самка мучнистого червеца;
в — капустная тля; 1 — бескрылая самка;
2 — крылатая самка; 3 — нимфа;
4, 5 — личинки

нетических самок, которые окрыляются и перелетают на другое кормовое растение — овес. Этих кочующих самок называют эмигрантами. На новом кормовом растении самки-эмигранты снова дают партеногенетическое поколение самок-полоносок. К осени самки-полоноски откладывают яйца — крупные и мелкие. Из крупных яиц развиваются крылатые самки (панмиктические), а из мелких — крылатые самцы. После спаривания самки летят на черемуху и откладывают оплодотворенные зимующие яйца.

В нашей стране известно более тысячи видов тлей. Особый ущерб наносят яблонная тля (*Aphis pomi*) и капустная тля (*Brevicoryne brassicae*). Опасным вредителем яблоневых садов, завезенным в нашу страну (родина тли — Америка), является кровяная тля (*Eriosoma lanigerum*). На яблонях зимуют личинки и реже взрослые самки, которые забираются в трещины коры, дупла, корни и другие укромные места. Весной личинки начинают питаться и дают потомство — самок. Развитие происходит очень быстро: через 2—3 нед личинки уже превращаются в самок, и так от 12 до 17 поколений за сезон. Большой вред приносят хермесы, повреждающие хвойные деревья.

Необычен жизненный цикл у опасного карантинного вредителя — виноградной филлоксеры (*Viteus vitifoliae*). В процессе жизненного цикла у себя на родине (в Америке) филлоксера мигрирует с надземной части виноградной лозы на корни (рис. 147). В Европе же, куда был завезен вредитель, развивается лишь его корневая форма.

У тлей много врагов: мелкие птицы, божьи коровки, личинки мух-журчалок и другие энтомофаги делают свое дело. Главный же враг тлей — афелинус, паразит из того же семейства, что и трихограмма. Родом он из Сев. Америки. Самка афелинуса откладывает в тлю или ее личинку яйцо, и уже через 2—4 дня из яйца выходит личинка паразита. Тля же погибает. Афелинус — однолюб, он откладывает яйца только в тело кровяной тли. Обычно самка афелинуса откладывает 60—140 яиц, давая 6—9 поколений (на юге — до 12 поколений). Зимует афелинус в теле тли и переносит относительно большие морозы. Афелинуса завезли в 1926 г., и он хорошо приспособился к нашим условиям.

Подотряд Кокциды (Coccoidea) представлен червецами и щитовками. Это весьма специализированные эктопаразиты южных растений, в частности цитрусовых. Они характеризуются наличием полового диморфизма: самки бескрылые и часто неподвижные (ноги редуцированы), а самцы крылатые. Личинки также неподвижны. Бескрылые самки покрыты сверху щитком (или восковыми выделениями) из особого воскоподобного вещества. Они прокалывают хоботком покровы растений и сосут сок (рис. 148). Большой вред наносят субтропическим культурам калифорнийская щитовка и австралийский червец. В борьбе с кокцидами используют хищных божьих коровок и паразитических перепончатокрылых — хальцид.

Подотряд Листоблошки (Psylloidea). Это мелкие крылатые равнокрылые насекомые с нежными покровами. В нашей стране обитает около 100 видов листоблошек, внешне напоминающих тлей, но в отли-

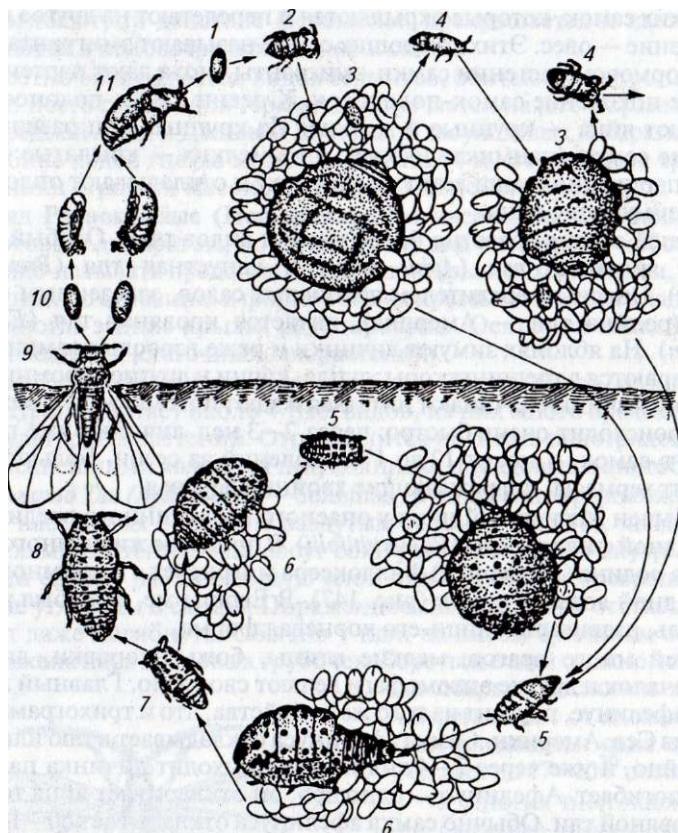


Рис. 147. Цикл развития филлоксера:
 1 — яйцо; 2 — самка-основательница; 3 — яйцекладущая самка; 4 — личинка; 5 — зимующая личинка; 6 — яйцекладущая самка корневой филлоксера; 7 — личинка корневой филлоксера; 8 — нимфа; 9 — крылатая форма; 10 — ее яйца; 11 — спаривание

чие от них способных прыгать. Самки откладывают яйца на листья растений. Эти яйца снабжены коротким стебельком. Взрослые особи весьма активны и при испуге прыгают с листьев. Размножаясь в огромных количествах, они причиняют заметный вред плодовым насаждениям. Листоблошки сосут соки растений. Весь сахар, содержащийся в соке, в организме листоблошек не усваивается, и часть его выделяется с экскрементами насекомых. Побеги, на которых находятся листоблошки, становятся липкими от этого сладкого продукта (падь). Падью питаются муравьи, мухи и зачастую пчелы и осы.

Среди листоблошек весьма распространены яблонная (*Psylla mali*) и грушевая (*Psylla pyri*) медяницы, нимфы которых повреждают почки и бутоны плодовых деревьев.

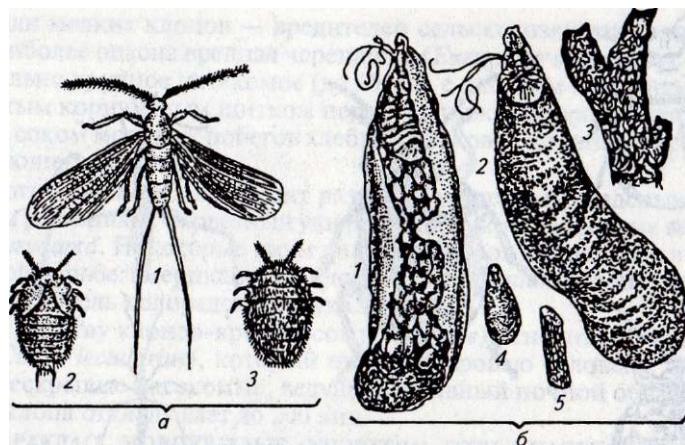


Рис. 148. Червецы и щитовки:
а — кошениль (*Dactylopins coccus*); 1 — самец; 2 — самка с брюшной стороны; 3 — самка со спинной стороны; б — запятовидный червец (*Lepidosaphes ulmi*) 1 — самка с брюшной стороны; 2 — самка со спинной стороны; 3 — веточка с самками; 4 — самец; 5 — веточка с самцами

(*П о д о т р я д Цикадовые (Auchenorrhyncha)* включает три семейства: настоящих певчих цикад (*Cicadidae*), цикадок (*Jassidae*) и пенниц (*Cercopidae*). Это крупные (до 5 см) насекомые. Обычно крылья имеют и самки, и самцы. Многие цикады вредят сельскохозяйственным растениям и могут переносить вирусные болезни растений.

Певчие цикады — наиболее крупные представители подотряда. Более тысячи видов обитают в нашей стране, предпочитая южные регионы. Так, в южных дубравах широко распространена дубовая цикада (*Tibicen haematodes*), а горная цикада (*Cicadetta montana*) встречается в дубравах даже на юге Московской области (рис. 149). Нимфы цикад руют в почве норки, питаются соком корней растений и развиваются в земле несколько лет. Взрослые певчие цикады живут один сезон, обитая на деревьях, питаясь соком стеблей и издавая стрекочущие звуки. Самки откладывают яйца в побеги кустарников и деревьев, подпиливая их своим яйцекладом. Вылупившиеся личинки сразу же мигрируют в почву.

Настоящие цикадки в больших количествах живут в травостое, питаюсь соком растений и быстро перелетая с места на место. Они похожи на певчих цикад, но меньше по размерам.

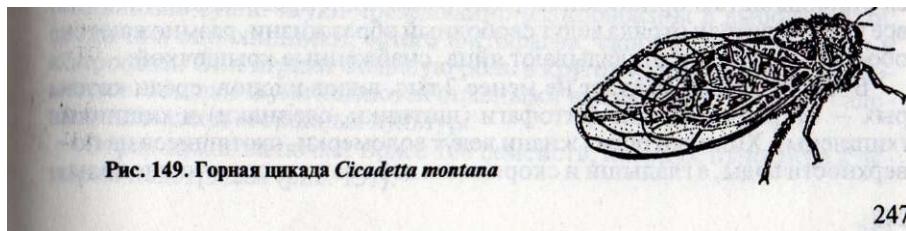


Рис. 149. Горная цикада *Cicadetta montana*

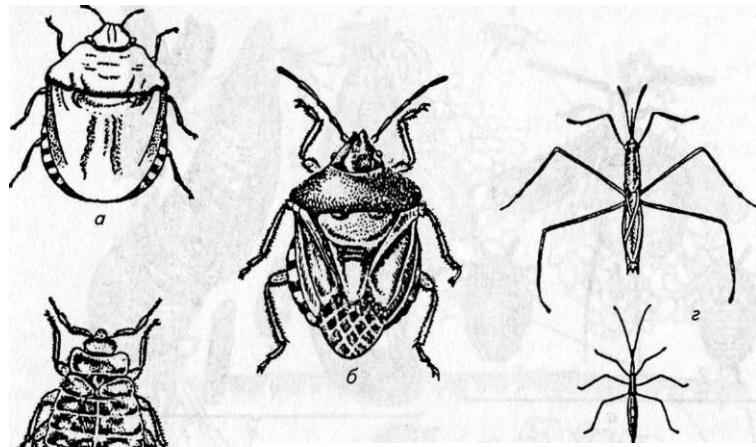


Рис. 150. Полужесткокрылые:
а — клоп-черепашка *Eurygaster integriceps*; б — ягодный клоп *Dofycoris baccarum*; в — постельный клоп *Cimex lectularius*;
г — водомерки *Gerris* и *Naucoris*

Пенницы (слюнявицы) питаются соками растений. Их личинки выделяют пену, защищающую их нежное тело от высыхания и врагов. Часто на траве можно обнаружить скопления пены, напоминающие скопления слюны. Наиболее часто на лугах встречается обыкновенная слюнявица (*Philaenus spumarius*).

Отряд Полужесткокрылые, или Клопы (Hemiptera). Это обширная группа насекомых, по особенностям строения близких к равнокрылым хоботным. Но у полужесткокрылых имеются колющий членистый хоботок и полужесткие передние крылья (рис. 150). В нерабочем положении крылья сложены плоско на спине. Среди полужесткокрылых много паразитов теплокровных животных и вредителей растений, а также хищников, питающихся гемолимфой насекомых. Для клопов типично наличие пахучих желез, секрет которых имеет защитное значение.

Среди насекомых с неполным превращением клопы лидируют по обилию видов (более 40 тыс.), разнообразию освоенных сред и источников пищи. Многие из них — фитофаги, заселившие все типы растительных ассоциаций, другие — паразиты и хищники или сапрофаги. Клопы обычно обитают на растениях и в пресных водах, часто в почве, гнездах птиц, норах грызунов, жилище человека. Все клопы отличаются стабильностью организации, хода развития и превращения. Почти все представители отряда ведут свободный образ жизни, размножаются обоеполым путем и откладывают яйца, снабженные крышечкой.

В нашей стране обитает не менее 2 тыс. видов клопов, среди которых — открытоживущие фитофаги (щитники, слепняки) и хищники (хищнецы). Хищный образ жизни ведут водомерки, охотящиеся на поверхности воды, а гладыши и скорпионы добывают пищу в толще воды.

Среди мелких клопов — вредителей сельскохозяйственных растений наиболее опасна вредная черепашка (*Eurygaster integriceps*). Это относительно крупное насекомое (до 15 мм) с плоским овальным телом, покрытым коричневым щитком переднеспинки. Взрослые клопы питаются соком молодых побегов хлебных злаков, а личинки — соком из наливающегося зерна.

Многие виды клопов вредят различным полевым, садовым и огородным растениям. Серьезный ущерб наносят крестоцветные клопы из рода *Eurydama*. Некоторые виды хищных клопов используют в биологической борьбе: американский клоп (*Perilus boiculatus*) культивируется как истребитель колорадского жука.

К семейству клопов-кровососов (*Cimicidae*) относится постельный клоп (*Cimex lectularius*), который питается кровью человека. Это вторично бескрылые насекомые, ведущие активный ночной образ жизни. Самка клопа откладывает до 200 яиц.

ИНФРАКЛАСС НОВОКРЫЛЫЕ (NEOPTERA). ОТДЕЛ НАСЕКОМЫЕ С ПОЛНЫМ ПРЕВРАЩЕНИЕМ (HOLOMETABOLA). Отряд Жесткокрылые, или Жуки (*Coleoptera*). Это самый многочисленный отряд насекомых, насчитывающий около 250 тыс. видов. Передние крылья жуков превратились в надкрылья — выпуклые и жесткие, без жилок. В покое они прикрывают сложенные тонкие, нежные и перепончатые задние крылья. Жуков можно считать двукрылыми насекомыми, поскольку их полет осуществляется с помощью лишь одной задней пары крыльев, надкрылья же выполняют сложную, стабилизирующую полет функцию. Крепкое и компактное тело жуков, длина которого варьирует от 0,3 до 260 мм, имеет особенно жесткие покровы. Несмотря на обилие видов, разнообразие размеров и жизненных форм, основная массажесткокрылых сохраняет однотипный внешний облик, план строения и однотипное развитие. Ротовой аппарат жуков грызущего типа позволяет питаться им твердой пищей. Глаза только фасеточные. Развитие у жуков происходит с полным метаморфозом (гистолитическим), изредка усложненным до гиперметаморфоза.

Личинки жуков червеобразные, с крупной склеротизированной головой и тремя парами членистых ножек на трех первых сегментах тела. У некоторых форм ножки могут быть редуцированы. Часто личинки жуков обитают в иной среде, чем взрослые насекомые. Так, у жуков-хрущей, живущих на растениях, личинки развиваются в почве, питаясь корнями. Божьи коровки и их личинки живут в одной среде, питаясь сходной пищей.

Жуки встречаются во всех ландшафтных зонах, заселяя все ярусы биоценозов суши. Жуки чрезвычайно разнообразны в выборе пищи: среди них есть хищники, много фитофагов, сапрофаги, некрофаги и копрофаги. Они играют большую роль в круговоротах биогенных элементов. Многие жуки являются опасными вредителями лесных, садовых, полевых и огородных культур.

Отряд жуков включает более 100 семейств, которые относятся к четырем подотрядам (рис. 151).

Рис. 151. Жесткокрылые:
а — хлебная жужелица (*Zabrus tenebrioides*); б — стафилин (*Ocypus*); в — июньский хрущ (*Amphimallon solstitialis*); г — полосатый щелкун (*Agriotes lineatus*); д — семиточечная божья коровка (*Coccinella septempunctata*); е — чернотелка; ж — черный сосновый усач (*Monochamus galloprovincialis*); з — вершинный короед (*Ips acuminatus*; имаго, его «тачка», его ходы); и — колорадский жук (*Leptinotarsa decemlineata*) и его личинка; к — яблонный цветоед (*Anthonomus pomorum*)

Подотряд Плотоядные жуки (Adephaga). Среди плотоядных жуков, включающих семь семейств, наиболее многочисленны два семейства: Жужелицы (Carabidae) и Плавунцы (Dytiscidae). В большинстве они хищники.

Жужелицы — наземные жуки, ведущие активный наземный образ жизни; питаются беспозвоночными (в основном насекомыми). Они выполняют большую работу по уничтожению гусениц бабочек, поеда-

ют слизней и улиток. Есть виды, питающиеся растительной пищей, в том числе пшеницей (хлебная жужелица рода *Zabrus*).

Плавунцы — водные жуки, живущие в стоячих и медленно текущих водоемах и активно охотящиеся на личинок стрекоз, поденок и прочих беспозвоночных животных. Задние ноги у плавунцов плавательные. Личинки плавунцов также хищники. Они оккукливаются на берегу в почве. В нашей стране насчитывают около 300 видов плавунцов, из которых наиболее крупный — окаймленный плавунец (*Dytiscus marginalis*).

Интересны представители плотоядных жуков из семейства Вертячки (*Gyrinidae*), насчитывающего 20 видов. Они двигаются кругами по поверхности пресных водоемов с помощью двух пар ластовидных ног. Глаза у вертячек разделены на верхнюю (видят над водой) и нижнюю (видят под водой) части, что позволяет жукам охотиться на поверхности воды и под водой.

Подотряд Разноядные жуки (*Polyphaga*). К разноядным относится большая часть семейств жуков, чрезвычайно разнообразных по типу питания. Наиболее обширны два семейства: Водолюбы (*Hydrophilidae*) и Страфилиниды, или Коротконадкрыльные (*Staphylinidae*).

Водолюбы — водные жуки, питающиеся в основном растительной пищей, а их личинки — хищники. У личинок на брюшке имеются жаберные выросты. Наиболее часто встречается черный водолюб (*Hydrouspiceus*).

Страфилиниды — многочисленное семейство жуков с короткими надкрыльями, не прикрывающими брюшка. Среди них встречаются в основном хищники, но есть и сапрофаги, и фитофаги. Эти жуки принимают участие в регулировании численности других насекомых и в почвообразовательных процессах.

Подотряд Пластинчатоусые (*Lamellicornia*) характеризуется тем, что у этих жуков усики имеют пластинчатую или гребенчатую булаву.

Семейство Пластинчатоусые (*Scarabaeidae*). Усики у представителей этого семейства имеют плотную пластинчатую булаву. Сюда относятся хрущи, навозники, бронзовки. Хрущи — фитофаги и поэтому среди них много вредителей сельского и лесного хозяйства. Это майский хрущ (*Melolontha hippocastani*), июньский хрущ (*Amphimallon solstitialis*), хлебные жуки (*Anisoplia austriaca*). Вредят как взрослые, так и их личинки. Например, хлебные жуки (*Anisoplia*) с блестящей зеленовато-черной переднеспинкой являются вредителями хлебов. Взрослые жуки питаются мягкими зернами хлебных злаков, а личинки грызут корни.

Навозные жуки питаются навозом и тем самым участвуют в почвообразовательном процессе. Наиболее типичны обыкновенный навозник (*Geotrupes stercorarius*), священный скарабей (*Scarabaeus sacer*) и жук-носорог (*Oryctes nasicornis*).

Семейство Щелкуны (*Elateridae*) — большая группа жуков-фитофагов, имеющих прыгательный аппарат на груди, с помощью которого, оказавшись на спине, жук подскакивает и переворачивается. Существенный вред оказывают проволочники — личинки жуков-щелкунов, живущие в почве и питающиеся корнями растений. На полях широко распространены щелкуны родов *Agriotes* (рис. 151) и *Selatosomus*.

Семейство Кокцинеллиды, или Божьи коровки (Coccinellidae). Это небольшие округлой формы жуки обычно с яркой предупреждающей окраской. В момент опасности божьи коровки выделяют едкую жидкость (гемолимфу). Божьи коровки и их личинки питаются тлей, сосущей соки растений, и поэтому их используют в биологической борьбе с вредными равнокрылыми: тлями, червецами, листоблошками. В нашей стране обычны семиточечная (*Coccinella septempunctata*) и двухточечная (*Adalia bipunctata*) коровки.

Семейство Чернотелки (Tenebrionidae) — в основном обитатели степей и пустынь. Их личинки подобно проволочникам вредят всходам полевых культур (чернотелок часто называют ложнопроволочниками). Мучной (*Tenebrio molitor*) и малый (*Tribolium confusum*) хрущаки поражают разные размолы зерна. Чернотелки выделяют едкую пахучую жидкость, поэтому продукты, зараженные мучным хрущаком, приобретают специфический запах.

Семейство Усачи (Cerambycidae). Жуки этого семейства имеют длинное тело и длинные усы. Взрослые насекомые питаются листьями и цветами, а их личинки развиваются под корой, в древесине стволов деревьев. Многие представители являются вредителями леса. Многие виды тесно связаны с одной породой деревьев, например сосновый усач (*Monochamus galloprovincialis*, рис. 151).

Семейство Листоеды (Chrysomelidae) включает небольших и ярко окрашенных жуков, питающихся растительной пищей. Их личинки тоже фитофаги. Среди представителей листоедов много опасных вредителей лесного и сельского хозяйства: огородные блошки (*Phyllotreta*), тополевый листоед (*Melasoma populi*), колорадский жук (*Leptinotarsa decemlineata*). Колорадский жук — вредитель картофеля, листьями которого питаются и жуки, и их личинки. Жука легко узнать по ярко-желтой окраске с пятью черными продольными полосами на каждом надкрылье.

Семейство Долгоносики (Curculionidae) — самое богатое по числу видов: более 40 тыс. Это фитофаги, поражающие различные части растений: бутоны, цветки, побеги, листья, стебли, корни. Окраска тела разнообразная. Передняя часть головы вытянута в хоботок (головотрубка), на конце которого расположены рот и усики (см. рис. 151). С помощью хоботка долгоносики могут проникать глубоко в ткани растений, где и откладывают яйца. Среди долгоносиков много вредителей. Свекловичный долгоносик (*Bothynoderes punctiventris*) повреждает всходы свеклы, а безногие личинки живут в почве, поражая корни. Яблонный цветоед поражает бутоны яблонь, клеверные семяеды — семена клевера.

Семейство Короеды (Ipidae). Жуки-короеды имеют цилиндрическое тело. Они прогрызают ходы под корой; эти ходы имеют специфическое для каждого вида строение (см. рис. 151). Самка прогрызает маточный ход, где откладывает яйца. От маточного хода отходят личиночные ходы, которые заканчиваются куколочной колыбелькой. Вышедшие из куколок молодые жуки прогрызают отверстия и выходят наружу. Обычно короедами заражаются ослабленные или срубленные деревья. Наиболее типичен для наших лесов короед-типограф (*Ips typographus*),

вредящий хвойным деревьям. Широко распространены большой и малый лубоеды.

Отряд Блохи (Siphonaptera, или Aphaniptera). Это мелкие бескрылые насекомые с удлиненными прытательными ногами и уплощенным с боков телом, что облегчает их движение в волосяном и перьевом покровах хозяев (рис. 152). Ротовой аппарат колюще-сосущий. Эктопаразиты теплокровных животных, каждому виду которых обычно свойствен определенный вид блох. Взрослые блохи питаются кровью хозяев. Личинки и куколки развиваются в почве, норах, трещинах полов, в мусоре, в гнездах птиц. Личинки пытаются разлагающимися органическими остатками. На человеке паразитирует человечья блоха (*Pulex irritans*), которая может жить на собаках, кошках и даже лошадях. Кошачья и собачья блохи не переходят на человека. Много блох паразитирует на грызунах, с которых могут переходить на других животных и человека.

Блохи переносят различные заболевания человека и животных, в частности чуму. Чумные бактерии, попав с кровью больного чумой человека или грызуна в кишечник блох, там не погибают, а размножаются. Зараженная блоха, кусая здорового человека или грызуна, выделяет экскременты с бактериями чумы, которые попадают в ранку при расчесе места укуса.

Отряд Ручейники (Thchoptera). Крылатые насекомые, имаго которых живут очень недолго вблизи воды и обычно не питаются. Личинки гусеницеобразные, живут под водой и строят домики из песчинок, камешков и других материалов, склеивая их паутиной. Дышат с помощью трахейных жабр, развиваются в воде 1—3 года, после чего оккукливаются в домиках. Ротовой аппарат грызущего типа. Личинками питаются многие виды рыб.

Отряд Чешуекрылые, или Бабочки (Lepidoptera). Две пары больших крыльев бабочек покрыты мелкими хитиновыми чешуйками, черепицеобразно налагающими друг на друга (рис. 153). Чешуики представляют собой видоизмененные волоски и зачастую ярко окрашены. У дневных бабочек крылья в покое сложены вертикально над спиной, а у ночных лежат вдоль тела крышеобразно. Ротовой аппарат сосущего типа в виде гибкого хоботка, сворачивающегося в спираль. С помощью хоботка насекомые высасывают нектар из цветков. Личинки бабочек — гусеницы — имеют червеобразную форму. Первые три членика тела ли-

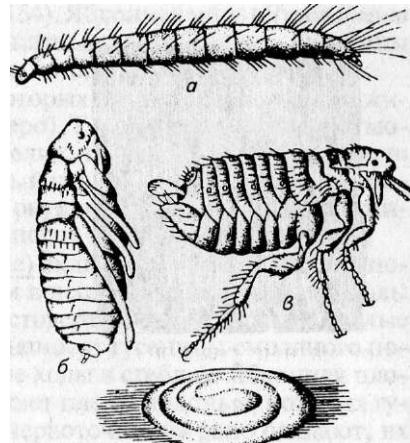


Рис. 152. Блоха собачья:
а — личинка; б — куколка; в — взрослое на-
секомое; г — кокон

Рис. 153. Чешуйки на крыле бабочки:
а — расположение чешуек на крыле дневной бабочки; б, в — форма чешуек разных видов
бабочек

чинок несут по паре настоящих членистых ножек, остальные членики тела — нерасчлененные ложные ножки (обычно пять пар). Ротовой аппарат гусениц грызущего типа. Куколки покрыты, часто в коконе.

Бабочки питаются нектаром цветков или сладкими соками растений, а гусеницы — фитофаги, часто наносят вред растениям.

Известно около 140 тыс. видов бабочек. Следует выделить три подотряда: Равнокрылые (*Jugata*), Разнокрылые (*Frenata*) и Булавоусые (*Rhopalocera*).

Подотряд Равнокрылые (*Jugata*). К подотряду относятся низшие бабочки с примерно одинаковыми по размерам и жилкованию крыльями. Передние крылья имеют на заднем крае выступ, скрепляющий их с задними крыльями. К равнокрылым относятся семейства первичных молей и тонкопрядов. В нашей стране наиболее типичен хмелевой тонкопряд (*Hepialus humuli*), гусеницы которого развиваются на траве.

Подотряд Разнокрылые (*Frenata*). Подотряд включает много семейств бабочек, имеющих крылья разной формы: передние треугольные, а задние закругленные. Подотряд делят на две группы: мелкие и крупные разнокрылые.

Наиболее разнообразна и многочисленна группа мелких разнокрылых чешуекрылых. В основном это мелкие бабочки, входящие в несколько семейств.

Семейство Настоящие моли (*Tineidae*) представлено мелкими и невзрачными по окраске бабочками, у которых зачастую ротовой аппарат не развит. Гусеницы живут в паутинных чехликах. Распространены везде. Большинство видов питается растениями, поэтому среди них много вредителей сельского хозяйства. Зерновая моль поражает зерно, яблон-

пая моль — побеги яблонь и т. п. (рис. 154). Яблонная моль (*Hypopomeita malinella*) — мелкая белая с черными пятнышками бабочка. Гусеницы живут группами на листьях яблонь под тонким слоем паутины.

Имеется много видов, гусеницы которых питаются продуктами животного происхождения (мех, кожа, перо). Так, гусеницы платяной моли (*Tineola biselliella*) повреждают изделия из шерсти, а шубной моли (*Tineapellionella*) паразитируют в теплый период года, а зимуют уже личинки последнего возраста в чехлике, окукливаясь в апреле. Имаго живут всего несколько дней и погибают после того, как отложат яйца.

Семейство Листовертки (Tortricidae) включает бабочек, внешне похожих на молей и вредящих деревьям в лесах, парках и садах. В годы массового размножения дубовая листовертка может оголить целые дубравы. Сосновым посадкам ущерб наносят гусеницы смоляного побеговьюна (*Evetria resinella*), делающие ходы в стеблях. Яблонная плодожорка (*Laspeyresia pomoneUa*) поражает плоды яблонь, в которых гусеницы выгрызают ходы. Яблоки с «червоточиной» рано опадают, их товарная ценность резко снижается.

Семейство Стеклянницы (Aegeriidae) включает бабочек с узкими крыльями, на некоторых участках которых отсутствуют чешуйки. По внешнему виду они похожи на ос (явление мимикрии). Гусеницы стеклянниц прогрызают ходы в корнях и стеблях травы и деревьев. Много среди них и вредителей.

Семейство Огневки (Pyralidae). Эти мелкие бабочки часто имеют яркую окраску. Гусеницы преимущественно фитофаги. Луговой моты-

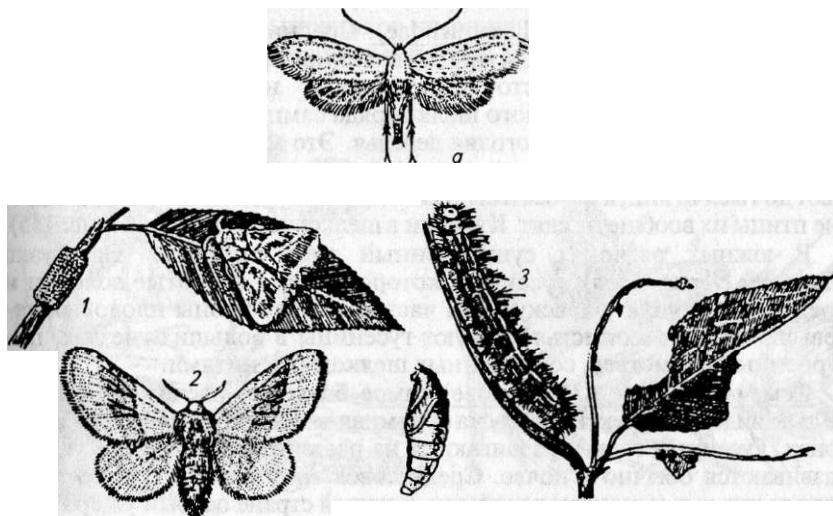


Рис. 154. Разнокрылые:
а — яблонная моль *Hypopomeita*; б — кольчатый шелкопряд (*Malacosoma neustria*);
1 — кладка яиц; 2 — имаго; 3 — гусеница; 4 — куколка

лек (*Pyrausta sticticalis*) вредит многим культурам, давая в южных регионах до трех поколений за сезон. Мучная огневка (*Pyralis farinalis*) вредит зерновым запасам. Восковая огневка (*Galleria mellonella*) питается пчелиным воском и поражает соты. Гусеницы крыжовниковой и смородинной огневок выедают ягоды и опутывают побеги паутиной.

К группе крупных разнокрылых относятся более крупные и специализированные представители бабочек, имеющих широкие задние крылья. К этой группе относится несколько семейств.

Семейство Коконопряды (Lasiocampidae) включает крупных и средних по размерам бабочек с редуцированным хоботком. Гусеницы волосатые с пятью парами брюшных ног; куколки в паутинном коконе. В семействе коконопрядов есть вредные виды: сосновый, сибирский и кольчатый коконопряды; последний вредит не только лиственным лесам, но и садовым деревьям. Особенно опасны вспышки численности кольчатого шелкопряда (*Malacosoma neustria*), которые буквально оголяют дубовые леса и сады (см. рис. 154). Бабочки шелкопряда откладывают яйца колечком вокруг веток деревьев.

Семейство Настоящие шелкопряды (Bombycidae). К семейству относится тутовый шелкопряд (*Bombyx mori*), используемый в шелководстве для получения натурального шелка. Гусеницы шелкопряда имеют железы, выделяющие белковое вещество — фибронин, застывающее на воздухе и превращающееся в шелковую нить. Эти нити гусеница использует для плетения кокона, в котором оккукливается. В некоторых странах разводят дубового шелкопряда (*Antherea pernyi*); из коконов этой бабочки получают более грубую пряжу, идущую на изготовление ткани чесучи.

Семейство Волнянки (Limenitidae). Средних размеров бабочки с волосистым телом. Гусеницы многих волнянок вредят лесу и плодовым культурам. Наиболее часто вред наносит непарный шелкопряд (*Limantria dispar*). У непарного шелкопряда самцы мельче самок. Гусеницы питаются листьями, оголяя деревья. Это многоядный вредитель, повреждающий более 300 видов растений. Самки плодовиты — откладывают до тысячи яиц, а волосатых гусениц птицы поедают неохотно, а многие птицы их вообще не едят. Куколки в шелковистых коконах (рис. 155).

В южных регионах существенный вред приносит златогузка (*Euproctis chrysorrhoea*), гусеницы которой имеют ядовитые волоски и предупреждающую окраску. Они часто оголяют кроны плодовых деревьев, обедая их листья. Зимуют гусеницы в больших гнездах, построенных из листьев, соединенных шелковыми нитями.

Семейство Совки (Noctuidae) самое большое, насчитывает более 30 тыс. видов бабочек. Их окраска скромная — обычно в серых или бурых тонах. Гусеницы голые, развиваются на растениях и в почве. Куколки развиваются обычно в почве. Среди совок много видов, приносящих вред лесному и сельскому хозяйству. В нашей стране особый ущерб причиняет озимая совка (*Agrotis segetum*), самки которой ведут ночной образ жизни и откладывают до 2 тыс. яиц. Гусеницы питаются разнообразными растениями, сильно вредят всходам зерновых культур (рис. 156).

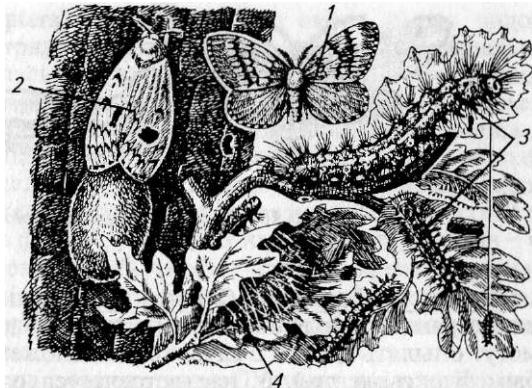


Рис. 155. Шелкопряд непарный (*Limantria dispar*):
1 — самец; 2 — самка, откладывающая яйца; 3 — гусеницы разного возраста; 4 — кокон с куколкой

К вредителям растений относятся также совка-гамма, капустная и сосновая совки.

Семейство Пяденицы (Geometridae). Это бабочки с нежными крыльями обычно белого цвета или с легким рисунком. Гусеницы передвигаются шагающими движениями, изгибая тело петлей (их движение напоминает измерение длины пядями — расстояние между указательным и большим пальцами руки). Гусеницы питаются многими видами растений. Так, сосновая пяденица (*Bupalus piniaria*) объедает хвою, окуливаясь в почве. Лиственным лесам вредят зимняя пяденица (*Operophtera brumata*), наносит ущерб и садовым деревьям. На крыжовнике и смородине обедает листья крыжовников пяденица (*Abraxas grossulariata*). Окуливание происходит на листьях и стеблях ягодных кустарников.

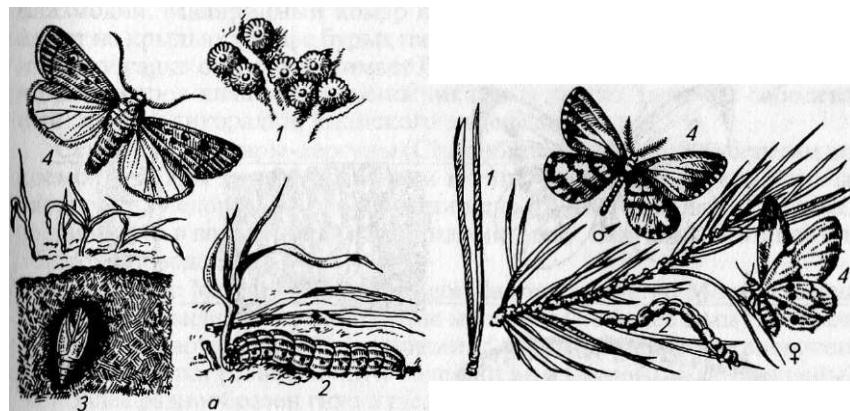


Рис. 156. Бабочки — вредители растений:
а — совка озимая (*Agrotis segetum*); б — пяденица сосновая (*Bupalus piniaria*)
1 — яйца на хвою; 2 — гусеница; 3 — куколка; 4 — имаго

Рис. 157. Сосновый бражник (*Sphinx pinastri*)—
а — имаго; б — гусеница

Семейство Бражники (Sphingidae). Бражники отлично летают. В нашей стране их около 50 видов. Они имеют длинный хоботок, поэтому могут опылять цветки с глубоко расположенными нектарниками (табак, флоксы и др.). У нас встречается сосновый бражник (*Sphinx pinastri*), гусеницы которого питаются хвоей сосны (рис. 157).

Подотряд Булавоусые (Rhopalocera). К этому подотряду относятся в основном бабочки, ведущие дневной образ жизни. Они ярко окрашены. Их крылья в период покоя складываются над спиной. Подотряд насчитывает много семейств, из которых рассмотрим наиболее часто встречающиеся.

Семейство Белянки (Pieridae). Бабочки обычно белой, реже желтой окраски. Взрослые насекомые опыляют растения, а их гусеницы питаются растениями. Гусеницы белянки капустной (*Pieris brassicae*), белянки репной (*Pieris rapae*) и брюквенницы (*Pieris napi*) наносят вред огородным культурам, развивааясь на крестоцветных растениях (рис. 158).

Бабочки, ведущие дневной образ жизни, по своей красоте и грациозности не имеют себе равных среди насекомых. Многие виды занесены в Красную книгу. Они являются важными опылителями, а гусеницы большинства из них развиваются на дикорастущих растениях, не нанося им существенного вреда.

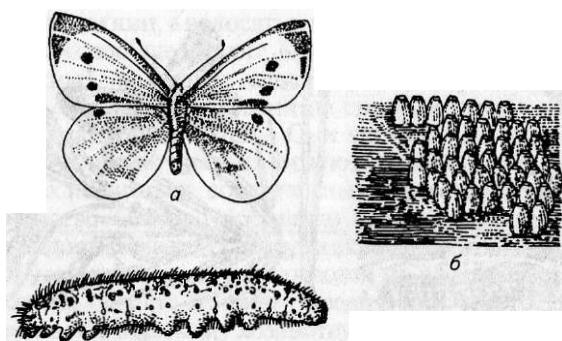


Рис. 158. Белянка капустная (*Pieris brassicae*):
а — бабочка; б — яйца; в — гусеница; г — куколка

Отряд Двукрылые (Diptera). Отряд включает около 80 тыс. видов. У представителей этого отряда сохранилась лишь первая пара крыльев, вторая же пара превратилась в жужжалыца. Жужжалыца — булавовидные придатки, которые расположены позади основания крыльев. Внутри жужжалец находится органы равновесия. Ротовые аппараты колючего или лижущего типа. Взрослые формы питаются только жидкой пищей различной природы. Среди них есть кровососы (комары, мошки, слепни, жигалки), опылители, которые питаются нектаром цветков (комары-долгоножки), а есть формы, у которых имаго не питаются.

Личинки двукрылых без ног и обычно без четко дифференцированной головы. Личинки развиваются в почве, воде, гниющих остатках, тканях животных и растений. У многих видов личинкам свойственно внекишечное пищеварение.

Отряд подразделяется на три подотряда: Длинноусые, или Комары (*Nematocera*), Короткоусые прямошовные (*Brachycera*—*Orthorrhapha*) и Короткоусые круглошовные (*Brachycera*—*Cyclorrhapha*), которые отличаются по форме усиков, жилкованию крыльев, ротовому аппарату и типу личинок и куколок.

Подотряд Длинноусые (*Nematocera*). Представители имеют длинные многочленниковые усики. Их личинки с головной капсулой. Куколка покрытого типа. К подотряду относится много семейств комаров и москитов (рис. 159).

Семейство Настоящие комары (Culicidae). Распространены повсеместно, но особенно многочисленны в регионах с влажным климатом. У комаров колюще-сосущий ротовой аппарат. Есть кровососущие формы (питаются кровью лишь самки). Личинки и куколки развиваются во влажной почве или в воде. Большие неприятности доставляет обыкновенный комар (*Culex pipiens*), опасен малярийный комар (*Anopheles maculipennis*), который является переносчиком малярийного плазмодия. Малярийный комар в отличие от обыкновенного комара имеет на крыльях четыре бурых пятна. Его ноги в два раза длиннее тела и при посадке он приподнимает брюшко вверх (рис. 160). Различные виды комаров являются переносчиками и других тяжелых заболеваний: желтой лихорадки, японского энцефалита и т. д.

Семейство Комары-дергуны (Chironomidae). Обитая по берегам водоемов, комары летают большими группами, издавая звуки. Ротовые части у них недоразвиты, живут они недолго и не питаются. Личинки развиваются в воде и носят название «мотыль». Они служат пищей для обитателей водоемов.

Семейство Мокрецы (Ceratopogonidae) и семейство Мошки (Simuliidae). Эти семейства представлены мелкими (не более 5 мм) кровососущими комариками, образующими с комарами армию кровососов (гнус). Личинки развиваются в воде или во влажной среде (мокрецы). Наиболее разнообразен гнус в таежной зоне. Мошки могут быть переносчиками опасных заболеваний (см. рис. 159).

Семейство Бабочницы (Psychodidae). К этому семейству относятся москиты — мелкие (не более 3 мм) кровососы, распространенные в

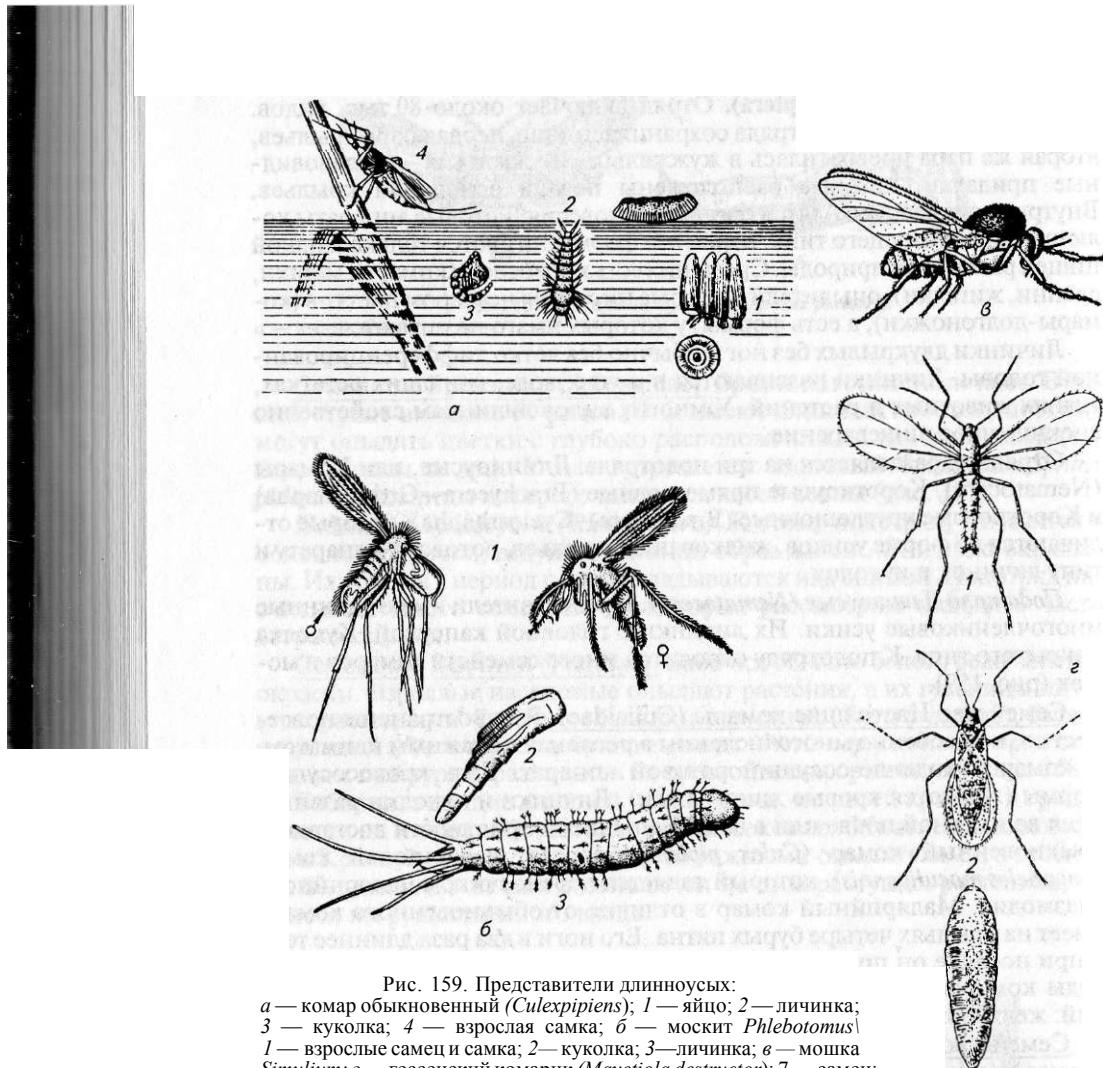


Рис. 159. Представители длинноусых:
 а — комар обыкновенный (*Culex pipiens*); 1 — яйцо; 2 — личинка;
 3 — куколка; 4 — взрослая самка; б — москит *Phlebotomus*:
 1 — взрослые самец и самка; 2 — куколка; 3 — личинка; в — мошка
Simulium; г — гессенский комарик (*Mayetiola destructor*); 7 — самец;
 2 — самка; 3 — личинка

южных регионах. Тело их покрыто волосками. Самцы питаются соками растений, а самки — кровью. Их личинки развиваются во влажных местообитаниях. Самки нападают на животных и человека. Могут переносить опасные заболевания, в том числе лейшманиозы.

Семейство Галлицы (Cecidomyiidae). Это обширная группа мелких комариков, которые в стадии имаго не питаются. Их личинки развиваются в тканях растений, вызывая образование галлов. Так, гессенский комарик (*Mayetiola destructor*) развивается в пазухах листьев пшеницы, стебли которой обламываются (см. рис. 159).

Рис. 160. Комары:
а — комар обыкновенный (*Culex pipiens*); б — комар малярийный (*Anopheles maculipennis*);
1 — имаго; 2 — личинка; 3 — куколка

Семейство Грибные комарики (Mycetophilidae). Личинки этих насекомых развиваются в грибах, а некоторые виды развиваются в гниющей древесине.

Подотряд Короткоусые прямошовные двукрылые (Brachycera—Orthorrhapha). Типичные мухи с короткими усиками. У личинок голова сильно редуцирована, куколки покрытого типа. При выходе имаго покровы куколки растрескиваются по Т-образной линии, что и отражено в их названии — прямошовные.

Семейство Слепни (Tabanidae). Это крупные кровососущие мухи (рис. 161), нападающие на животных и человека. Слепни могут переносить возбудителей заболеваний, в частности полиомиелита. Сосут кровь только самки, тогда как самцы питаются нектаром. Личинки развиваются в воде и во влажной почве.

Семейство Жужжалы (Bombyliidae) — мухи, которые питаются некотором цветков с глубокими нектарниками и являются опылителями рас-

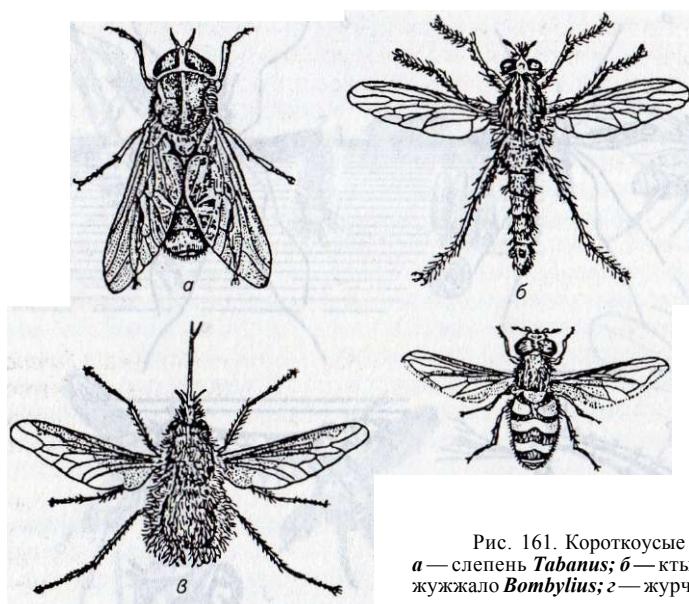


Рис. 161. Короткоусые двукрылые:
а — слепень *Tabanus*; б — ктырь *Philonicus*; в —
жуужало *Bombylius*; г — журчалка *Syrphidus*

тений. По внешнему виду жужжалы напоминают шмелей. Личинки паразитируют в гнездах пчел, в кубышках саранчи, гусеницах бабочек и т. п.

Среди короткоусых прямощовных двукрылых много хищных форм: Ктыри (сем. Asilidae), Толкунчики (сем. Empididae) и другие, которые охотятся на насекомых, в том числе вредоносных.

Подотряд Короткоусые круглошовные двукрылые (Brachycera-Cyclorrhapha). Усики у мух короткие. У личинок редуцирована головная капсула, но остались ротовые стилеты. Куколка свободная, заключена в пупарий (оболочка из несброшенных личинкой покровов). При выходе имаго куколка открывается по окружной линии, что отражено в названии подотряда — круглошовные.

Семейство Журчалки (Syrphidae) — множество видов мух-опылителей, у которых проявляется мимикрия с жалящими перепончатокрылыми. По форме тела похожи на пчел и других жалящих перепончатокрылых. Окраска преимущественно желтая с черным. Для журчалок типичен «стоячий» полет, когда муха зависает в воздухе. Личинки журчалок — хищники, питающиеся тлей и листоблошками.

Семейство Злаковые мухи (Chloropidae) — вредители злаков. Личинки этих мелких мух повреждают верх стебля, что может вызывать гибель растений. К опасным вредителям пшеницы относят шведскую муху.

Семейство Навозные мухи (Scatophagidae). У большинства навозных мух личинки развиваются в навозе, питаясь личинками других насекомых. Среди навозных мух встречаются виды, личинки которых вредят злаковым культурам.

Семейство Настоящие мухи (Muscidae). Это одно из самых многочисленных по числу видов (около 3 тыс.) семейство. Личинки настоящих мух развиваются в гниющих органических остатках, питаясь как сапрофаги или как хищники. Есть виды, личинки которых паразитируют на растениях, много синантропных видов: комнатная муха (*Musca domestica*), личинки которой развиваются в навозе и гниющих остатках (рис. 162). Для личинок комнатной мухи характерно внекишечное пищеварение. Выделяя пищеварительные соки на субстрат, они затем всасывают полупереваренную пищу. Учитывая высокую плотность заселения (в 1 дм³ может развиваться до 1,5 тыс. личинок), личинок выращивают на корм животным. Комнатные мухи распространяют яйца гельминтов и различные инфекции.

В жилище человека встречается малая комнатная муха (*Fannia canicularis*) и осенняя жигалка (*Stomoxys calcitrans*). Жигалок особенно много в местах содержания и выпаса скота, так как они сосут кровь у животных (см. рис. 162). К кровососам относится муха цепе (*Glossina palpalis*), переносящая трипаносому.

Много видов настоящих мух являются вредителями растений. Капустная муха (*Chortophila brassica*) на стадии личинки поражает рассаду капусты, личинки луковой мухи (*Chortophila antiqua*) развиваются в листьях лука, вызывая их засыхание. Встречаются мухи — вредители пшеницы, свеклы и других растений.

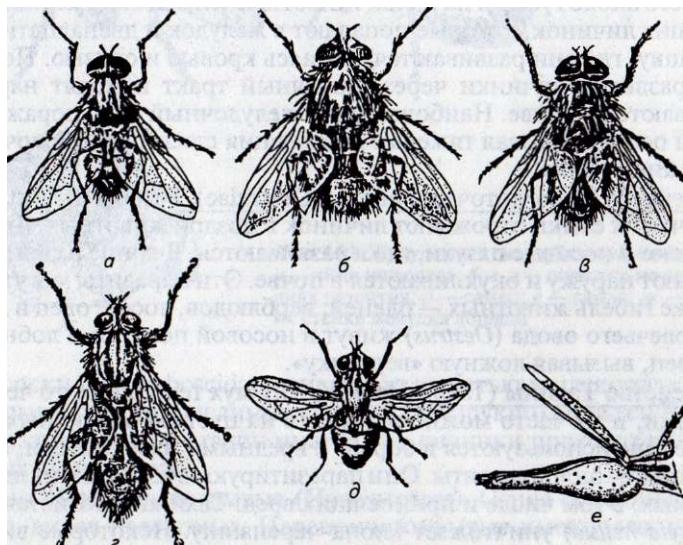


Рис. 162. Настоящие и падальные мухи:
а — комнатная муха (*Musca domestica*); б — синяя падальная муха (*Calliphora erythrocephala*); в — зеленая падальная муха (*Lucilia caesar*); г — серая мясная муха (*Sarcophaga camaria*); д — муха-жигалка (*Stomoxys calcitrans*); е — хоботок мухи

Семейство Серые мясные мухи (Sarcophagidae) распространено в регионах с умеренным климатом. В нашей стране часто встречается се-рая мясная муха (*Sarcophaga earnaria*), откладывающая на продукты питания живых личинок, развитие которых происходит в короткие сроки (см. рис. 162). В Средней Азии вольфартова муха (*Wohlfahrtia magnifica*) представляет опасность для животных и даже человека, отреждая личинок в глаза, уши или раны.

Семейство Подкожные оводы (Hypodermatidae). В отличие от слепней тело оводов покрыто волосками, а сами они имеют небольшие глаза. Семейство представлено паразитами, личинки которых развиваются под кожей животных. Имаго не питаются, так как их ротовые органы недоразвиты. Они откладывают яйца на шерстный покров животных. Вылупившиеся личинки внедряются в кожу, где и проходят свое развитие. Перед оккулированием личинки выходят из кожных вздутий через отверстия во внешнюю среду, падают на землю, где и оккуливаются. Подкожные оводы наносят значительный ущерб скотоводству и оленеводству.

Личинки бычьего овода (*Hypoderma bovis*) и овода крупного рогатого скота (*Hypoderma lineata*) паразитируют в теле коров и быков, скапливаясь в больших количествах под кожей животных на последней стадии своего развития.

Семейство Желудочные оводы (Gastrophilidae) — специализированная группа паразитических мух, личинки которых развиваются как эндопаразиты (рис. 163). Яйца самки откладывают на кожу животных, чаще всего около губ или на щеки. Животные слизывают и заглатывают вышедших личинок, которые попадают в желудок и двенадцатиперстную кишку, где они развиваются, питаясь кровью и слизью. По окончании развития личинки через кишечный тракт выходят наружу и оккуливаются в почве. Наиболее часто желудочный овод поражает лошадей и ослов, вызывая тяжелые воспаления слизистой оболочки желудка и кишечника.

Семейство Носоглоточные оводы (Oestridae). Эти оводы отличаются тем, что их самки отреждают личинок в ноздри животных. Личинки проникают в носовые пазухи, где и развиваются. В конце развития они выползают наружу и оккуливаются в почве. Эти паразиты могут вызывать даже гибель животных — оленей, верблюдов, лосей, овец и др. Личинки овечьего овода (*Oestrus*) живут в носовой полости и лобных пазухах овец, вызывая ложную «вертячку».

Семейство Тахины (Tachinidae). У этих мух тело покрыто черными волосками, и их часто можно встретить на цветках зонтичных растений. Тахины используются в борьбе с вредными насекомыми, так как их личинки — эндопаразиты. Они паразитируют в теле многих личинок насекомых, в том числе и приносящих вред. Тахина золотистая фазия (*Clytiomyia helluo*) уничтожает клопа-черепашку. Некоторые виды тахин завезли из других стран для борьбы с колорадским жуком, непарным шелкопрядом и другими опасными вредителями.

В подотряде короткоусых круглошовных двукрылых встречаются представители, ведущие исключительно паразитический образ жизни.

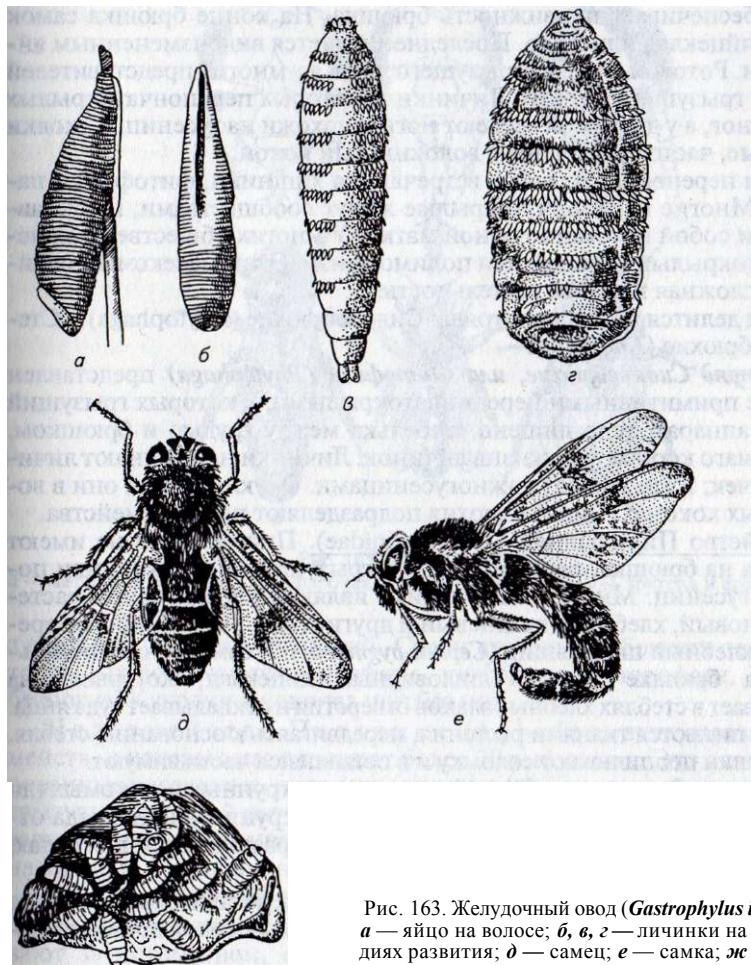


Рис. 163. Желудочный овод (*Gastrophylus intestinalis*):
а — яйцо на волосе; б, в, г — личинки на разных стадиях развития; д — самец; е — самка; ж — личинки на стенке желудка лошади

Кровососки (Hippoboscidae) живут под шерстью у млекопитающих и под перьями у птиц, пчелиные вши паразитируют на пчелах. Кровососки паразитируют на стадии имаго, а их личинки проходят развитие в половых путях самки.

Отряд Перепончатокрылые (Hymenoptera). Отряд объединяет более 300 тыс. видов насекомых. Перепончатокрылые имеют, как правило, две пары небольших прозрачных перепончатых крыльев, из которых задние меньше передних и сцеплены с ними, т. е. функционально перепончатокрылые являются двукрылыми. Часто грудь соединяется с брюшком тонким стебельком (второй и третий сегменты брюшка), ко-

торый обеспечивает подвижность брюшка. На конце брюшка самок имеется яйцеклад или жало. Последнее является видоизмененным яйцекладом. Ротовой аппарат грызущего типа, а у многих представителей является грызуще-сосущим. Личинки некоторых перепончатокрылых лишены ног, а у других они имеют ноги и похожи на гусеницы. Куколки свободные, часто заключены в волокнистый кокон.

Среди перепончатокрылых встречаются хищники, фитофаги и паразиты. Многие перепончатокрылые живут сообществами, представляющими собой потомство одной матки. У многих общественных перепончатокрылых наблюдается полиморфизм. Этим насекомым свойственна сложная нервная деятельность.

Отряд делится на два подотряда: Сидячебрюхие (*Phytophaga*) и Стебельчатобрюхие (*Arocryta*).

*Подотряд Сидячебрюхие, или Фитофаги (*Phytophaga*)* представлен наиболее примитивными перепончатокрылыми, у которых грызущий ротовой аппарат, тело лишено стебелька между грудью и брюшком. Жизнь имаго короче, чем жизнь личинок. Личинки напоминают личинок бабочек; их называют ложногусеницами. Окукливаются они в волокнистых коконах. Сидячебрюхих подразделяются на два семейства.

*Семейство Пилильщики (*Tenthredinidae*)*. Пилильщики не имеют стебелька на брюшке, ротовой аппарат грызущего типа. Личинки похожи на гусеницы. Многие пилильщики являются вредителями растений: сосновый, хлебный, вишневый и другие (рис. 164). Наиболее вредоносен хлебный пилильщик (*Sephis rugutaeus*). У самок этого пилильщика на брюшке имеется пиловидный яйцеклад, которым она пропиливает в стеблях хлебных злаков отверстия и откладывает туда яйца. Личинки питаются тканями растения, передвигаясь к основанию стебля. Там личинки подпиливают соломку и в оставшейся части зимуют.

*Семейство Рогохвосты (*Siricidae*)* включает крупных насекомых (до 4 см в длину). Самки рогохвостов с помощью крупного яйцеклада откладывают яйца под кору деревьев. Личинки развиваются в стволах, проделывая в них крупные ходы (см. рис. 164).

*Подотряд Стебельчатобрюхие (*Arocryta*)*. Представители подотряда имеют узкий стебелек между грудью и брюшком. Ротовой аппарат гры-

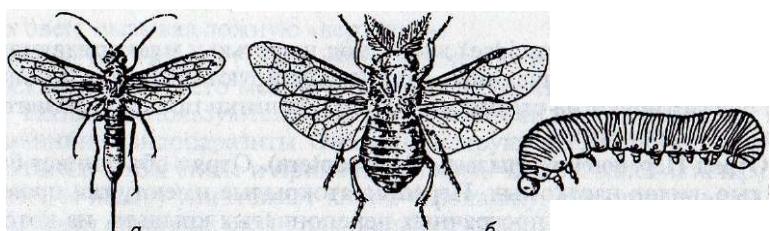


Рис. 164. Сидячебрюхие перепончатокрылые:
а — большой хвойный рогохвост (*Sirex gigas*); б — сосновый пилильщик (*Diprion pini*),
имаго и личинка

Рис. 165. Медоносная пчела:
а — матка и ее голова (1); б — рабочая пчела и ее голова (2); в — трутень и его голова (3)

зущий или грызуще-лижущий. Среди стебельчатобрюхих перепончатокрылых встречаются фитофаги, зоофаги и нектарофаги. Личинки безногие. Куколки в коконах или без них.

Надсемейство Пчелиные (*Apoidea*) включает шесть семейств, в которых насчитывают более 30 тыс. видов. Это специализированные насекомые, питающиеся пыльцой и нектаром цветков. У них грызуще-лижущий ротовой аппарат. Тело опущено, и первый членник задней лапки превращен в аппарат для сбора пыльцы — щеточка. Эти насекомые — опылители.

Пчелы ведут одиночный и общественный образ жизни. Одиночные пчелы представлены самками и самцами. Самки строят гнездо, ухаживают за потомством, собирают корм. Общественные пчелы живут семьями. Кроме самок и самцов в семье имеются рабочие особи — не-половорезные самки, выполняющие все функции в семье. Иными словами, в семье особи различаются морфологически и функционально (рис. 165).

Наибольший интерес представляет медоносная пчела (*Apis mellifera*), встречающаяся в диком состоянии и разводимая в пчеловодческих хозяйствах. На зимний период самцы (трутни) изгоняются из семей и погибают. Весной рабочие пчелы строят из воска соты с шестиугольными ячейками, в которые матка откладывает оплодотворенные и неоплодотворенные яйца. Из оплодотворенных яиц развиваются рабочие пчелы, а из неоплодотворенных — самцы (трутни). Пчеловодство — важнейшая отрасль сельского хозяйства, дающая ценную продукцию:

мед, воск, пергу, маточное молочко, прополис, яд и т. п. Пчелы используются как опылители сельскохозяйственных культур.

К общественным пчелам относятся шмели (сем. Bombidae). Они отличаются крупными размерами, их тело густо покрыто волосками. Зимуют только самки шмелей. Весной они строят гнездо с сотами и выводят первых рабочих особей, которые берут на себя все работы в гнезде, а матка занята лишь откладыванием яиц. К осени в семье появляются трутни и молодые самки. После спаривания трутни погибают, а оплодотворенные самки зимуют в укромных местах. Шмели выполняют важную работу по опылению дикорастущих растений, особенно клевера, и тепличных растений. Шмелей разводят в искусственных условиях. Для привлечения шмелей на полях устанавливают специальные гнезда.

Надсемейство Осообразные (Vespoidea). Осы — жалящие перепончатокрылые. Из нескольких семейств наиболее многообразны складчатокрылые осы (сем. Vespidae). К ним относится обыкновенная оса (*Vespa vespae*). Осы, как и пчелы, бывают одиночные и общественные. Обыкновенная оса строит гнездо из пережеванной древесины на деревьях, под крышей, под крышкой ульев и т. п. Из яиц развиваются личинки, которых осы выкармливают полупереваренной пищей. Осы — хищники, поедают также сладкие плоды.

К осообразным относятся роющие осы, охотящиеся на различных насекомых; эти осы парализуют своих жертв, затаскивают их в норки и скармливают личинкам (забота о потомстве).

Надсемейство Муравьи (Formicoidea) представлено одним семейством Formicidae, в котором более 5 тыс. видов. Это преимущественно хищники. У них грызущий ротовой аппарат, имеется брюшной стебелек. Питаются насекомыми и другими животными. Встречаются фитофаги и сапрофаги. Муравьи — общественные насекомые, для которых характерны сложные взаимоотношения. Живут они в муравейниках. Самцы имеют крылья. Самки откладывают яйца, а всю работу в гнезде выполняют бескрылые бесплодные самки — рабочие особи (рис. 166). В гнездах муравьев живут насекомые симбионты. Многие муравьи находятся в симбиозе с тлями, питаясь их сладкими выделениями и охраняя их.

В середине лета на муравейниках появляются крылатые самцы и более массивные самки. После брачного полета самки теряют крылья и строят новое гнездо — муравейник, где откладывают яйца, из которых развиваются рабочие особи.

Многих хищных муравьев разводят как истребителей вредных насекомых в лесах, ограждая и охраняя их муравейники. Весьма полезны в лесах рыжие лесные муравьи из рода *Formica* (*F. rufa* и *F. polyctena*). В жилище человека живут домовые муравьи (*Monomorium pharaonis*). Муравьи-жнецы (*Messor*) вредят зерновым культурам. Многие виды муравьев участвуют в процессах почвообразования.

Надсемейство Наездники (Ichneumonidae). Наездники — паразиты насекомых и пауков. Это стройные насекомые с брюшком, сидящим на длинном стебельке. У самок имеется длинный яйцеплад, с помощью которого они вводят яйца в тело жертвы (чаще

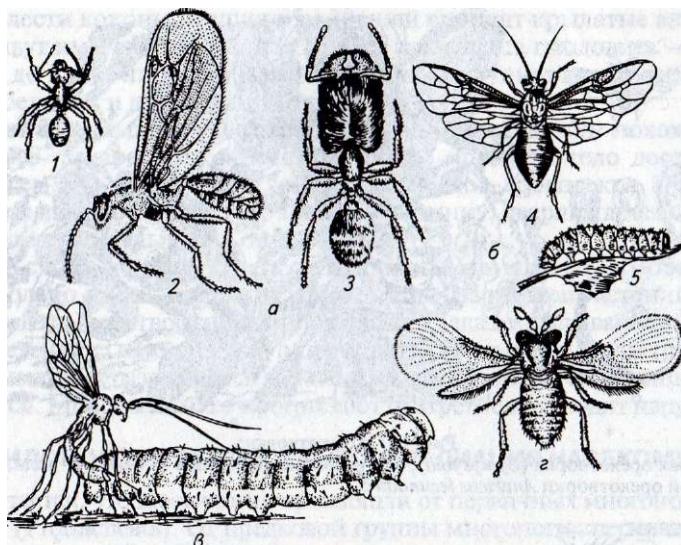


Рис. 166. Перепончатокрылые насекомые:
а — муравьи; 1 — рабочий; 2 — крылатый самец; 3 — солдат; б — крыжовниковый пилильщик; 4 — взрослая самка; 5 — личинка; 6 — наездник парализует гусеницу бабочки перед откладкой в нее яйца; г — яйцеед-трихограмма

всего это гусеницы бабочек). Личинки наездников паразитируют в теле насекомых (эндопаразиты), вызывая их гибель. Встречаются наездники эктопаразиты. У многих видов имаго не питаются. Одни наездники (мелкие по размерам) откладывают яйца в яйца других насекомых (яйцееды). Вышедшие из яиц личинки яйцеедов питаются тканями личинок хозяев. Среди них ряд видов рода Трихограмма (*Trichogramma*) и рода Теленомус (*Telenomus*) используются для биологической борьбы с вредными насекомыми. Яйцеед трихограмма паразитирует в яйцах различных бабочек, а Теленомус — в яйцах клопа вредной черепашки.

Более крупные наездники откладывают яйца в гусениц бабочек, ложногусениц пилильщиков или в коконы с куколками. Так, наездники рода *Apanteles* паразитируют в гусеницах непарного шелкопряда и бабочек-капустниц (см. рис. 166). Другие наездники паразитируют на взрослых насекомых. У некоторых наездников отмечена полиэмбриония: при развитии яйца зародыш делится, в результате чего получается от 2—10 до 2 тыс. личинок.

Среди наездников наиболее известны в нашей стране ихневмониды и бракониды. Из браконид славится апантелес. Этот малыш (белянковый мелкобрюх) разыскивает гусениц бабочки капустной белянки. Сев на гусеницу, наездник прокалывает эпидермис и откладывает в нее 15—35 яиц. Вышедшие из яиц личинки апантелеса питаются гемолимфой и жировым телом гусеницы. Через 8—12 дней личинки выходят и начи-

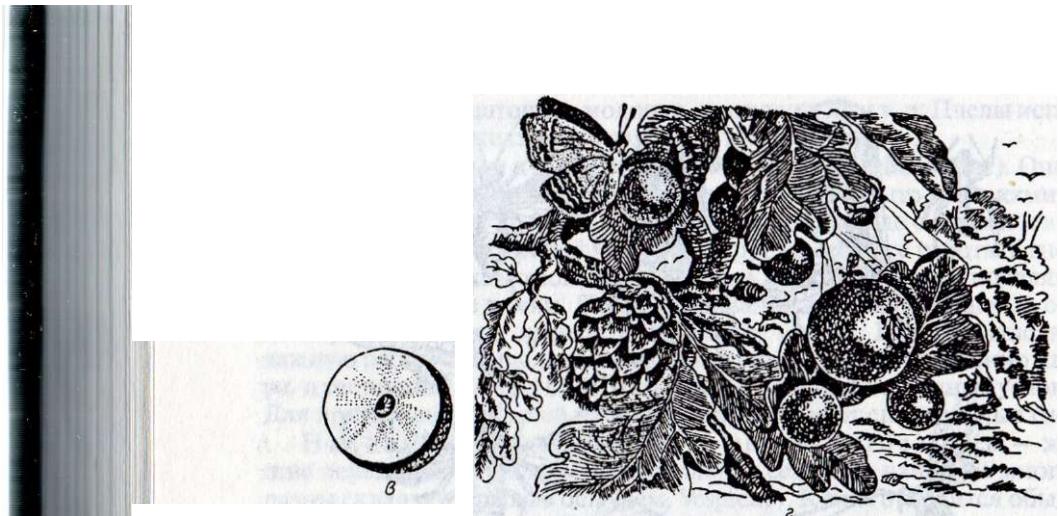


Рис. 167. Орехотворки:
а — дубовая орехотворка (*Cynips folii*); 6 — развитие личинки в побеге; 8 — галл; 2 — галл
шишковой орехотворки *Andricus secundatrix*

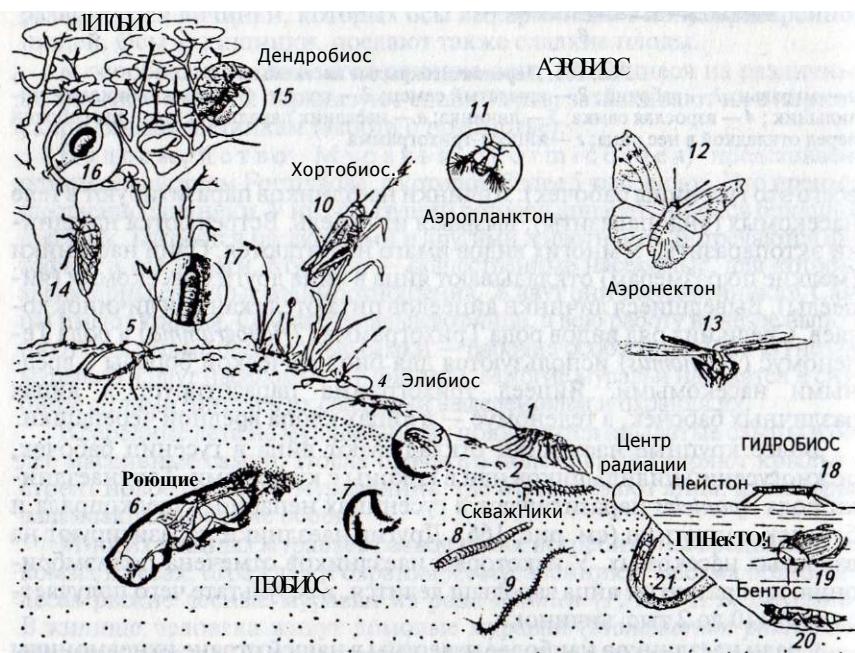


Рис. 168. Экологическая радиация трахейных — многоножек и насекомых:
1 — многоножка; 2 — многоножка (костянка); 3 — поверхностная коллембola; 4 — тара-
кан; 5 — жук (жукалица); 6 — медведка; 7 — почвенная коллембola; 8 — кивсяк; Р — мно-
гоНожка-геофил; 10 — саранча; 11 — жук-веерокрылка; 12 — бабочка; 13 — стрекоза;
14 — цикада; 15 — листоед; 16 — личинка орехотворки; 17 — личинка усача; 18 — водо-
мерки; 19 — плавунец; 20 — личинка стрекозы; 21 — личинка поденки

нают плести коконы, а спустя 5—10 дней выходят крылатые апантелесы. Живут имаго около месяца. Самка апантелеса плодовита — откладывает до 2 тыс. яиц. Вылупившиеся личинки уничтожают до 90 % гусениц белянок и других вредителей.

Один из самых крупных наездников — эфиальт. Он похож на гигантского комара: высокие ноги, четыре крыла, а тело достигает в длину 4 см. Эфиальтистrebляетличинокжуков-древесок. Ползая по коре, наездник безошибочно находит личинку, сверлит древесину яйцекладом и откладывает в личинку яйцо.

Надсемейство Орехотворки (Сунироидеа) объединяет несколько семейств мелких насекомых — паразитов растений и реже насекомых. Орехотворки с помощью яйцеклада откладывают яйца в листья растений. Ткани листьев разрастаются и образуют галлы (рис. 167), внутри которых развиваются личинки орехотворок. Затем личинки оккуливаются. Молодые имаго прогрызают отверстие и выходят наружу.

ФИЛОГЕНИЯ И ЭКОЛОГИЧЕСКАЯ РАДИАЦИЯ НАСЕКОМЫХ

Считается, что шестиногие произошли от первичных многоногих трахейных (Protracheata). От предковой группы многоногих первичных трахейных сначала обособились шестиногие сrudиментарными конечностями. Эволюция шестиногих развивалась двумя путями: к шестиногим со скрытым ротовым аппаратом и к многоядным формам. Открыточелостные развивались от почвенных форм к обитанию на поверхности почвы и на растениях. Лазанье по растительности способствовало развитию крыльев: сначала появились грудные кожные складки, затем нескладывающиеся крылья и, наконец, современные разнообразные крылья. С появлением крыльев насекомые активно освоили среду обитания на растениях, совершая активные воздушные миграции. Насекомые заняли практически все экологические ниши и образовали множество жизненных форм (рис. 168).

НАДТИП ВТОРИЧНОРОТОВЫЕ (Deuterostomia)

Вторичноротые — особая филогенетическая ветвь целомических животных, к которым относится несколько типов: Иглокожие (Echinodermata), Полухордовые (Hemichordata) и Хордовые (Chordata). Хордовые достигли наивысшего развития среди животного мира. Вторичноротые животные характеризуются общими чертами организации, которые отличают их от первичноротых животных (кольчатые черви, моллюски, членистоногие).

Кожа у вторичноротых животных состоит из двух слоев: эктодермального эпителия и соединительнотканного слоя — кутицы мезодермального происхождения. Скелет мезодермального происхождения образуется в соединительнотканном слое кожи и пропитан известью. В процессе эмбриогенеза у вторичноротых рот образуется вторично, а на месте первичного рта (blastopora) формируется анальное отверстие (анус).

ТИП ИГЛОКОЖИЕ (*Echinodermata*)

Общая характеристика. Представители этого типа, число которых составляет около 6 тыс. видов, населяют океаны и моря. Это вторичнонеполостные животные, имеющие во взрослом состоянии радиальную симметрию тела. У большинства видов органы расположены по пяти радиусам, но у части форм число лучей иное. Свободноплавающие личинки иглокожих имеют двустороннюю симметрию тела. Вторичная полость заполнена полостной жидкостью.

Иглокожие преимущественно донные животные, способные к медленному перемещению по субстрату, реже они прикрепляются к дну посредством особого стебелька и ведут прикрепленный образ жизни (рис. 169). Некоторые иглокожие служат пищей для морских обитателей, в том числе позвоночных, встречаются среди иглокожих промысловые виды. Размеры иглокожих колеблются от нескольких миллиметров до 2 м (голотурии).

Строение и жизненные отправления. Форма тела иглокожих весьма разнообразна, что отражают их названия: морские огурцы, морские ежи, морские звезды и лилии. В соединительнотканном слое кожи иглокожих развивается скелет из известковых пластинок с торчащими на поверхности тела шипами, иглами и т. п. Скелет служит защитой и опорой для организма. Вторичная полость тела (целом), где расположены внутренние органы, заполнена целомической жидкостью и выполняет транспортные и опорные функции. За счет целома образуется амбулакральная система органов движения.

Покровы состоят из двух слоев: наружного ресничного эпителия и лежащего под ним внутреннего соединительнотканного. В наружном слое кожи имеются пигментные, железистые и чувствительные клетки. Колебания ресничек обеспечивают токи воды вдоль тела, которые поддерживают кожное дыхание, очищают тело и приносят пищевые частицы к ротовому отверстию. Среди железистых клеток могут быть и клетки, выделяющие ядовитую слизь. В соединительнотканном слое кожи образуется внутренний известковый скелет мезодермального происхождения. У морских звезд скелет имеет вид известковых пластинок, расположенных рядами; у морских ежей скелет представлен известковым панцирем из радиально расположенных парных рядов пластинок; сплошной панцирь имеется у некоторых звезд и морских лилий. К скелетным образованиям относится мадрепоровая пластинка с мелкими порами, ведущими в амбулакральную (водно-сосудистую) систему.

Нервная система примитивна и имеет радиальное строение. От окологлоточного нервного кольца отходят радиальные нервные тяжи, число которых соответствует числу лучей. Как у всех животных, ведущих малоподвижный образ жизни, органы чувств развиты слабо: у некоторых

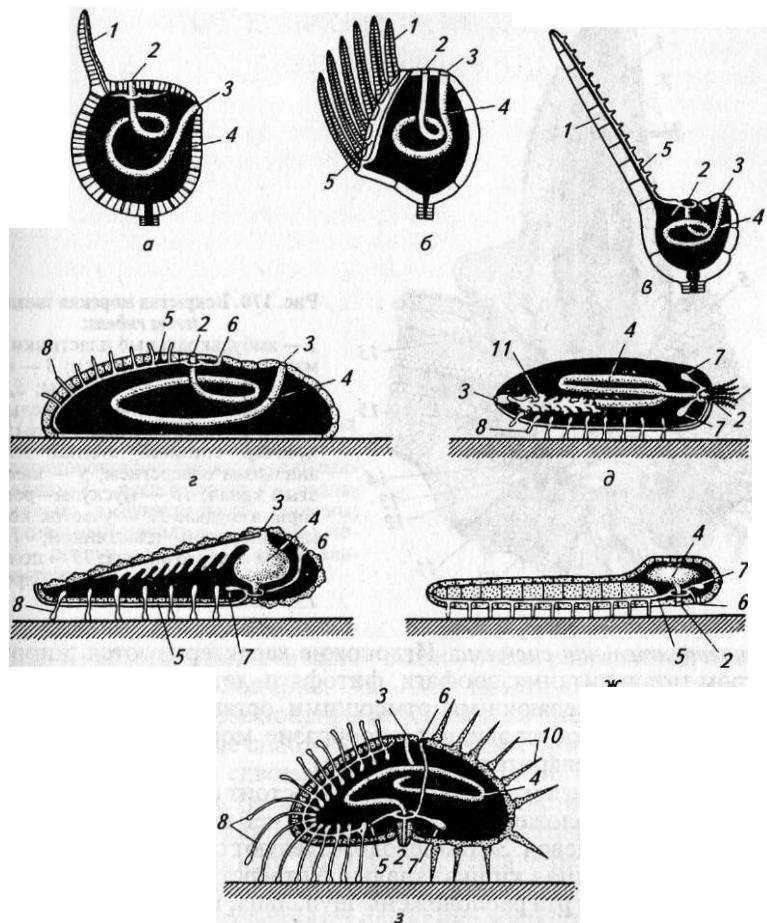


Рис. 169. Схема строения иглокожих различных классов:
 а — цистоиды; б — blaстоиды; в — морские лилии; г — эдиоастериоиды; д — голотурии;
 « — морские звезды; ж — оphiуры; з — морские ежи; / — руки; 2 — рот; 3 — анус; 4 — кишечник;
 5 — амбулакральная система; 6 — мадрепоровая пластинка; 7 — поливевые пузыри;
 8 — амбулакральные ножки; 9 — щупальца; 10 — иглы; 11 — водное легкое

морских звезд на лучах находятся примитивные глазки, а у морских ежей они располагаются на поверхности тела. Имеются органы осознания.

Мускулатура. У голотурий хорошо развит кожно-мускульный мешок, у других представителей, имеющих развитый известковый скелет, мускулатура развита слабее. Интересными скелетными производными иглокожих являются *педицеллярии* (щипчики), с помощью которых они очищают тело. Особенно хорошо развиты щипчики у морских звезд и морских ежей.

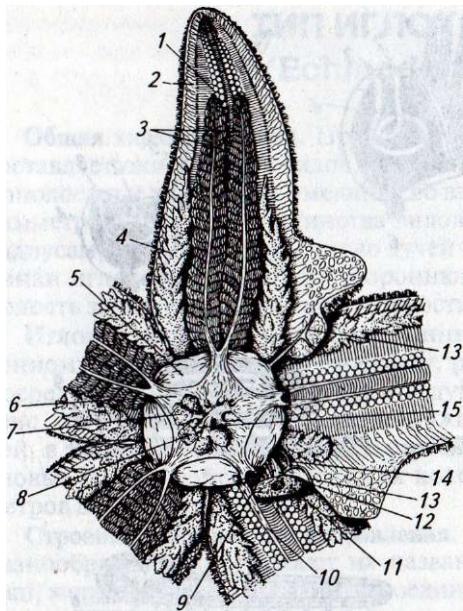


Рис. 170. Вскрытая морская звезда *Asterias rubens*:

1 — амбулакральные пластинки; 2 — маргинальные пластинки; 3 — печеночные мешки; 4 — гонады; 5, 6 — оральный и аборальный отделы желудка; 7 — ректальные железы; 8 — кусочек спинной стенки тела с анальным отверстием; 9 — каменистый канал; 10 — мускулы—ретракторы желудка; 11 — участок кожи с мадрепоровой пластинкой; 12 — стенка осевого синуса; 13 — половой столон; 14 — половой проток; 15 — задняя кишка

Пищеварительная система. Иглокожие характеризуются широким спектром типов питания: зоофаги, фитофаги, детритофаги и сестонофаги (питаются оседающими отмершими организмами и частично планктоном). Это обусловило разнообразие морфофункциональных особенностей пищеварительной системы.

У большинства иглокожих кишечник состоит из трех отделов. Ротовое отверстие расположено посередине нижней поверхности тела и ведет в короткий пищевод. За пищеводом следуют средняя кишка (желудок) и короткая задняя кишка, анальное отверстие имеется не всегда. У голотурий вокруг рта расположены щупальца, на которые налипают частицы пищи, отправляемые в рот. У некоторых ежей частицы пищи направляются в рост ресничками эпителиальных клеток, покрывающих специальные бороздки.

Особенностью пищеварения у иглокожих является преобладание внутриклеточного пищеварения над полостным: под влиянием пищеварительных соков, выделяемых стенкой желудка, пища распадается на мелкие частицы (морская звезда), которые перевариваются внутриклеточно в хорошо развитых печеночных придатках (рис. 170). Непереваренные скелетные остатки у иглокожих выбрасываются из желудка преимущественно через рот.

Амбулакральная система предназначена в основном для передвижения иглокожих и уникальна среди животного мира по своему строению. Система состоит из околосотового канала и отходящих от него радиальных каналов. От радиальных каналов отходят боковые канальцы,

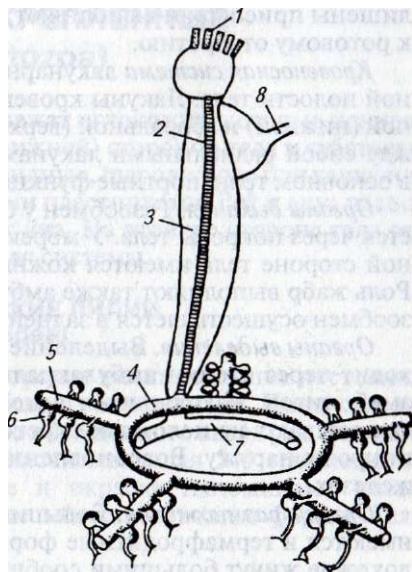


Рис. 171. Амбулакральная система морской звезды:

1 — мадрепоровая пластинка; 2 — каменистый канал; 3 — осевой синус; 4 — оральное кольцо амбулакральной системы; 5 — радиальный канал; 6 — ампулы ножек; 7 — оральное кольцо псевдогемальной системы; 8 — половой столон

каждый из которых заканчивается полой, очень растяжимой и мускулистой ножкой с ампулой (рис. 171). Ампулы находятся в полости тела, а ножки проходят сквозь покровы иглокожих во внешнюю среду. Ножки на свободном конце снабжены маленькими присосками. При этом ножки располагаются сдвоенными рядами в специальных радиальных бороздках. У большинства иглокожих от кольцевого канала отходит непарный каменистый канал, который открывается наружу пористой мадрепоровой пластинкой. Вода фильтруется через поры мадрепоровой пластинки и поступает внутрь через каменистый канал, ресничный эпителий которого обеспечивает движение жидкости внутри тела. Через мадрепоровую пластинку происходит также и регуляция полостного давления.

Амбулакральная система заполнена полостной жидкостью, состав которой близок к морской воде. Жидкость вгоняется в радиальный канал луча (морская звезда), расположенного по направлению движения звезды, и поступает в ампулы. При сокращении ампул жидкость нагнетается в ножки, которые под давлением жидкости вытягиваются по направлению движения и присасываются к субстрату с помощью присосок, которыми заканчиваются ножки. Когда мускулатура ножек сокращается, жидкость возвращается в ампулы, ножки укорачиваются и подтягивают животное вперед. Так перемещаются ежи, морские звезды и голотурии. Скорость передвижения составляет несколько сантиметров в 1 мин. У морских лилий и офиур ножки

лишены присосок и выполняют функции дыхания и передачи пищи к ротовому отверстию.

Кровеносная система лакунарного типа. Лакуны — остатки первичной полости тела. Лакуны кровеносной системы находятся на оральной (нижней) и аборальной (верхней) сторонах тела и сообщаются между собой радиальными лакунами. Кровеносная система выполняет в основном транспортные функции.

Органы дыхания. Газообмен у большинства иглокожих осуществляется через покровы тела. У морских звезд и морских ежей на аборальной стороне тела имеются кожные жабры в виде выпячиваний тела. Роль жабр выполняют также амбулакральные ножки. У голотурий газообмен осуществляется в задней кишке.

Органы выделения. Выделение конечных продуктов обмена происходит через стенки амбулакральной системы, а также с помощью амебоцитов, находящихся в целомической жидкости. Амебоциты захватывают из целома экскреторные вещества и выводят их через покровы наружу. Возобновление амебоцитов происходит в особых железах.

Органы размножения. Большинство иглокожих раздельнополы, но имеются и гермафродитные формы. Оплодотворение наружное. Иглокожие живут большими сообществами и выпускают половые продукты наружу, где и происходит их слияние. Иногда оплодотворенные яйца развиваются в специальных выводковых камерах материнского организма (живородящие виды). Развитие иглокожих происходит со сложным метаморфозом. Из яиц выходят планктонные двустороннесимметричные личинки, которые проходят через несколько личиночных фаз, характеризующихся кардинальными изменениями симметрии, строения, формы и внутренней организации (катастрофический метаморфоз).

Некоторые представители морских звезд, голотурий и офиур размножаются бесполым путем: материнский организм распадается на отдельные части, из которых развиваются дочерние особи.

Многие иглокожие обладают способностью к регенерации частей тела. Из отчлененного луча звезды с частью диска может восстановиться все тело. Поэтому у иглокожих наблюдают явление аутотомии — самокалечения. При нападении хищников звезды и офиуры могут обламывать свои лучи, и из этих обломков в дальнейшем происходит восстановление целого организма.

Тип Иглокожие (*Echinodermata*) подразделяется на два подтипа, включающие пять современных классов:

- Подтип Прикрепленные (*Pelmatozoa*)
 - Класс Морские лилии (*Crinoidea*)
- Подтип Подвижные (*Eleutherozoa*)
 - Класс Морские звезды (*Asteroidea*)
 - Класс Офиуры (*Ophiuroidea*)
 - Класс Морские ежи (*Echinoidea*)
 - Класс Голотурии (*Holothuroidea*)

ПОДТИП ПРИКРЕПЛЕННЫЕ (*Pelmatozoa*)

К подтипу Пельматозои принадлежат иглокожие, которые прикрепляются стеблем или аборальной (нижней) стороной тела к субстрату. У этих животных тело имеет мешковидную, шаровидную или чашевидную форму. Оно покрыто скелетными пластинками. Рот и анус расположены на стороне, обращенной от дна. На этой же стороне тела открываются амбулакральная и половая системы.

КЛАСС МОРСКИЕ ЛИЛИИ (*Crinoidea*)

Представители этого класса — морские лилии, ведущие постоянно или временно прикрепленный к субстрату образ жизни. Пищу они захватывают из толщи воды с помощью обращенного вверх рта. Это наиболее древняя группа иглокожих, насчитывающая 540 современных видов. Морские лилии по форме и окраске напоминают цветы (рис. 172). Встречаются стебельчатые и бесстебельчатые формы. Тело

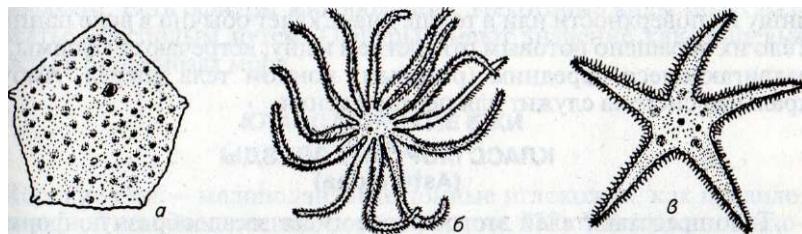


Рис. 172. Иглокожие:
a, б, в — морские звезды; *г* — змеевостка, или оphiура; *д* — морской еж; *е* — морской огурец, или голотурия; *ж* — морская лилия

их обращено оральной стороной вверх, на этой стороне расположены ротовое и анальное отверстия. Мадрепоровой пластинки нет, ее заменяет пористость стенок тела. Амбулакральные ножки лишены присосок и выполняют функции дыхания, осязания и передачи пищи ко рту. Большинство лилий десятирукие — с пятью раздвоенными у самого основания лучами, отходящими от тела (чашечки). Известковые членики мускулистого стебелька подвижны, нижние членики снабжены специальными усиками, служащими для прикрепления лилии к субстрату. Подвижные лучи-руки имеют побочные веточки — пиннулы. По верхней стороне лучей проходит амбулакральная бороздка, усаженная амбулакральными ножками без присосок.

В наших северных морях типична гелиометра (*Heliometra gracialis*) — бесстебельчатая лилия с лучами до 35 см. Лилии — очистители морской воды от органических загрязнений.

ПОДТИП ПОДВИЖНЫЕ (*Eleutherozoa*)

К этому подтипу относят иглокожих звездообразной, шаровидной или червеобразной формы, передвигающихся по дну и добывающих пищу на поверхности или в толще дна. Скелет обычно в виде панциря. Тело их обращено ротовым отверстием к дну; встречаются формы, передвигающиеся передним (ротовым) концом тела вперед. Амбулакральная система служит для передвижения.

КЛАСС МОРСКИЕ ЗВЕЗДЫ (*Asteroidea*)

Тело представителей этого класса имеет звездообразную форму, а число лучей чаще всего бывает пять, но иногда достигает 50. Размеры варьируют от 1 до 70 см. Морские звезды окрашены в разнообразные цвета. Их тело может быть уплощенным, выпуклым и даже без лучей (см. рис. 172). Известно около 1,7 тыс. видов морских звезд.

Полость тела и внутренние органы заходят в лучи. Движутся эти животные с помощью амбулакральных ножек с присосками. Ротовое отверстие расположено в центре диска с оральной стороны тела. Рот через короткий пищевод сообщается с большим складчатым желудком. От желудка в целомы лучей отходят пять пар длинных слепых выпячиваний — печеночных мешков. Анальное отверстие и мадрепоровая пластиинка находятся на верхней (аборальной) стороне тела (см. рис. 170).

Подавляющее большинство морских звезд — хищники, питающиеся двустворчатыми моллюсками, ракообразными, коралловыми полипами и т. п. Одни звезды заглатывают добычу целиком в желудок, для других свойственно внешнее пищеварение. В последнем случае они раздвигают лучами створки моллюска, выворачивают через рот наружу желудок, охватывают им жертву и переваривают ее. В морских биоценозах звезды являются одним из звеньев пищевых цепей; они могут на-

носить ущерб мидиевым и устричным хозяйствам. Морские звезды живут на глубине до 1 тыс. м, во время отливов могут оставаться по несколько часов вне воды. Очень чувствительны к солености воды, что объясняет их отсутствие в Черном и Балтийском морях.

КЛАСС ОФИУРЫ, ИЛИ ЗМЕЕХВОСТКИ (Ophiuroidea)

По своей организации и внешнему виду офиуры близки к морским звездам (см. рис. 172), но их лучи узкие, длинные и подвижные; с их помощью офиуры передвигаются. В лучи не заходят внутренние органы и полость тела. Лучи резко обособлены от диска и не переходят в него постепенно, как у морских звезд. Лишенные присосок амбулакральные ножки служат для дыхания и осаждания. Задняя кишечная и анальное отверстие отсутствуют. Мадрепоровая пластинка и ротовое отверстие расположены на оральной стороне тела.

Питаются офиуры различными мелкими животными, донным детритом, а отдельные виды исключительно водорослями. Известно более 1,5 тыс. видов офиур. В наших северных и дальневосточных морях часто встречаются горгоны (*Gorgonocephalus*), у которых диск тела достигает в диаметре 10 см, а лучи — 50 см в длину. Развитие происходит с метаморфозом. Есть офиуры живородящие. Некоторые виды могут размножаться бесполым путем. Офиуры имеют значение как пищевые объекты в биоценозах моря.

КЛАСС МОРСКИЕ ЕЖИ (Echinoidea)

Морские ежи — малоподвижные донные иглокожие, как правило, шаровидной или уплощенной формы (см. рис. 172). У большинства хорошо развит скелет в виде сплошного панциря из плотно соединенных известковых пластинок. Ежи покрыты многочисленными иглами, которые подвижно прикреплены к телу. Амбулакральные ножки с присосками; некоторые виды передвигаются на иглах. Известно около 800 видов морских ежей. Большинство ежей питаются растительной пищей, но есть много и зоофагов. Рот, расположенный на нижней стороне тела, вооружен особым жевательным аппаратом с пятью выступающими зубами — аристотелев фонарь (рис. 173). Кишечник заканчивается анальным отверстием на вершине тела. Иглы, покрывающие тело ежей, наряду с защитными функциями иногда принимают участие в передвижении (как ходули). Среди обыкновенных игл разбросаны иглы-щипчики (педицеллярии), предназначенные для очистки тела от экскрементов. Некоторые щипчики выполняют защитную роль, так как имеют ядовитые железы.

У многих ежей имеются кожные жабры, расположенные на околоворотовой площадке. Оплодотворение наружное. Развитие происходит со сложным метаморфозом. Ежами питаются чайки, каланы, рыбы, морские звезды, крабы. Промысел ежей ведется в дальневосточных морях.

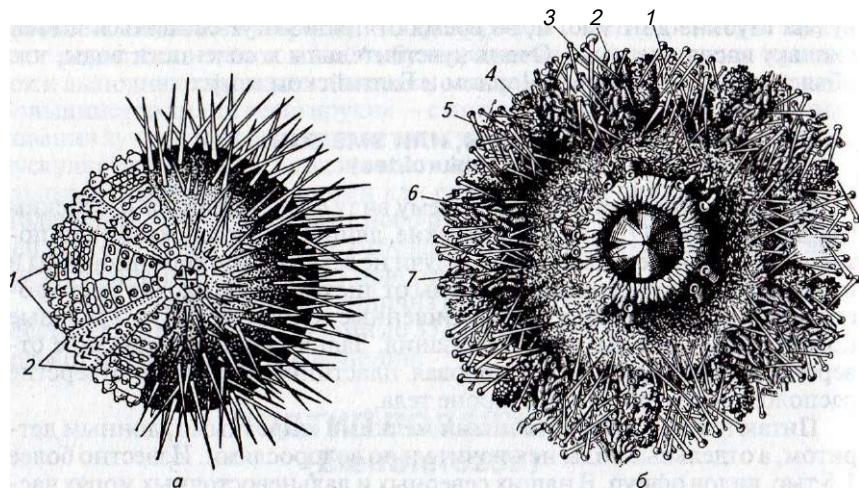


Рис. 173. Морской еж:
а — с аборального полюса, наполовину очищенный от игл; 1 — интерамбулакральные ряды пластинок; 2 — амбулакральные ряды пластинок; б — перистомальное поле; 1 — кожа; 2 — амбулакральные ножки; 3 — иглы; 4 — жабры; 5 — перистомальные амбулакральные ножки; 6 — околоротовой валик; 7 — ротовое отверстие с выступающими зубами аристотелева фонаря

КЛАСС ГОЛОТУРИИ (*Holothuroidea*)

Голотурии, или морские огурцы, или морские кубышки, имеют мешковидное тело, у некоторых видов оно червеобразной формы с венчиком ветвистых щупалец вокруг рта, расположенного на переднем конце. Вдоль тела тянутся пять полос амбулакральных ножек с присосками (см. рис. 172). В основном это донные животные, но встречаются роющие и плавающие виды. Питаются голотурии органикой и мелкими животными. Насчитывают около 900 видов голотурий. В их коже разбросаны мелкие известковые тельца. Дышат голотурии с помощью особых двух водяных легких, лежащих в полости тела по бокам кишечника. Задние концы легких сливаются и открываются в клоаку, из которой в них периодически нагнетается и отсасывается морская вода. Кислород через тонкие стенки легких поступает в полость тела. Через стенки легких из полости тела наружу выделяются амебоциты, загруженные конечными продуктами метаболизма.

Размеры голотурий колеблются от нескольких миллиметров до 1 м. Плавающие формы имеют небольшое студенистое тело, питаются планктоном. При опасности голотурии выбрасывают из задней кишки особые липкие нити или даже все содержимое полости тела. Секрет, из которого образуются липкие нити, поступает по протоку в кишечник

из киовьевой железы. Некоторые виды при нападении на них хищника отбрасывают заднюю часть своего тела. Недостающие части тела у голотурий регенерируют.

Голотурии раздельнополы, но встречаются гермафродиты, которые функционируют и как самцы, и как самки. Развитие с метаморфозом. Личинки голотурий выполняют расселительную функцию. Голотурии служат пищей для крупных обитателей морей и океанов, являются объектом промысла (40 видов). Основным объектом промысла в нашей стране является дальневосточный трепанг (*Stichopus japonicus*).

ФИЛОГЕНИЯ ИГЛОКОЖИХ

О происхождении иглокожих можно судить по строению их личинок, которые имеют двустороннюю симметрию, вторичный рот, двухслойную кожу, три пары целомических мешков и т. п. Все это дает основание считать предками иглокожих примитивных вторичноротовых животных.

Можно предположить, что первые иглокожие были малоподвижными животными с горизонтальной осью тела и прямым кишечником. В процессе эволюции обособились две группы иглокожих: подвижные и неподвижные.

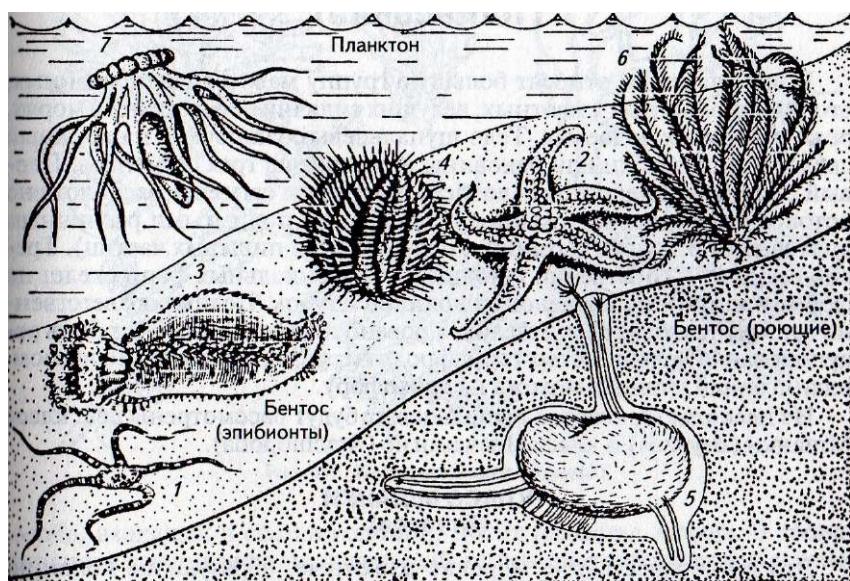


Рис. 174. Экологическая радиация иглокожих:
1 — офиура; 2 — морская звезда; 3 — голотурия; 4 — морской еж; 5 — неправильный морской еж — сердцевидка; 6 — морская лилия; 7 — плавающая голотурия

У неподвижных иглокожих в связи с типом питания произошло перемещение рта и ануса на верхнюю поверхность тела. Амбулакральные ножки вокруг рта служили для перемещения пищи ко рту. Подвижные формы развивались по пути специализации к питанию донной пищей, что привело к смешению рта на нижнюю поверхность тела, а анального отверстия — на верхнюю поверхность тела.

Голотурии сохранили горизонтальное положение оси тела, но они приобрели околоворотовые щупальца, выбирающие пищу со дна. Наиболее эволюционно прогрессивными оказались подвижные иглокожие, представляющие в настоящее время четыре класса.

В морских биоценозах иглокожие населяют три яруса: поверхность дна, толщу грунта и реже встречаются пелагические формы (рис. 174). К неподвижным и слабоподвижным относятся морские лилии, офиуры и древовиднощупальцевые голотурии. Подвижные формы, населяющие поверхность дна, образуют несколько жизненных форм. Морские ежи и некоторые звезды, лазающие по скалам и камням — фитофаги. Большинство звезд и офиуры — зоофаги. К детритофагам относятся некоторые ежи и голотурии. Есть и роющие иглокожие: червеобразные голотурии, сердцевидные ежи. Очень немногие обитают в пелагии: зонтикообразные и змеевидные голотурии.

ТИП ЩУПАЛЬЦЕВЫЕ (*Tentaculata*)

К щупальцевым относят большую группу малосегментированных, вторичнополостных животных, ведущих сидячий образ жизни в морях, реже в пресных водоемах. Тело щупальцевых заключено в наружный скелет в виде трубки или раковины и состоит из трех сегментов. Первый сегмент — предротовая лопасть, на втором сегменте расположено ротовое отверстие, окруженное щупальцами, покрытыми ресничным эпителием (функции дыхания и перемещения пищевых частиц). Третий сегмент — собственно туловище. У колониальных форм скелет по форме напоминает коралловых полипов и губок. Целом соответственно состоит также из трех отделов. Кровеносная система может быть рециклирована. Большинство гермафродиты, развитие с метаморфозом, планктонная личинка похожа на трохофору.

Из нескольких классов щупальцевых будут рассмотрены два: класс Мшанки (*Bryozoa*) и класс Плеченогие (*Brachiopoda*).

КЛАСС МШАНКИ (*Bryozoa*)

Это преимущественно морские животные, образующие колонии, которые имеют вид веточек, наростов, корочек, пучка листьев и т. п. (рис. 175). Известно около 4 тыс. видов. Обычно длина особи не превышает 1 см, а площадь колонии — несколько сантиметров. Каждая

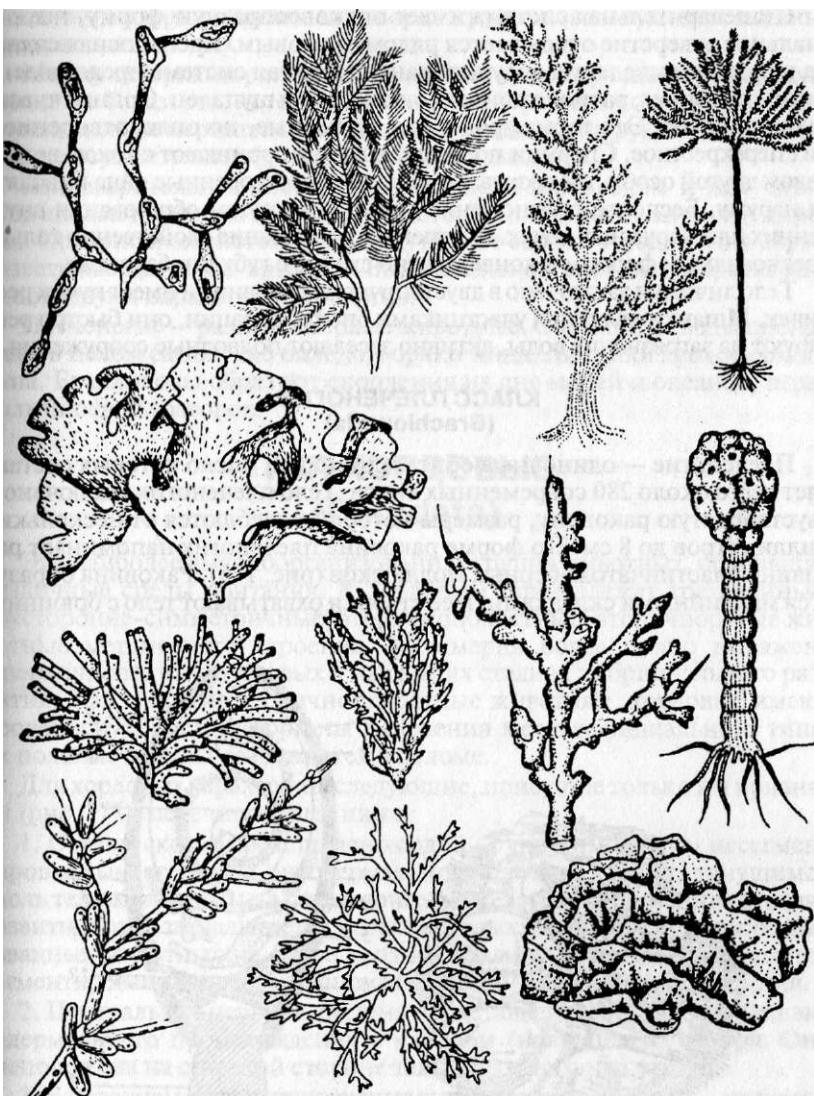


Рис. 175. Колонии мшанок (Bryozoa)

особь, входящая в состав колонии, носит название зоида. В зоиде различают передний конец, который лишен плотной кутикулы и легко втягивается в заднюю часть, имеющую форму чашечки и утолщенную кутикулу. Передний конец несет рот, окруженный щупальцами с мерцательным эпителием.

Пищеварительная система имеет подковообразную форму, так как анальное отверстие открывается рядом с ротовым. Кровеносная система и органы выделения редуцированы. Нервная система представлена одним ганглием, расположенным у основания щупалец. Органы чувств развиты слабо. Это гермафродитные животные, но оплодотворение у них перекрестное. Спермии попадают в воду и проникают с током воды в целом другой особи, где созревают яйца. Оплодотворенные яйца выводятся наружу. Бесполое размножение происходит путем образования внутренних или наружных почек. Внутреннее почкование свойственно только пресноводным формам и конвергентно сходно с губками бадягами.

Тело личинки заключено в двустворчатую раковину и имеет пучок ресничек. Мшанки являются участниками биофильтрации, они быстро реагируют на загрязнение воды, активно заселяют подводные сооружения.

КЛАСС ПЛЕЧЕНОГИЕ (*Brachiopoda*)

Плеченогие — одиночные обитатели морей, число которых составляет всего около 280 современных видов. Тело плеченогих заключено в двустворчатую раковину, размеры которой колеблются от нескольких миллиметров до 8 см. По форме раковина плеченогих напоминает раковину пластинчатожаберных моллюсков (рис. 176). Раковина образуется мантийными складками, и ее створки охватывают тело с брюшной

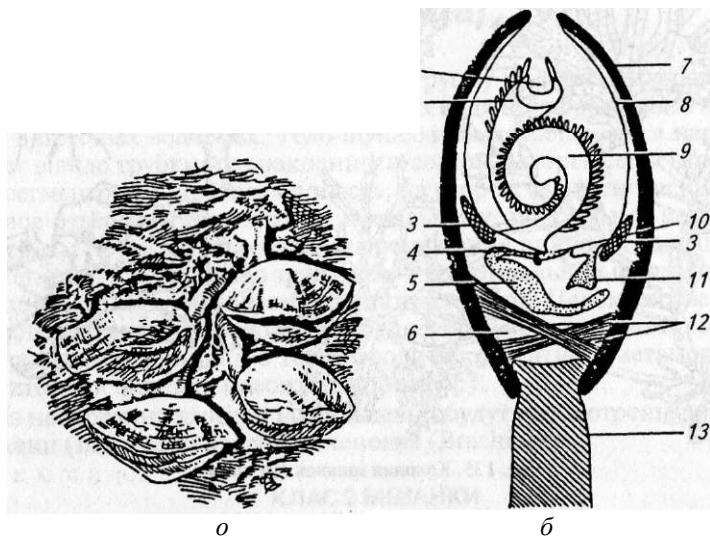


Рис. 176. Плеченогие:
а — общий вид; б — схема внутреннего строения; 1 — пищевой желобок; 2 — боковая рука лохофора; 3 — гонада; 4 — нервное кольцо; 5 — желудок; 6 — полость тела; 7 — раковина; 8 — мантия; 9 — центральная рука лохофора; 10 — рот; 11 — нефридий; 12 — мускулы; 13 — стебелек

и спинной сторон. Брюшная створка крупнее и выпуклее. Раковина прикрепляется к дну.

Между створками раковины находятся две спиральные руки, отходящие по бокам рта, которые служат для сбора пищи с помощью множества мелких мерцательных щупалец, сидящих на руках. Эти же щупальца осуществляют функцию кожного дыхания.

Пищеварительная система представлена пищеводом и желудком, куда впадают протоки печени. У некоторых плеченогих анус редуцирован. Кровеносная система незамкнутая. От сердца отходит одна аорта, разветвляющаяся на артерии. Выделительная система представлена одной-двумя парами целомодуктов.

Плеченогие — раздельнополые животные. Спермии из воды проникают в целом самок, где оплодотворяют яйца. Личинки трохофорного типа. Брахиоподы образуют скопления на дне морей и океанов, играя роль биофильтраторов.

ТИП ХОРДОВЫЕ (Chordata)

Это наиболее высокоорганизованная группа животных, освоивших различные среды обитания. К хордовым принадлежат трехслойные, двусторонне-симметричные, вторичнополостные; вторичноротые животные метамерного строения. Метамерия более полно выражена у первичноводных хордовых и на ранних стадиях эмбрионального развития. Как и другие вторичнополостные животные, хордовые имеют кровеносную систему, органы выделения метанефридиального типа, их половые железы развиваются в целоме.

Для хордовых характерны следующие, присущие только им признаки (рис. 177), перечисленные ниже.

1. Осевой скелет представлен хордой — упругим гибким несегментированным стержнем энтодермального происхождения, тянущимся вдоль тела животного над пищеварительной трубкой. На ранних стадиях развития хорда закладывается у всех хордовых, но у более высокоорганизованных животных на более поздних стадиях онтогенеза замещается сегментированным позвоночником мезодермального происхождения.

2. Центральная нервная система представлена нервной трубкой эктодермального происхождения с каналом (невроцелем) внутри. Она расположена на спинной стороне тела животного над хордой.

3. В переднем отделе пищеварительной трубы — глотке — по бокам имеются жаберные щели, которые образуются у всех хордовых на эмбриональной стадии развития, но пожизненно сохраняются только у первичноводных животных.

Известно около 45 тыс. видов хордовых, обитающих как в водной среде, так и на суше. Большинство из них ведет подвижный образ жизни. Хордовые имеют огромное хозяйственное значение. К этому типу принадлежат практически все сельскохозяйственные животные. Мно-

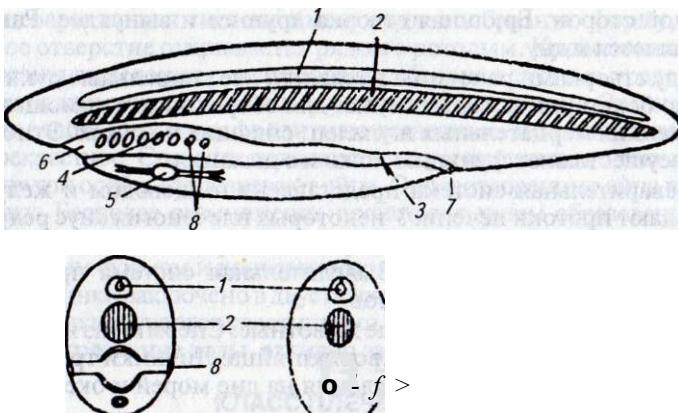


Рис. 177. Схема строения хордовых:
1 — нервная трубка; 2 — хорда; 3 — кишечник; 4 — глотка; 5 — сердце; 6 — рот; 7 — анальное отверстие; 8 — жаберные щели

гие хордовые служат объектом промысла, давая большое количество мяса, жира, кожи, пушнины и других продуктов промышленного сырья. Среди хордовых имеются виды, приносящие пользу истреблением паразитов и вредителей сельского хозяйства, но имеется также немало видов, причиняющих значительный ущерб растениеводству и животноводству. Некоторые виды участвуют в распространении опасных заболеваний растений, животных и человека.

Хордовые — это вершина эволюции вторичноротовых. Существование типа было обосновано известным русским зоологом А. О. Ковалевским, а название типа предложено Бэллом в 1878 г. Современную классификацию типа хордовых можно представить следующим образом:

Тип Хордовые (Chordata)

Подтип Бесчерепные (Acrania)

Класс Головохордовые (Cephalochordata)

Подтип Личнохордовые (Urochordata)

Класс Асцидии (Ascidiae)

Класс Аппендикулярии (Appendiculariae)

Класс Сальпы (Salpae)

Подтип Позвоночные (Vertebrata)

ПОДТИП БЕСЧЕРЕПНЫЕ (Acrania)

Бесчерепные — мелкие морские, преимущественно донные животные (около 30 видов), сохраняющие все основные признаки типа в течение всей жизни (осевой скелет в виде хорды, нервная сис-

тема в виде недифференцированной нервной трубы и глотка с жаберными щелями). Их организация — это как бы схема строения хордового животного. Бесчерепные представляют большой интерес для решения вопроса о происхождении позвоночных животных. Познанию бесчерепных наука обязана прежде всего исследованиям А. О. Ковалевского.

КЛАСС ГОЛОВОХОРДОВЫЕ (*Cephalochordata*)

Головохордовые — это единственный класс бесчерепных. К числу представителей этого класса относятся ланцетники. Наиболее обычным и хорошо изученным является европейский ланцетник (*Amphioxus lanceolatus*), обитающий в Черном море (рис. 178, а). Это небольшое животное (длиной до 8 см) обитает на мелководье морей; обычно он лежит на грунте или зарывается в песок, выставив наружу передний отдел тела. Ланцетники питаются мелкими пищевыми частицами, находящимися в воде и оседающими на дно. Таким образом, ланцетники являются хорошими биофильтрами.

Форма тела ланцетника вытянутая, сжатая с боков, заостренная спереди и сзади. По спине тянется невысокая продольная складка кожи — спинной плавник, переходящий в хвостовой плавник характерной копьевидной формы. Парных конечностей нет.

Кожа голая, покрыта слизью. Эпидермис однослойный, в нем располагаются одноклеточные кожные железы. Под эпидермисом находится соединительнотканная дерма.

Скелет представлен хордой, тянувшейся вдоль всего тела. Хорда и лежащая над ней нервная трубка окружены соединительнотканной оболочкой. Соединительнотканые образования располагаются в основании плавников, между мышечными сегментами.

Мускулатура тянется лентами по обе стороны тела. Эти мускульные ленты метамерно разделены тонкими соединительнотканными перегородками (миосептами) на ряд мышечных сегментов (миомеров).

Центральная нервная система имеет вид трубы, с полостью (невроподобием), образующей в передней части расширение — зачаток желудочка головного мозга. От центральной нервной системы попарно отходят спинные — двигательно-чувствительные и брюшные — двигательные нервы, которые не соединяются в общие смешанные нервы, как у позвоночных животных.

Органы чувств примитивные. Околоротовые щупальца и вся поверхность тела выполняют осязательную функцию. На переднем конце тела находится обонятельная ямка. На ранних стадиях развития невроцель сообщается отверстием с внешней средой; позже это отверстие застывает, превращаясь в обонятельную ямку. По всей длине нервной трубы, в области невроцеля располагаются светочувствительные органы — глазки Гессе, состоящие из пигментной и светочувствительной клеток.

Питание и дыхание пассивное. Пищеварительная система начинается предротовой воронкой, окруженной щупальцами. На дне ее рас-

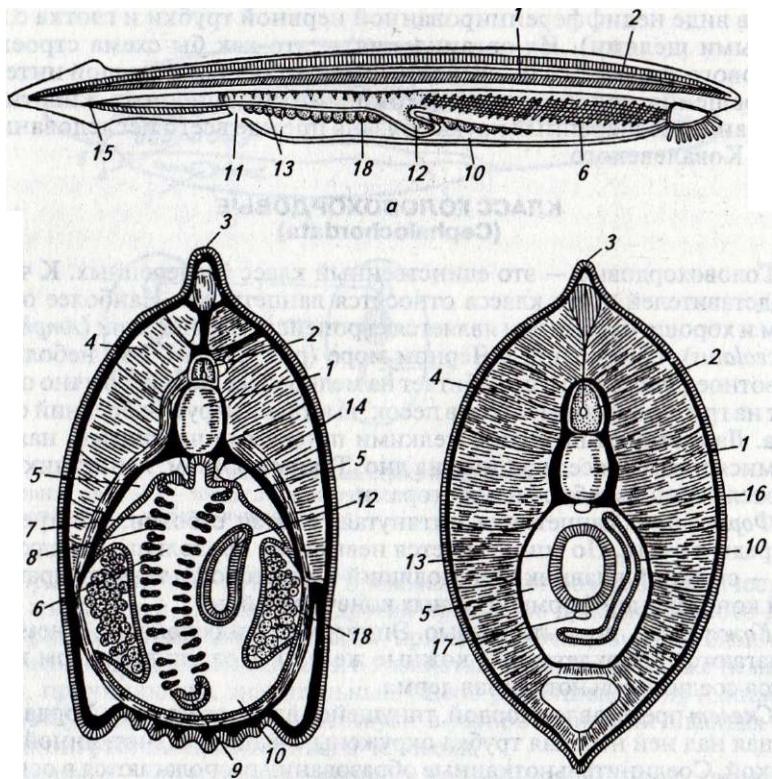


Рис. 178. Внутреннее строение ланцетника:
а — продольный разрез тела; поперечные разрезы тела в области глотки (б) и в области кишечника (в); 1 — хорда; 2 — нервная трубка; 3 — спинной плавник; 4 — миомер; 5 — целом; 6 — глотка; 7 — жаберная щель; 8 — межжаберная перегородка; 9 — эндостиль; 10 — околожаберная (атриальная) полость; 11 — атриопор; 12 — печень; 13 — кишка; 14 — нефридий; 15 — анальное отверстие; 16 — спинная аорта; 17 — подкишечная вена; 18 — половые железы

положен рот, ведущий в обширную глотку, переходящую в кишечник. Длина глотки может быть более половины длины кишечника. Кишечник тянется без изгибов и заметных расширений от глотки до анального отверстия. От брюшной стороны кишечника сразу за глоткой отходит печеночный вырост, который является гомологом печени позвоночных и совмещает в себе функции печени и поджелудочной железы.

В глотке ланцетника имеется эндостиль, обеспечивающий улавливание пищи и поступление ее в кишечник. Эндостиль — это продольный желобок, выстланный ресничным эпителием и железистыми клетками, который тянется под дном глотки. Он огибает рот с двух сторон и по спинной стороне глотки впадает в кишечник. С помощью щупалец и

ресничного эпителия в глотке ланцетник создает ток воды, которая поступает через рот и приносит взвешенные пищевые частицы, оседающие на дно глотки, и через жаберные щели выводится наружу. Осевшие частицы ослизываются и движением ресничек сначала подаются вперед, а затем по спинной борозде глотки попадают в кишечник.

Глотка велика и прободена многочисленными (около 100 пар) косо расположеными жаберными щелями. Они открываются в окологлаберную полость, которая образуется у эмбриона путем срастания по средней линии брюха двух боковых складок кожи. Из окологлаберной полости вода выводится наружу через непарное отверстие (атриопор) на брюшной стороне тела.

Кровеносная система замкнутая (рис. 179). Имеется один круг кровообращения. Сердце отсутствует, а кровь движется благодаря пульсации некоторых крупных сосудов. От венозного синуса начинается брюшная аорта, несущая венозную кровь и тянущаяся под глоткой. От брюшной аорты в обе стороны к межжаберным перегородкам отходят приносящие жаберные артерии. Через тонкие покровы происходит поглощение кровью растворенного в воде кислорода. Артериальная кровь через выносящие жаберные артерии поступает в парные наджаберные сосуды (корни спинной аорты), расположенные над глоткой, которые позади глотки сливаются в спинную аорту. Спинная аорта тянется под хордой, от нее отходят артерии к различным органам задней половины тела. Наджаберные сосуды продолжаются вперед сонными артериями, снабжающими кровью головной отдел животного. Венозная кровь от кишечника оттекает по подкишечной вене к печеночному выросту, где образует воротную систему. Из печени кровь по печеночной вене поступает в венозный синус, лежащий у корня брюшной аорты. Из передней и задней частей тела кровь собирается в передние и задние парные кардинальные вены. Последние, сливаясь, образуют правый и левый кювьеровы протоки, которые впадают в венозный синус. Так замыкается круг кровообращения.

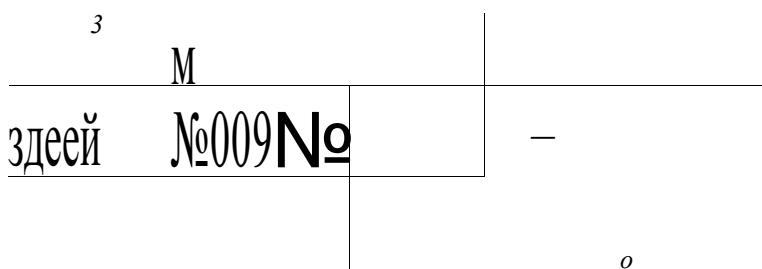


Рис. 179. Схема кровеносной системы ланцетника:
 / — кювьеровы протоки; 2 — брюшная аорта; 3 — жаберная артерия; 4 — корни спинной аорты; 5 — сонные артерии; 6 — спинная аорта; 7 — передние кардинальные вены; Я — задние кардинальные вены; 9 — подкишечная вена; 10 — печеночная вена; 11 — венозный синус

Рис. 180. Метанефридиий ланцетника:
1 — отверстие в целом; 2 — отверстие в атриальную полость; 3 — соленоциты

Органы выделения представлены видоизмененными метанефридиями (около 100 пар), расположеными метамерно в области глотки. Выводными протоками они открываются в окологабернную полость (рис. 180).

Органы размножения имеют вид двух рядов парных половых желез, лежащих на дне окологаберной полости. Половые продукты выводятся в окологаберную полость через временно образующиеся протоки и через атриопор попадают в воду; оплодотворение наружное.

ПОДТИП ЛИЧИНОЧНОХОРДОВЫЕ (*Urochordata*)

Известно около 1500 видов личиночнохордовых. Все они морские животные. Большая их часть во взрослом состоянии ведет сидячий образ жизни, прочно прикрепляясь к субстрату; некоторые живут в толще воды, перемещаясь течениями. Для личиночнохордовых характерно также то, что их тело заключено в оболочку, образованную веществами, близкими к клетчатке. В отличие от подавляющего большинства других хордовых личиночнохордовые — гермафродиты. Некоторые из них способны размножаться и бесполым путем, образуя почки. Хозяйственного значения не имеют.

КЛАСС АСЦИДИИ (*Ascidiae*)

Асцидии — самая многочисленная группа личиночнохордовых. Это примитивные хордовые животные, которые на личиночной стадии развития имеют все характерные для типа Chordata черты строения.

При переходе во взрослое состояние они утрачивают хорду, центральная нервная система из нервной трубки превращается в компактный нервный узел (лишь аппендикулярии сохраняют хорду и нервную трубку в течение всей жизни). Упрощение организма с возрастом у этих животных связано с переходом от подвижного существования личинки к неподвижному — взрослых особей (ретрессивный метаморфоз).

Большинство асцидий обитает на дне морей, прирастаю к камням и другим подводным предметам. Некоторые образуют колонии, пассивно плавающие в водах океана. Тело асцидий гладкое или бугристое (рис. 181). На верхнем конце его выдается короткий ротовой сифон с ротовым отверстием, у некоторых видов оно окружено щупальцами. Сбоку от ротового сифона расположен выводной (колоакальный) сифон. При раздражении тело асцидий может сжиматься, а сифоны втягиваться.

Покровы своеобразны. Снаружи тело асцидий одето толстой плотной оболочкой — туникой (рис. 182). Туника нередко имеет яркую окраску.

Скелет у взрослых особей редуцируется.

Нервная система состоит из лишенного полости ганглия, лежащего между сифонами, и отходящих от него нервов.

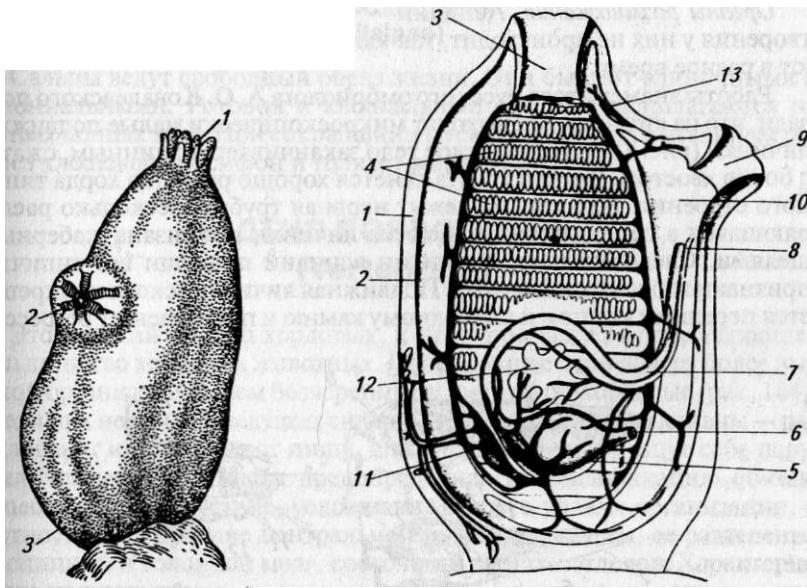


Рис. 181. Асцидия:
1 — ротовой сифон; 2 — клоакальный сифон; 3 — туника

Рис. 182. Внутреннее строение асцидии:
1 — туника; 2 — мантия; 3 — полость ротового сифона; 4 — жаберные щели; 5 — желудок; 6 — пищевод; 7 — задняя кишка; 8 — анальное отверстие; 9 — полость клоакального сифона; 10 — атриальная полость; 11 — сердце; 12 — половая железа; 13 — нервный узел

Пищеварительная система начинается ротовым сифоном, ведущим в обширную глотку, стенки которой прободены многочисленными мелкими отверстиями — жаберными щелями. Через жаберные щели вода поступает в окологлаберную полость и выводится наружу через клоакальный сифон. Ток воды через глотку вызывается колебаниями особой вибрирующей пластинки, расположенной в глотке, и движением ресничек мерцательного эпителия, выстилающего глотку. Вместе с водой в глотку попадают различные пищевые частицы, которые оседают на ее дно, где имеется эндостиль (как у ланцетника). Далее пища следует в пищевод, затем в мешковидный желудок и далее в короткую кишку, открывающуюся в окологлаберную полость. Непереваренные остатки пищи с током воды выносятся через клоакальный сифон наружу.

Кровеносная система незамкнутая. От мешковидного сердца отходят два сосуда, один из которых (жаберный) ветвится в межжаберных перегородках глотки, где и происходит газообмен, другой (кишечный) — идет к внутренним органам. Из этих сосудов кровь изливается в полости между органами, омывая их. Сердце работает маятникообразно — вначале гонит кровь по жаберному сосуду, затем в обратном направлении к внутренним органам.

Органы выделения у асцидий отсутствуют.

Органы размножения. Асцидин — гермафродиты, но самооплодотворения у них не происходит, так как яйцеклетки и спермии созревают в разное время.

Работы знаменитого русского эмбриолога А. О. Ковалевского показали, что из яиц асцидии выходят микроскопически малые подвижные личинки (рис. 183). Их округлое тело заканчивается длинным, сжатым с боков хвостом. Внутри хвоста тянется хорошо развитая хорда типичного строения. Над хордой лежит нервная трубка, несколько расширяющаяся в передней части. Глотка личинки пронизана жаберными щелями. Следовательно, личинкам асцидий присущи все типичные признаки хордовых животных. Подвижная личинка вскоре прикрепляется передним концом к подводному камню и претерпевает регрессив-

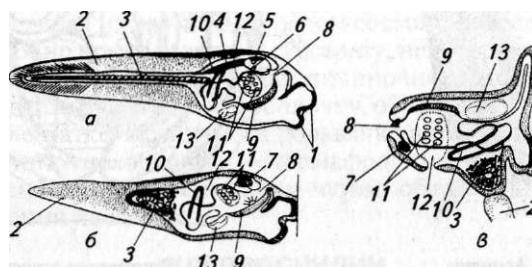


Рис. 183. Превращение личинки во взрослую асцидию:
а — стадия свободноплавающей личинки; б — только что прикрепившаяся личинка;
в — более поздняя стадия; 1 — сосочки прикрепления; 2 — хвост; 3 — хорда; 4 — нервная трубка;
5 — глазок; 6 — стомодеум; 7 — нервный узел; 8 — рот; 9 — эндостиль; 10 — задняя кишка;
11 — жаберные щели; 12 — атриальная полость; 13 — сердце

ные превращения. Хвост с находящейся в нем хордой и большей частью нервной трубы атрофируется. Передняя часть нервной трубы видоизменяется в нервный узел взрослой асцидии. Складками покровов образуется окологаберная полость. Снаружи тело покрывается туникой. Так подвижная личинка, несущая характерные для хордовых животных признаки, постепенно превращается в сидячую взрослую асцидию, утратившую большую часть признаков, свойственных представителям типа хордовых. Именно поэтому описываемые животные получили название личнонохордовых.

КЛАСС АППЕНДИКУЛЯРИИ (*Appendiculariae*)

В морях встречаются плавающие в толще воды своеобразные представители личнонохордовых — аппендикулярии. По строению они напоминают личинок асцидий. У них в течение всей жизни сохраняется длинный хвост, вдоль оси которого тянется хорда. Окологаберной полости нет. Кожа выделяет студенистый футляр, который соответствует тунике асцидий.

КЛАСС САЛЬПЫ (*Salpae*)

Сальпы ведут свободный образ жизни. Они бывают одиночными и колониальными. Ротовой и клоакальный сифоны располагаются на разных концах тела. При сокращении мускулатуры вода выталкивается через клоакальный сифон и толкает животное вперед.

ПОДТИП ПОЗВОНОЧНЫЕ (*Vertebrate*)

Это высший подтип хордовых, к нему принадлежит подавляющее большинство хордовых животных. Позвоночные отличаются более высокой организацией, чем бесчерепные и личнонохордовые (рис. 184). Среди них нет видов, ведущих сидячий образ жизни. Они активны — разыскивают и захватывают пищу, спасаются от врагов, ищут себе пару. Увеличение подвижности предопределило интенсификацию обмена веществ и, как следствие, усложнение общего уровня организации, в частности, усложнение центральной нервной системы, ее разделение на спинной и головной мозг, состоящий из пяти отделов. Характерно также наличие сердца, расположенного на брюшной стороне тела под кишечной трубкой. Название подтипа связано с замещением хорды сегментированным хрящевым или костным осевым скелетом — позвоночником.

Позвоночные животные широко распространены по земному шару. Многие позвоночные имеют большое хозяйственное значение.

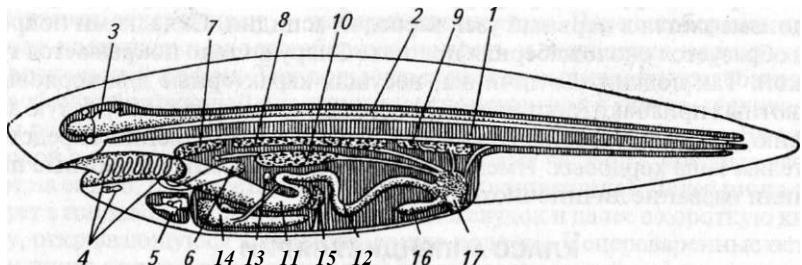


Рис. 184. Схема строения позвоночного животного:
 / — хорда; 2 — спинной мозг; 3 — головной мозг; 4 — жаберные щели; 5 — сердце; 6 — легкое; 7 — головная почка; 8 — туловищная почка; 9 — тазовая почка; 10 — половая железа; // — желудок; 12 — кишечник; 13 — поджелудочная железа; 14 — печень; 15 — селезенка; 16 — мочевой пузырь; 17 — клоака

Подтип позвоночных включает следующие систематические группы:
 Подтип Позвоночные — *Vertebrata*

Раздел Бесчелюстные — *Agnatha*

Надкласс Бесчелюстные — *Agnatha*

Класс Круглоротые — *Cyclostomata*

Раздел Челюстноротые — *Gnathostomata*

Надкласс Рыбы — *Pisces*

Класс Хрящевые рыбы — *Chondrichthyes*

Класс Костные рыбы — *Osteichthyes*

Надкласс Четвероногие, или Наземные позвоночные — *Tetrapoda*

Класс Земноводные, или Амфибии — *Amphibia*

Класс Пресмыкающиеся, или Рептилии — *Reptilia*

Класс Птицы — *Aves*

Класс Млекопитающие, или Звери — *Mammalia*

Строение и жизненные отправления. Форма тела позвоночных разнообразна. В теле различают голову, шею, туловище, хвост и конечности. Но у первичноводных видов шейный отдел отсутствует. У некоторых нет парных конечностей — отсутствие их бывает либо первичным, либо объясняется редукцией. Размеры позвоночных колеблются от нескольких миллиметров (некоторые рыбы) до 33 м (отдельные виды китов).

Покровы позвоночных образованы кожей, состоящей из двух слоев: наружного — эпидермиса и внутреннего — дермы. Эпидермис представлен многослойным эпителием. У круглоротых, рыб и личинок земноводных, жизнь которых тесно связана с водной средой, в эпидермисе залегает много железистых клеток, выделяющих слизь, облегчающую движение животного в воде. У наземных позвоночных — пресмыкающихся, птиц и млекопитающих — наружные слои клеток эпидермиса ороговевают. На их коже обычно образуются различные роговые производные — роговая чешуя, роговые щитки, перья, волосы и др. Дерма

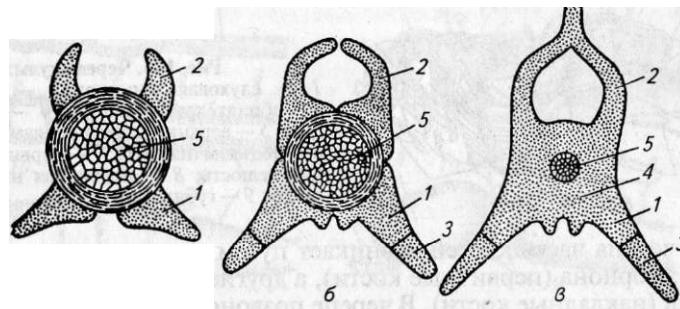


Рис. 185. Образование позвонка:
 а — стадия образования верхних и нижних дуг позвонка; б — стадия смыкания концов верхних дуг и образование спинномозгового канала; в — стадия формирования тела позвонка; 1 — нижняя дуга; 2 — верхняя дуга; 3 — боковой отросток позвонка; 4 — тело позвонка; 5 — хорда

кожи состоит из волокнистой соединительной ткани. В ней образуются костная чешуя (у рыб), костные щитки и кожные (накладные) кости. В коже позвоночных залегают железы различные по строению и функциям (слизистые, потовые, сальные и др.).

Скелет. У позвоночных первичной опорной структурой служит хорда. Но у взрослых особей хорда в той или иной степени замещается позвоночником, образованным отдельными позвонками (рис. 185). У мигонг (круглоротые) хорда полностью сохраняется в течение всей жизни животного, но в соединительнотканной оболочке ее и лежащего над ней спинного мозга развиваются метамерно расположенные парные хрящи, защищающие спинной мозг. Хорошо сохраняется хорда в течение всей жизни у осетровых рыб, но у них вокруг хорды образуются верхние и нижние хрящевые дуги позвонков. У костистых рыб в процессе онтогенеза хорда в значительной степени вытесняется позвоночником, образованным метамерно расположенными двояковогнутыми позвонками. Между позвонками сохраняются четкообразные или линзообразные остатки хорды. У взрослых наземных позвоночных двояковогнутые позвонки встречаются редко, и хорда сохраняется во взрослом состоянии лишь в виде незначительных остатков между позвонками. Отдельный позвонок обычно имеет тело, верхнюю дугу (через канал которой проходит спинной мозг) и нижнюю дугу.

Если тело позвонка вогнуто с обеих сторон, то позвонок называется амфицельным, если он вогнут спереди и выпуклый сзади — то процельным, если наоборот — то опистоцельным, а если спереди и сзади он плоский — то платицельным. К позвонкам прикрепляются ребра.

Череп позвоночных животных бывает хрящевым, костно-хрящевым или костным. При развитии костно-хрящевого черепа на хрящевую основу накладываются плоские кости, образующиеся за счет окостенения окружающей соединительной ткани. При формировании ко-

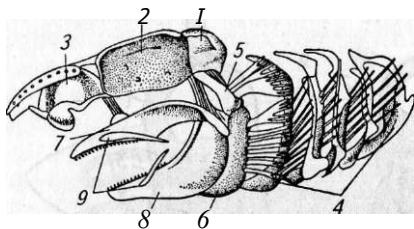


Рис. 186. Череп акулы:
 1 — слуховая капсула; 2 — глазница;
 3 — обонятельная капсула; 4 — жаберные
 дуги; 5 — верхний отдел подъязычной дуги;
 6 — подъязычная дуга; 7 — первичная верх-
 няя челюсть; 8 — первичная нижняя че-
 люсть; 9 — губные хрящи

стного черепа часть костей возникает путем окостенения хрящевого черепа эмбриона (первичные кости), а другие накладываются на него снаружи (накладные кости). В черепе позвоночного различают мозговую коробку и висцеральный скелет (рис. 186). Мозговая коробка защищает от механических воздействий головной мозг и органы чувств, расположенные на голове.

Висцеральный скелет черепа образован у круглоротых решетчатой хрящевой коробкой, а у прочих позвоночных в основе его лежит ряд парных жаберных дуг. В процессе эволюции одна из передних пар жаберных дуг видоизменилась в челюсти, а следующая за ней — в подъязычную дугу. У костных рыб и наземных позвоночных первичные хрящевые челюсти заменены вторичными, образованными накладными костями. Остальные жаберные дуги служат у рыб скелетом жаберного аппарата, а у наземных позвоночных редуцируются, частично входя в подъязычный аппарат.

У всех представителей класса, кроме миног и миксин, имеется скелет парных конечностей и их поясов (иногда он бывает редуцирован при атрофии конечностей). У рыб парные конечности имеют вид плавников, а у наземных позвоночных они построены по типу пятипалых ног (рис. 187).

В состав скелета передней пятипалой конечности (при ее типичном строении) входят плечевая кость, две кости предплечья (локтевая и лу-

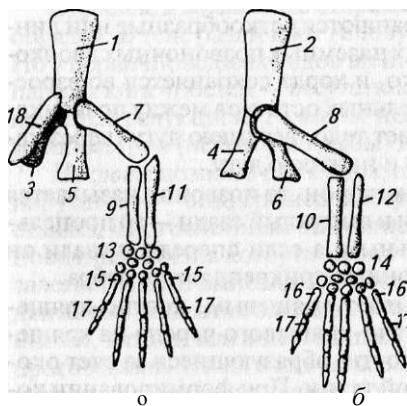


Рис. 187. Схема строения пятипалой конечности наземных позвоночных:
 а — передняя конечность; б — задняя ко-
 нечность; 1 — лопатка; 2 — подвздошная
 кость; 3 — прокоракоид; 4 — лобковая
 кость; 5 — коракоид; 6 — седалищная кость;
 7 — плечевая кость; 8 — бедренная кость;
 9 — лучевая кость; 10 — большая берцовая
 кость; 11 — локтевая кость; 12 — малая бер-
 цовидная кость; 13 — запястье; 14 — предплюс-
 на; 15 — пясть; 16 — плюсна; 17 — фаланги
 пальцев; 18 — ключица

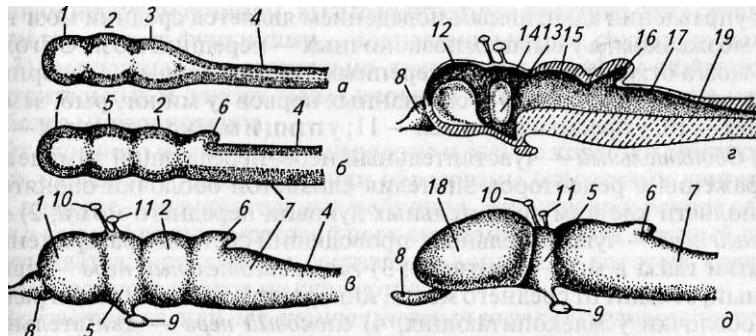


Рис. 188. Схема развития головного мозга позвоночных:

а—в — стадии развития мозга; г — продольный разрез мозга на стадии, представленной на рис. в; д — сформированный мозг с развитыми полушариями; 1 — зародыш переднего мозга; 2 — зародыш среднего мозга; 3 — зародыш заднего мозга; 4 — спинной мозг; 5 — промежуточный мозг; 6 — мозжечок; 7 — продолговатый мозг; 8 — обонятельные доли; 9 — гипофиз; 10 — теменной орган; 11 — эпифиз; 12 — желудочек переднего мозга; 13 — зрителные бугры; 14 — третий желудочек; 15 — желудочек среднего мозга; 16 — желудочек мозжечка; 17 — четвертый желудочек; 18 — полушария переднего мозга; 19 — невроцель

чевая), ряд костей запястья, пять костей пясти и фаланги пальцев. Скелет типичной пятипалой задней конечности образован бедром, двумя костями голени (большой и малой берцовой), несколькими косточками предплюсны, пятью костями плюсны и фалангами пальцев.

Плечевой пояс служит опорой для скелета передних конечностей. У наземных позвоночных он образован лежащими на спинной стороне тела лопатками и расположенными на брюшной стороне ключицами и коракоидами. Задние конечности сочленяются с тазовым поясом, который у земноводных, пресмыкающихся, птиц и млекопитающих слагается из трех пар костей — подвздошных, лобковых и седалищных. У рыб пояса конечностей имеют иное строение.

Мускулатура. У рыб и круглоротых с каждой стороны тела тянутся мощные продольные мышцы, разделенные тонкими соединительно-ткаными перегородками (миосептами) на ряд сегментов (миомеров). У наземных позвоночных метамерность мускулатуры во многом нарушена образованием многочисленных отдельных мышц.

Нервная система позвоночных разделяется на центральную, периферическую, симпатическую, парасимпатическую и вегетативную.

Центральная нервная система состоит из головного и спинного мозга. У эмбрионов позвоночных передний конец нервной трубы расширяется, образуя сначала три первичных мозговых пузыря (зародыши головного мозга), а в последующем пять отделов головного мозга (рис. 188). Из переднего пузыря в дальнейшем образуются передний и промежуточный мозг, из среднего — средний мозг, а из заднего — продолговатый мозг, на крыше которого путем выпячивания возникает мозжечок. Невроцель в головном мозге представлен желудочками мозга — полостями, заполненными жидкостью. У низших позвоночных высшим цен-

тром управления адаптивным поведением является средний мозг вместе с мозжечком, а у высших позвоночных — передний мозг. От головного мозга отходят 10—12 пар черепных чувствительных (сенсорных), двигательных (моторных) и смешанных нервов (у миног, рыб, земноводных — 10; у пресмыкающихся — 11; у птиц и млекопитающих — 12).

1) *Обонятельный* — чувствительный нерв, передающий химические раздражения с рецепторами эпителия слизистой оболочки обонятельной полости клеткам обонятельных луковиц переднего мозга; 2) *зрительный нерв* — чувствительный, проводящий световые раздражения с сетчатки глаза в мозг животного; 3) *глазодвигательный нерв* — двигательный, отходит от среднего мозга, иннервируя мышцы глаза и радужной оболочки у млекопитающих; 4) *блоковый нерв* — двигательный, идущий от среднего мозга к некоторым мышцам глазного яблока; 5) *тройничный нерв* — смешанный, отходящий от заднего мозга; иннервирует многие мышцы, железы и участки кожи головы, зубы, а также слизистую оболочку носовой и ротовой полостей; 6) *отводящий нерв* — двигательный, отходящий (как и все последующие) от продолговатого мозга и иннервирующий некоторые мышцы глаза, мигательной перепонки у пресмыкающихся и птиц; 7) *лицевой нерв* — смешанный, идущий к некоторым наружным мышцам головы; 8) *слуховой нерв* — чувствительный, передающий раздражения, воспринятые органом слуха и равновесия, в продолговатый мозг; 9) *языкоглоточный нерв* — смешанный, иннервирующий часть слизистой оболочки ротовой и глоточной полостей и органа слуха, а также мышцы глотки и языка; 10) *бужедающий нерв* — смешанный, участвующий в образовании нервных сплетений сердца и некоторых других органов (у низших позвоночных он иннервирует жаберные дуги, кроме первой, и органы боковой линии, а у высших — кожу задней части головы, часть глотки, щитовидную железу, дыхательные пути, легкие, печень, передний отдел кишечника, почки и некоторые другие органы); 11) *добавочный нерв* — двигательный, имеется только у высших позвоночных — иннервирует мышцы плечевого пояса и шеи; 12) *подъязычный нерв* — двигательный, контролирующий сокращение мышц языка и подъязычного аппарата.

Спинной мозг позвоночных животных в течение всей жизни сохраняет трубчатое строение и состоит из расположенного внутри серого и снаружи — белого мозгового вещества. В сером веществе находится большое число нервных клеток и нервных волокон. В белом веществе располагаются проводящие пути, образованные отростками нервных клеток.

Периферическая нервная система позвоночных представляет собой сложную систему нервов, отходящих от головного и спинного мозга к различным органам тела. Нервы слагаются из многих нервных волокон, являющихся отростками нервных клеток, находящихся в мозге и в нервных узлах — ганглиях. Различают три категории нервов: 1) чувствительные нервы проводят раздражения, получаемые из внешней среды или от внутренних органов, к нервным клеткам мозга. По ним нервные импульсы идут центrostремительно, т. е. к мозгу; 2) двигательные нервы передают нервные импульсы центробежно от мозга к

мышцам и другим органам, вызывая ответную реакцию этих органов в соответствии с их функциями — сокращение мышц, секрецию желез и др.; 3) смешанные (чувствительно-двигательные) нервы слагаются из чувствительных и двигательных нервных волокон. Эта группа нервов наиболее многочисленна.

От спинного мозга отходят передние и задние корешки спинномозговых нервов. Передние корешки образованы отростками двигательных, а задние — чувствительных нейронов. Эти корешки вскоре сливаются в единый спинномозговой нерв смешанного типа, который вновь разветвляется; исключение составляют миноги, у которых корешки спинномозговых нервов не сливаются.

Вегетативная, или автономная, нервная система осуществляет регуляцию работы внутренних органов животного — сокращение сердца, перистальтику кишечника, секрецию желез и др. Деятельность вегетативной нервной системы имеет некоторую степень автономности, но все же контролируется центральной нервной системой. Центры вегетативной нервной системы находятся в сером веществе головного и спинного мозга. Возбуждение по парным нервам вегетативной нервной системы проходит последовательно по двум нейронам. Тело одного нейрона находится в центральной нервной системе, а тело другого — за ее пределами в нервных узлах, расположенных по бокам позвоночника в полости тела и во внутренних органах. Нервный импульс из ЦНС поступает в нервный узел и далее — к иннервируемому органу. Таким образом, нервный импульс, получаемый органом через вегетативную систему, контролируется центральной нервной системой.

Органы чувств в связи с активной жизнью позвоночных обычно отличаются сложностью строения и функции. Глаза позвоночных имеют форму бокала или яблока, внутренняя полость которого заполнена студенистым стекловидным телом (рис. 189). Спереди расположен круглый или линзовидный хрусталик. Его форма у наземных позвоночных может меняться; это изменяет фокусное расстояние хрусталика и обусловливает аккомодацию глаза. У рыб аккомодация достигается путем перемещения хрусталика. Стенки глазного бокала состоят из трех слоев (оболочек). Наружная фиброзная оболочка образует

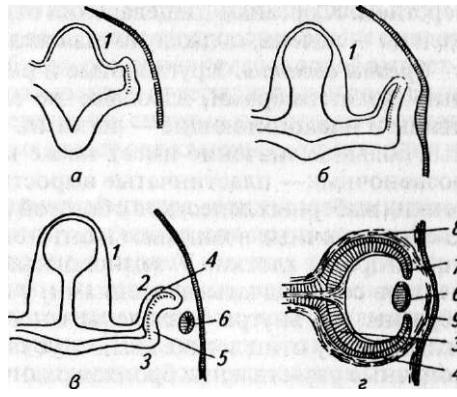


Рис. 189. Схема развития и строения глаза позвоночного животного:
a—c — последовательные стадии развития глаза; 1 — промежуточный мозг; 2 — глазной бокал; 3 — ножка глазного бокала; 4 — пигментная оболочка; 5 — сетчатка; 6 — хрусталик; 7 — радужка; 8 — склеры; 9 — роговица

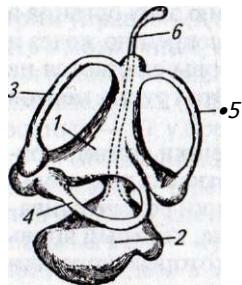


Рис. 190. Схема строения слухового аппарата позвоночных:
1 — овальный мешочек; 2 — круглый мешочек; 3 — передний полукружный канал; 4 — горизонтальный полукружный канал; 5 — задний полукружный канал; 6 — эндолимфатический проток

спереди прозрачную роговицу. Средняя сосудистая оболочка богата кровеносными сосудами и покрыта изнутри клетками, содержащими пигмент; этот пигмент служит для световой изоляции глаза. Внутренняя оболочка — сетчатка, клетки которой (палочки и колбочки) способны воспринимать световые и цветовые раздражения. Спереди глаза сосудистая оболочка переходит в радужную, или радужку, в центре которой располагается зрачок. Органы слуха позвоночных животных и связанные с ними органы равновесия (рис. 190) имеют различное строение; оно будет рассмотрено при описании отдельных групп животных. В передней части головы расположена непарная (у круглоротых) или парная (у всех остальных позвоночных) обонятельная полость, открывающаяся наружу соответственно непарной или парными ноздрями. У позвоночных, дышащих легкими, обонятельные полости соединяются с ротовой полостью внутренними ноздрями — хоанами, что позволяет им дышать, не раскрывая рта. В коже позвоночных имеются рецепторы осязания, восприятия температуры, давления и иных раздражений внешней среды. Имеются также рецепторы, воспринимающие раздражения внутренней среды.

Органы пищеварения имеют различное строение. Обычно пищеварительный тракт дифференцирован на ротовую полость, глотку, пищевод, желудок и кишечник и заканчивается клоакой или анальным отверстием. К органам пищеварения относятся также пищеварительные железы — печень, поджелудочная железа и другие.

Органы дыхания. Круглоротые и рыбы, а также личинки земноводных дышат жабрами, взрослые же земноводные, пресмыкающиеся, птицы и млекопитающие — легкими. В газообмене низших позвоночных большое значение имеет также кожное дыхание. Жабры водных позвоночных — пластинчатые выросты стенок жаберных щелей в виде тонких жаберных лепестков с богатой сетью кровеносных сосудов. Легкие позвоночных возникают в онтогенезе как парные выросты брюшной стороны глотки. У земноводных они имеют вид тонкостенных мешков со складчатыми стенками, у пресмыкающихся также мешкообразны, но внутри разделены многочисленными перегородками и складками, у птиц легкие имеют губчатое строение. У млекопитающих конечные разветвления бронхов оканчиваются в легких мельчайшими

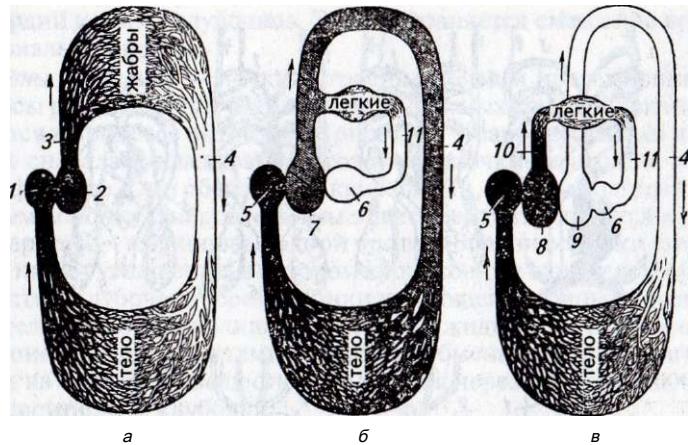


Рис. 191. Схема кровообращения у различных позвоночных животных (венозная кровь обозначена черным цветом; смешанная — заштрихованная; артериальная — белая):
а — рыба; б — земноводное; в — млекопитающее; 1 — предсердие; 2 — желудочек; 3 — брюшная аорта; 4 — спинная аорта; 5 — правое предсердие; 6 — левое предсердие; 7 — общий желудочек; 8 — правый желудочек; 9 — левый желудочек; 10 — легочная артерия; 11 — легочная вена

пузырьками — альвеолами. Стенки легких всех видов пронизаны сетью кровеносных капилляров. Воздух поступает к легким через ноздри в глотку, где начинается дыхательное горло — трахея, которая в грудной полости делится на два бронха, идущие к легким.

Кровеносная система замкнутая (рис. 191). Она состоит из сердца, артериальных сосудов, несущих кровь от сердца к различным органам, венозных сосудов, по которым кровь возвращается в сердце, и капилляров — мельчайших сосудов, соединяющих окончания разветвлений артерий и вен. Кровь позвоночных представляет собой бесцветную вязкую жидкость — плазму, в которой взвешены форменные элементы: эритроциты и лейкоциты. У круглоротых, рыб и личинок земноводных, дышащих жабрами, один круг кровообращения: от сердца венозная кровь идет к жабрам, где окисляется и освобождается от диоксида углерода, а затем разносится артериальными сосудами по всему телу, возвращаясь в сердце по системе венозных сосудов (рис. 192). Сердце у этих животных имеет две камеры — предсердие и желудочек.

У взрослых земноводных, пресмыкающихся, птиц и млекопитающих в связи с переходом к легочному дыханию два круга кровообращения (см. рис. 192). Соответственно в сердце имеется два предсердия. Малый круг образован легочными артериями, несущими венозную кровь от сердца к легким, и легочными венами, возвращающими окисленную кровь в левое предсердие. По артериям большого круга кровообращения обогащенная кислородом кровь разносится от серд-

а *е* *ж* *з*

Рис. 192. Схема артериального кровообращения и изменений жаберных артерий у позвоночных (венозная кровь обозначена черным цветом; смешанная — серая, артериальная — белая): *а* — рыба; *б* — двоякодышащие; *в* — личинка амфибии и хвостатые жаберные амфибии; *г* — хвостатая амфибия; *д* — бесхвостая амфибия; *е* — крокодил; *ж* — птица; *з* — млекопитающее; 1 — первая пара жаберных артериальных дуг (у наземных — дуги аорты); 2 — вторая пара жаберных артериальных дуг (у наземных — дуги аорты); 3 — третья пара жаберных артериальных дуг (у наземных недоразвивается); 4 — четвертая пара жаберных артериальных дуг (легочные артерии наземных); 5 — спинная аорта; 6 — брюшная аорта

ца по всему телу, где отдает кислород и растворенные в ней питательные вещества тканям и органам и насыщается диоксидом углерода и конечными продуктами метаболизма. В сердце (в правое предсердие) кровь возвращается по венозным сосудам. У всех позвоночных вены, несущие кровь от кишечника, входят в печень, где разветвляются в сложную систему капилляров, образуя воротную систему печени. Воротная система почек хорошо развита у рыб и слабо — у наземных позвоночных.

Сердце земноводных и большинства пресмыкающихся трехкамерное: оно имеет два предсердия и один желудочек. В желудочке происходит частичное смешение венозной крови, поступающей из правого предсердия, с артериальной кровью, поступающей из левого предсердия. У птиц и млекопитающих сердце состоит из четырех камер — двух

предсердий и двух желудочков. Этим устраняется смешение венозной и артериальной крови.

Органы выделения — почки. Строение почек и протекающие в них процессы различаются у представителей разных групп животных и изменяются в процессе онтогенеза (рис. 193). У зародышей рыб и земноводных сначала закладываются головные почки, имеющие характер метанефридиев. Они образованы канальцами, которые одним концом, несущим воронку с мерцательным эпителием, открываются в полость тела, а другим — в общий выводной проток. Вблизи воронки стенки канальца имеют утолщение, в котором кровеносные капилляры образуют сосудистый клубочек. Через воронки канальцев головных почек из полости тела удаляются излишки полостной жидкости с растворенными в ней конечными продуктами азотистого обмена. Часть же влаги и удаляемых из организма веществ фильтруется через стенки канальцев из крови сосудистого клубочка.

По мере развития зародышей рыб и земноводных головные почки сменяются туловищными почками. Последние располагаются позади головных, которые постепенно атрофируются. Туловищные почки имеют более сложное строение, чем головные. В них излишняя для организма вода и конечные продукты обмена выделяются из крови в особые тонкие сосуды, которые впадают в полость буменовых кап-

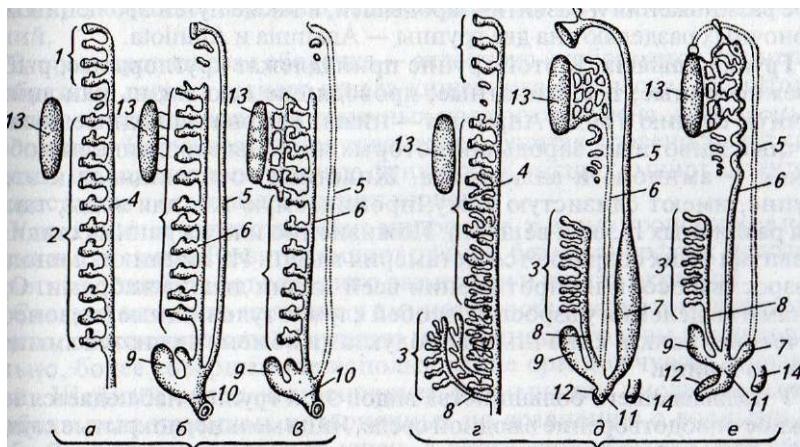


Рис. 193. Схема строения мочеполовых органов позвоночных:
 а — зародыш низшего позвоночного; б — самка низшего позвоночного; в — самец низшего позвоночного; г — зародыш высшего позвоночного; д — самка высшего позвоночного; е — самец высшего позвоночного; 7 — головная почка; 2 — туловищная почка; 3 — тазовая почка; 4 — проток эмбриональной почки; 5 — мюллеров канал, служащий у самок яйцеводом; 6 — вольфов канал, являющийся у низших позвоночных обоего пола мочеточником, а у самцов также и семяпроводом, у самцов высших позвоночных — только семяпроводом; 7 — матка; 8 — мочеточник тазовых почек; 9 — мочевой пузырь; 10 — клоака; 11 — мочеполовой синус; 12 — задняя кишечная труба; 13 — половая железа; 14 — кишка

сул, откуда жидкость стекает по коротким протокам в выводной канал почки.

В процессе развития зародышей рыб и земноводных меняются также и выводные протоки почек. Эмбриональные выводные протоки головных почек расщепляются вдоль на два канала — мюллеров и вольфов. Вольфов канал становится мочеточником первичной почки, тогда как мюллеров проток у самцов редуцируется, а у самок выполняет функцию яйцевода. У самцов вольфов канал выполняет функцию семяпровода.

У пресмыкающихся, птиц и млекопитающих в процессе эмбриогенеза туловищные почки заменяются тазовыми почками более сложного строения (в частности, с более сложным фильтрационным аппаратом), лежащими в полости таза. Мочеточники тазовых почек представляют собой новообразование.

Органы размножения. Почти все позвоночные животные раздельнополы. Число гермафронтитных видов очень невелико. Половые железы — семенники у самцов и яичники у самок — обычно парные. У круглоротых половые клетки из половых желез попадают в полость тела, откуда через особые поры, расположенные около мочевого отверстия, выводятся наружу. У других позвоночных для выведения наружу половых продуктов развиваются специальные протоки. Нередко они дифференцированы на ряд отделов.

На основе различий в образе жизни, особенностях строения, характере размножения и развития зародышей, а также путей эволюции позвоночных разделяют на две группы — *Anamnia* и *Amniota*.

Группа *Anamnia*. К этой группе принадлежат круглоротые, рыбы и земноводные, т. е. животные, проводящие всю жизнь или личиночную стадию в воде. Анамнии — низшие первичноводные позвоночные животные, зародыши которых лишены зародышевых оболочек — амниона и аллантоиса. Животные, относящиеся к этой группе, имеют слизистую кожу, проницаемую как для воды, так и для различных газов и веществ. Пожизненно или на ранних стадиях развития у них сохраняется метамерия мышц. Их личинки, а иногда и взрослые особи на протяжении всей жизни дышат жабрами. Органами выделения у взрослых особей служат туловищные (мезонефрические) почки; конечными продуктами обмена являются аммиак или мочевина.

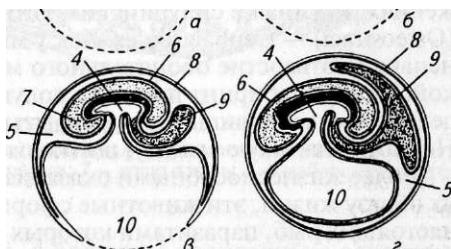
У подавляющего большинства видов этой группы наблюдается наружное оплодотворение в водной среде. Яйца мелкие, покрытые студенистой оболочкой, с небольшим запасом питательных веществ. Развитие зародыша протекает в водной среде без образования специальных зародышевых оболочек. Развитие с метаморфозом.

Группа *Amniota* включает высших наземных позвоночных животных — пресмыкающихся, птиц и млекопитающих.

Для эмбрионального развития высших позвоночных характерно возникновение особых зародышевых оболочек. В яйце в период развития образуются оболочки — амнион, окружающий зародыш, и ал-

Рис. 194. Зародышевые оболочки амниот:

а—г — стадии развития зародыши;
1 — эктодерма; 2 — энтодерма; 3 — мезодерма; 4 — полость кишечника;
5 —внезародышевая полость; 6 — амнион;
7 — амниотическая полость;
8 —сероза; 9 —аллантоис; 10 —желточный мешок



лантоис (рис. 194). По названию одной из зародышевых оболочек (амнион) высшие позвоночные именуются амниотами. Амнион развивается в виде складки эктодермы и мезодермы. После срастания краев этой складки зародыш оказывается сразу в двух оболочках — во внутренней (амнион) и наружной (сероза). Между амнионом и зародышем находится околоплодная, или амниотическая, жидкость, которая предохраняет зародыш от высыхания и механических повреждений.

Вторая зародышевая оболочка — аллантоис — образуется как мешкообразный вырост заднего отдела кишечника зародыша. Аллантоис служит местом накопления конечных продуктов обмена, выделяемых зародышем. Наружная стенка аллантоиса, богатая кровеносными сосудами, выполняет дыхательную функцию. У высших амниот аллантоис срастается с серозой и образует хорион.

У всех амниот оплодотворение внутреннее, т. е. оно происходит в половых путях самки. Развитие амниот прямое, без метаморфоза; молодое животное отличается от взрослого размерами и иногда окраской.

Следует отметить наличие у амниот хорошо развитого шейного отдела, обеспечивающего лучшую подвижность головы и, следовательно, более совершенное использование органов чувств и челюстей. Шейный отдел компенсирует уменьшение числа степеней свободы тела наземных позвоночных по сравнению с водными — рыбами. Кожа не выделяет слизи, а ороговевший наружный слой эпидермиса защищает тело от излишних потерь влаги. Головной мозг хорошо развит, характерно появление вторичного мозгового свода — неопалиума. Дышат амниоты только легкими; более совершенный механизм дыхания обеспечивают грудная клетка и воздушносные пути — трахея и бронхи. Туловищная почка (мезонефрос) заменена тазовой почкой (метанефросом), и конечным продуктом обмена является мочевая кислота.

РАЗДЕЛ БЕСЧЕЛЮСТНЫЕ (*Agnatha*)

НАДКЛАСС БЕСЧЕЛЮСТНЫЕ (*Agnatha*)

Много разнообразных ископаемых бесчелюстных найдено в отложениях, начиная с силурийских. Они составляют подкласс щитковых (*Osteostraci*) — *Cephalaspides*. Как у миног и миксин, у щитковых было непарное отверстие обонятельного мешка, и он не сообщался с глоткой. Ушной лабиринт имел два полукружных канала. Их голова и передняя часть тулowiща были покрыты костным головогрудным щитом. Несмотря на такую защиту, щитковые вымерли в верхнем девоне.

Более жизнеспособными оказались миноги и миксины, но, судя по их образу жизни, эти животные сформировались уже после появления настоящих рыб, паразитами которых они и стали.

Современные бесчелюстные — небольшая группа примитивных позвоночных животных с сосущим ротовым аппаратом без подвижных челюстей. Жаберных дуг нет. Парные конечности отсутствуют. Имеется непарная ноздря, ведущая в непарный обонятельный мешок.

КЛАСС КРУГЛОРЫТЫЕ (*Cyclostomata*)

Примитивная группа позвоночных животных, включающая миног и миксины. Это рыбообразные животные, не имеющие челюстей. Тело круглоротых вытянутое, цилиндрическое, несколько уплощенное с боков, особенно в задней части. Кожа голая, с многочисленными железами, покрыта слизью. Парные конечности отсутствуют — их не было и у предков круглоротых. Рот без челюстей, расположен в глубине ротовой присоски. Ноздря непарная. Хорда полностью сохраняется в течение всей жизни животного. По бокам спинного мозга метамерно расположены попарно небольшие хрящи — зачатки верхних дуг позвонков. Череп образован несколькими хрящами. Миксины и большинство миног — обитатели моря, но некоторые миноги заходят в реки для икрометания, а некоторые из них постоянно живут в пресных водах.

Строение и жизненные отправления. Тело круглоротых разделено на голову, тулowiще и хвост (рис. 195). Хвост оторочен узким хвостовым плавником. У миног на спине имеются также непарные спинные плавники.

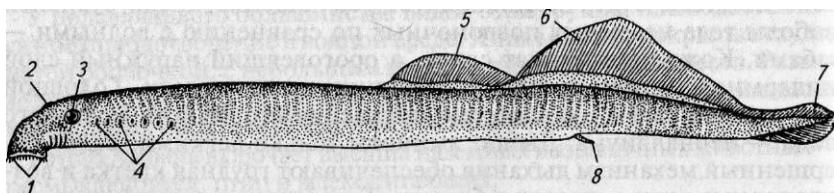


Рис. 195. Внешний вид миноги:
1 — ротовая присоска; 2 — ноздря; 3 — глаз; 4 — жаберные щели; 5, 6 — спинные плавники; 7 — хвостовой плавник; 8 — мочеполовой сосочек

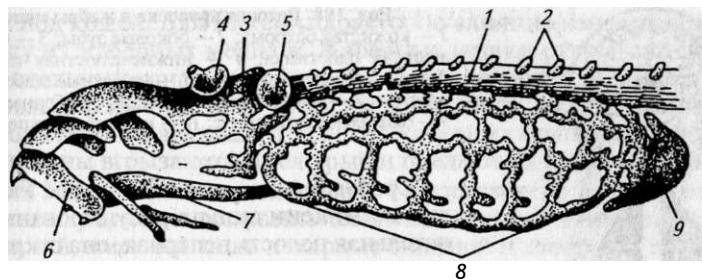


Рис. 196. Скелет миоги:
1 — хорда; 2 — верхние дуги; 3 — черепная коробка; 4 — обонятельная капсула; 5 — слуховая капсула; 6 — хрящ предротовой воронки; 7 — хрящ языка; 8 — жаберная решетка; 9 — околосердечный хрящ

Покровы круглоротых тонкие, с большим количеством слизистых железок.

Скелет представлен хорошо развитой хордой. По бокам спинного мозга в толще соединительнотканной оболочки, окружающей его и хорду, расположены два ряда небольших хрящиков, являющихся зародышами верхних дуг позвонков. Череп состоит из нескольких отдельных хрящей, соединенных тонкой перепонкой. Основанием черепа служит хрящевая пластинка, по бокам которой лежат слуховые капсулы, а спереди — обонятельная капсула. Скелет глоточной области имеет вид хрящевой решетки (рис. 196). Жаберные дуги и челюсти отсутствуют.

Мускулатура четко разделена миосептами на ряд миомеров.

Нервная система весьма примитивна. Головной мозг мал (рис. 197). В крыше переднего мозга нет нервных клеток. Мозжечок имеет вид валика на передней стенке продолговатого мозга, который занимает около половины всего головного мозга.

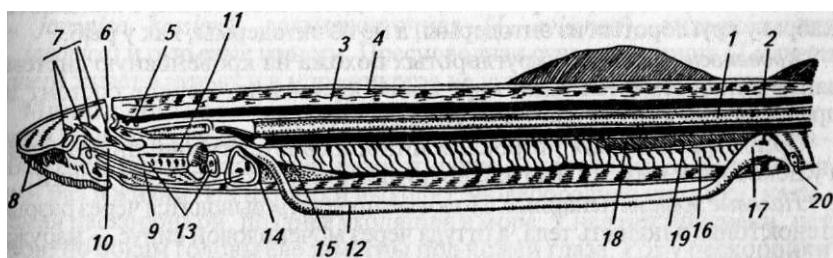


Рис. 197. Внутреннее строение миоги:
1 — хорда; 2 — спинной мозг; 3 — оболочка спинного мозга; 4 — мускулатура; 5 — головной мозг; 6 — орган обоняния; 7 — хрящи черепа; 8 — сосочки ротовой воронки; 9 — подъязычный хрящ; 10 — ротовая полость; // — пищевод; 12 — кишечник; 13 — дыхательный отдел глотки; 14 — сердце; 15 — печень; 16 — семенник; 17 — задняя кишка; 18 — почка; 19 — мочеточник; 20 — мочеполовой сосочек

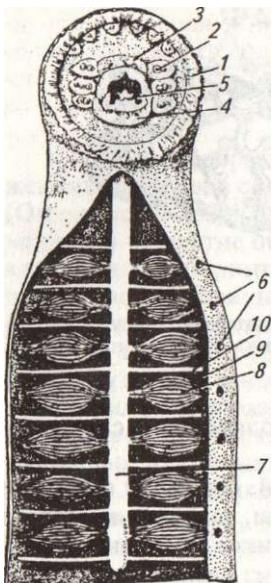


Рис. 198. Ротовая присоска и жабры миноги:
1 — кожистая бахрома; 2 — боковые зубы; 3 — верхнечелюстная пластина; 4 — нижнечелюстная пластина; 5 — язычковая пластина; 6 — наружные жаберные отверстия; 7 — дыхательная трубка; 8 — жаберный мешок; 9 — околожаберный синус; 10 — межжаберная перегородка

Органы чувств развиты слабо. Глаза малы, а у миксин сильно редуцированы. Обонятельная полость непарная, она открывается наружу одной ноздрей (но обонятельные нервы парные). Обонятельный мешок продолжается в так называемый питуитарный вырост. Ухо внутреннее; орган равновесия имеет только два полукружных канала. Имеются кожные рецепторы различного назначения.

Пищеварительная система начинается околовротовой присоской, усаженной роговыми зубчиками (рис. 198). В глубине присоски расположен рот, который ведет в обширную глотку. Глотка делится горизонтальной перегородкой на две части: спереди заканчивающуюся дыхательную и расположенную над ней пищеварительную (пищевод). Последний переходит в прямой, недифференцированный кишечник, заканчивающийся анальным отверстием. Имеется большая печень; желчного пузыря нет.

Органами дыхания служат жаберные мешки. Их строение отличается от строения жабр рыб. В боковых стенах дыхательной части глотки имеются парные отверстия, ведущие в жаберные мешки, стени которых несут многочисленные тонкие лепестки. В них расположена сеть кровеносных сосудов. У миног отверстия жаберных мешков открываются наружу самостоятельными жаберными отверстиями по бокам головы. Обычно их семь пар. У миксин же наружные жаберные отверстия ведут в продольный канал, открывающийся одним отверстием. Развиваются жабры у круглоротых из энтодермы, а не из эктодермы, как у рыб.

Кровеносная система круглоротых похожа на кровеносную систему ланцетника. Имеется один круг кровообращения. Сердце состоит из предсердия и желудочка. Селезенка отсутствует.

Органами выделения у взрослых миног являются туловищные почки, а у некоторых миксин всю жизнь функционируют головные почки.

Половые железы непарные. Половые продукты выводятся через разрыв стенок гонад в полость тела, а оттуда через мочеполовой синус — наружу.

Развитие у миксин прямое, у миног — со стадией личинки.

В класс Круглоротые входят два отряда: Миксины — Myxiniiformes и Миноги — Petromyzoniformes.

Отряд Миксины (Myxiniiformes). Представители этого отряда ведут паразитический образ жизни. Спинной плавник отсутствует. Глаза

скрыты под кожей. Для миксин характерно слияние каналов, выводящих воду из жаберных мешков. Жаберные мешки открываются в общий продольный канал, выходное отверстие которого находится позади головы. Таким образом, у них имеются лишь два сближенных на брюшной стороне наружных отверстия жаберных мешков. Живут в морях. Миксины вгрызаются в тело рыб и поедают их внутренности.

Отряд Миноги (Petromyzoniformes). Представители этого отряда — свободноживущие и полупаразитические животные. У миног нет парных конечностей, имеются лишь непарные спинной и хвостовой плавники. Гело удлиненное, цилиндрическое, без чешуи. Снизу на переднем конце головы открывается почти круглое отверстие большой ротовой воронки. Она служит присасывательным диском. Ротовая воронка несет на своей внутренней стенке роговые периодически сменяющиеся зубы различного размеров. Отверстие в глубине воронки ведет в ротовую полость, переходящую в пищевод и дыхательную часть глотки. Позади головы с каждой стороны тела имеются по семь жаберных отверстий. Каждое из них ведет в мешок с жаберными лепестками на его стенках. Своим внутренним отверстием мешок сообщается с дыхательной частью глотки — водопроводом. Через этот отдел глотки вода проникает в жаберные мешки, а из мешков — наружу. У присосавшейся миноги циркуляция воды осуществляется только через наружные жаберные отверстия. Пищевод переходит в едва намеченный, несколько расширенный по сравнению с кишкой желудок. Далее кишечник без изгибов продолжается до анального отверстия, имеется спиральный клапан. Кровеносная система и почки у миног имеют те же основные черты, что и у рыб. Миноги широко распространены в морях и пресных водах обоих полушарий.

Морская минога (*Petromyzon marinus*) — крупное животное, достигающее почти метра в длину. Встречается преимущественно в Атлантическом океане, между европейским и североамериканским берегами, в Средиземном и Балтийском морях. Для откладывания икры входят в реки. Более многочисленна речная минога (*Lampetra fluviatilis*). В Каспийском море обитает свой вид — волжская минога (*Caspiomyzon wagneri*). В таких же условиях встречаются украинская (*L. mariae*), сибирская (*L. japonica kessleri*), дальневосточная (*L. reissneri*), тихоокеанская (*L. japonica*) и ручьевые миноги. Пресноводная ручьевая минога (*Lampetra planeri*) живет в ручьях и в море никогда не уходит. Она мельче речных.

Миноги обитают в морях, за исключением ручьевых миног, живущих в ручьях и речках. Морские формы заходят в реки для икрометания. Питаются мелкими донными животными и падалью. Икру мечут один раз в жизни. Из икринки выходит червеобразная личинка, называемая пескоройкой, позднее уходящая в море. У нее нет присоски, нет зубов, по бокам головы еле заметны под кожей глаза. Рот у пескоройки поперечный, снабженный верхней и нижней губами. Верхняя очень подвижная губа помогает разгребать песок и ил, где пескоройка прячется. Она питается органическими компонентами ила, взвешенными в воде, которые пропускает через свой кишечник подобно ланцетнику или моллюску беззубке.

Пескоройка повторяет в своем зародышевом развитии и строении некоторые общие с ланцетником черты. Ланцетника и пескоройку можно признать близкими к предкам позвоночных. У пескоройки осевой скелет состоит только из хорды, но вокруг слабо еще развитого головного мозга, равно как и между жаберными щелями, появляются хрящи.

Около 3—4 лет пескоройка проводит в реке, растет и у некоторых миног почти достигает размеров взрослой особи. Далее наступает метаморфоз. По сторонам спинного мозга, над хордой, возникают хрящевые первичные еще неполные позвонки, головной мозг покрывается хрящевым черепом, глаза принимают нормальный вид, печень и поджелудочная железа утрачивают сообщение с кишечником. В печени начинает накапливаться жир, поджелудочная железа становится исключительно гормональным органом. Желтоватая окраска брюшной стороны тела пескоройки заменяется серебристой, темная же спина у речной миноги чернеет. Губы превращаются в присасывательный диск. Пескоройки скатываются в море еще до конца метаморфоза, завершая его уже в предустьевых участках. Молодая минога готова стать «кровопийцей». Впрочем, вместе с кровью минога поглощает и вырываемые зубами кусочки мяса своей жертвы, поедает и мертвую рыбу, способна также использовать отбросы рыбных промыслов, а у волжской миноги находили в кишечнике зеленые водоросли.

Мясо миног употребляется в пищу. Промышляют во время хода на нерест.

РАЗДЕЛ ЧЕЛЮСТНОРОТЫЕ (*Gnathostomata*)

Более высокоорганизованные позвоночные животные. Рот с подвижными челюстями. Две пары конечностей имеют вид плавников или ног. Ноздри и обонятельные полости парные. К данному разделу относятся хрящевые и костные рыбы, земноводные, пресмыкающиеся, птицы и млекопитающие.

НАДКЛАСС РЫБЫ (*Pisces*)

Рыбы — пойкилотермные водные позвоночные, органами движения которым служат парные и непарные плавники. У большинства рыб кожа содержит многочисленные железы и покрыта чешуей различного строения. По бокам тела имеются специфичные для первичноводных позвоночных органы боковой линии. Рот ограничен подвижными челюстями. Органами дыхания служат жабры эктодермального происхождения. Обонятельные отверстия парные. У всех рыб, кроме двоякодышащих, один круг кровообращения. Сердце имеет две камеры — предсердие и желудочек. Рыбы, как правило, раздельнополы, но встречаются и гермафродиты. Размножаются обычно икрометанием, но есть также и живородящие.

В современной фауне насчитывается около 25 тыс. видов рыб, большинство из которых живет в морях. Рыбы имеют огромное значение как продуценты ценных пищевых продуктов и технического сырья.

Современных рыб обычно подразделяют на два класса — хрящевые и костные.

Класс Хрящевые рыбы — *Chondrichthyes*

Подкласс Пластиноножаберные — *Elasmobranchii*

Надотряд Акулы — *Selachomorpha*

Надотряд Скаты — *Batomorpha*

Класс Костные рыбы — *Osteichthyes*

Подкласс Лучеперые — *Actinopterygii*

Надотряд Ганоидные — *Ganoidomorpha*

Надотряд Костищие — *Teleostei*

Подкласс Лопастеперые — *Sarcopterygii*

Надотряд Двоякодышащие — *Dipnoi*

Надотряд Кистеперые — *Crossopterygii*

КЛАСС ХРЯЩЕВЫЕ РЫБЫ (*Chondrichthyes*)

Древняя группа рыб, появившаяся около 300 млн лет назад и включающая около 700 современных видов. Скелет хрящевой, без костных элементов (рис. 199). Кожа покрыта плакоидной чешуй — примитивным типом чешуи, в образовании которой участвуют эпидермис и дерма (рис. 200). Группа включает два подкласса — Пластиноножаберные и Химеры.

ПОДКЛАСС ПЛАСТИНОНОЖАБЕРНЫЕ (ELASMOBRANCHII). В современной фауне этот подкласс представлен акулами и скатами (рис. 201), в основном обитающими в морях. Размеры колеблются от 20 см до 20 м. Среди них есть быстрые и ловкие пловцы, питающиеся подвижной добычей; относительно малоподвижные виды, питающиеся бентосом и планктоноядные — самые крупные. Подобно всем позвоночным концентра-

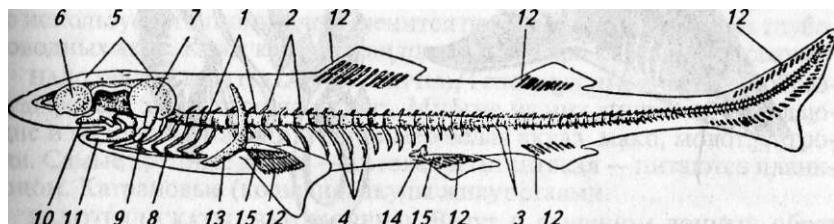


Рис. 199. Скелет акулы:
1 — позвонки; 2 — верхние дуги позвонков; 3 — нижние дуги позвонков; 4 — ребра;
5 — черепная коробка; 6 — обонятельная капсула; 7 — слуховая капсула; 8 — жаберная дуга;
9 — подъязычная дуга; 10 — небноквадратный хрящ; 11 — меккелев хрящ; 12 — радиалии;
13 — плечевой пояс; 14 — тазовый пояс; 15 — базалии

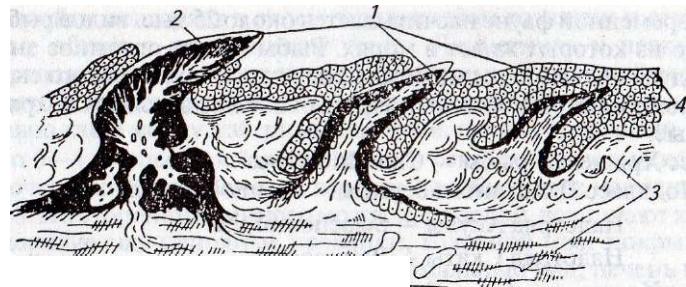


Рис. 200. Продольный разрез через плацоидную чешую и кожу акулы:
7 — плацоидные чешуи на разных стадиях развития (дентин обозначен черным, внутренняя полость, занятая мякотью, — белым); 2 — слой эмали; 3 — дерма; 4 — эпидермис

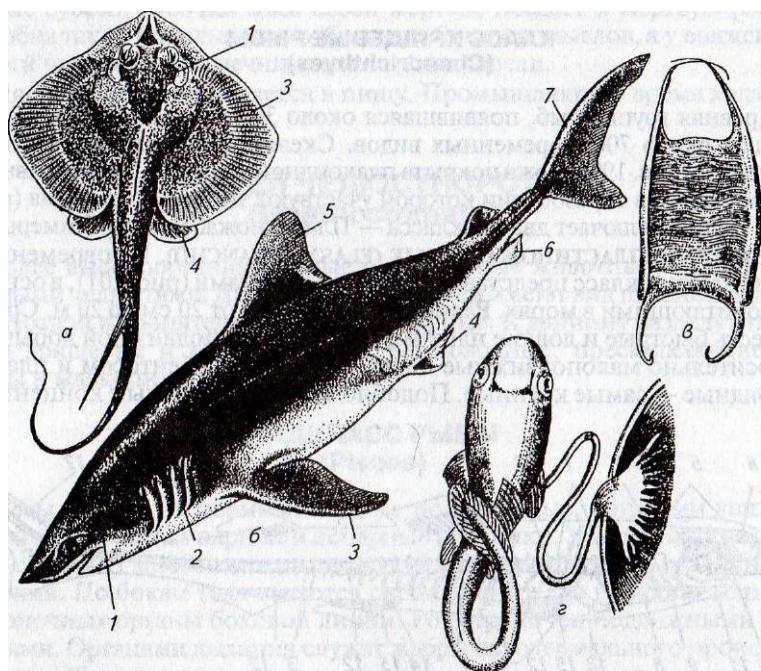


Рис. 201. Хрящевые рыбы:
а — скат хвостокол; б — гигантская акула; в — яйцо ската в оболочке; г — зародыш живородящей акулы; 1 — брызгальце; 2 — жаберные щели; 3 — грудные плавники; 4 — брюшные плавники; 5 — спинной плавник; 6 — анальный плавник; 7 — хвостовой плавник

ция солей в жидкостях тела у них приблизительно в три раза ниже, чем в морской воде, но при этом сохраняется осмотическое равновесие. Это достигается за счет высоких концентраций органических веществ в жидкостях, главным образом мочевины. У акул почки активно реабсорбируют мочевину из мочи, и она остается в крови. Ее содержание в крови у пластиножаберных в сто с лишним раз больше, чем у млекопитающих, это смертельно для других позвоночных. У пластиножаберных мочевина является необходимым компонентом всех жидкостей тела, и без ее высокой концентрации ткани не могут нормально функционировать. Позвоночник хорошо развит, но хорда сохраняется пожизненно. Грудные плавники располагаются горизонтально. Хвостовой плавник нециркониопасгенный (гетероцеркальный), в верхнюю лопасть заходит конец I позвоночника. Передний конец тела вытянут в рыло — рострум. Щелевидный рот расположен на нижней стороне головы. Жаберные щели в числе II яти — семи пар разделены межжаберными перегородками и открываются наружу поперечными отверстиями. На передней и задней стенках жаберных перегородок сидят рядами тонкие жаберные лепестки, имеющие густую сеть кровеносных сосудов. Желудочек сердца продолжается клоакой. В кишечнике хорошо развит спиральный клапан. Плавательного пузыря нет. Органами выделения являются туловищные почки, мочеточниками служат вольфовы каналы, которые открываются в клоаку. Прогрессивной чертой является внутреннее оплодотворение. У акул и скатов созревшие яйца выпадают из яичников в полость тела и увлекаются движением ресничек в воронки яйцеводов, которыми являются мюллеровы каналы. Оплодотворение происходит в половых путях самки, куда семя вводится с помощью особого копулятивного аппарата. Некоторые из акул живородящие, другие откладывают крупные яйца. У отдельных видов живородящих акул зародыши получают от материнского организма кислород и питательные вещества, поскольку связаны со стенками половых путей самки. Хорошо развиты головной мозг и органы чувств. Нервная ткань имеется и в крыше переднего отдела головного мозга. Однако масса головного мозга невелика — у акул она составляет 1/3700 массы тела.

Акулы и скаты во многих странах служат объектами промысла. Мясо используется в пищу, очень ценится печень, особенно печень глубоководных акул. Кожа крупных видов ценится как кожевенное сырье.

НАДОТРЯД АКУЛЫ (SELACHOMORPHA). Тело акул имеет торпедообразную форму, хвост хорошо развит. Многие из них хищники, нападающие и на млекопитающих, например белая акула, мако, молот, тигровая. Самые крупные акулы — китовая и гигантская — питаются планктоном. Катрановые (колючие) акулы живут стаями.

НАДОТРЯД СКАТЫ (BATOIDEA). Ведут в основном донный образ жизни, тело их уплощено в спинно-брюшном направлении, имеются (юльшие) грудные плавники. Питаются эти рыбы различными донными животными. Наиболее обычны морская лисица, морской кот; самый крупный скат — манта — питается планктоном; есть электрические скаты и скаты, похожие на акул — рыбы-пилы.

КЛАСС КОСТНЫЕ РЫБЫ

(*Osteichthyes*)

Строение и жизненные направления. По числу видов это самый многочисленный класс позвоночных животных. К этому классу относится около 25 тыс. видов. Костные рыбы населяют самые различные водоемы земного шара, как пресные, так и соленые. Форма тела рыб крайне разнообразна (рис. 202), что связано с многообразием их мест обитания и образа жизни. Размеры рыб колеблются в очень широких пределах — от 0,7 см до 5—7 м. Масса некоторых рыб достигает 2 т.

Плотность тела костных рыб равна или несколько выше плотности воды, т. е. у рыб нулевая или близкая к ней плавучесть. У рыб, ведущих

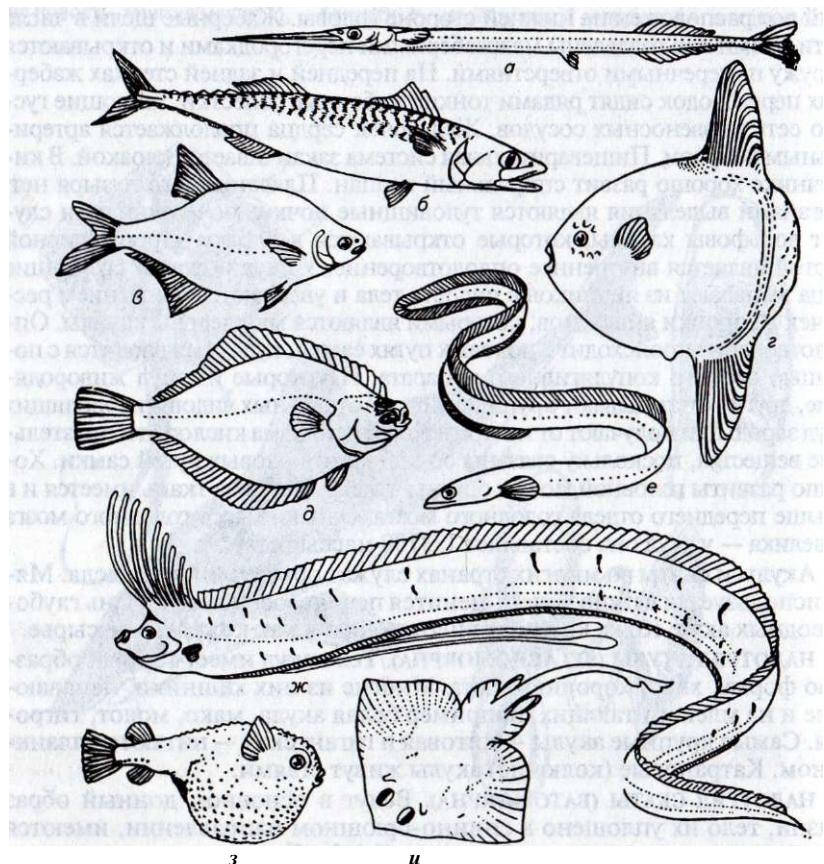


Рис. 202. Форма тела различных рыб:
a — сарган; *b* — скумбрия; *c* — лещ; *d* — луна-рыба; *e* — камбала; *f* — угорь; *g* — сельдяной король; *h* — иглобрюх; *i* — скат

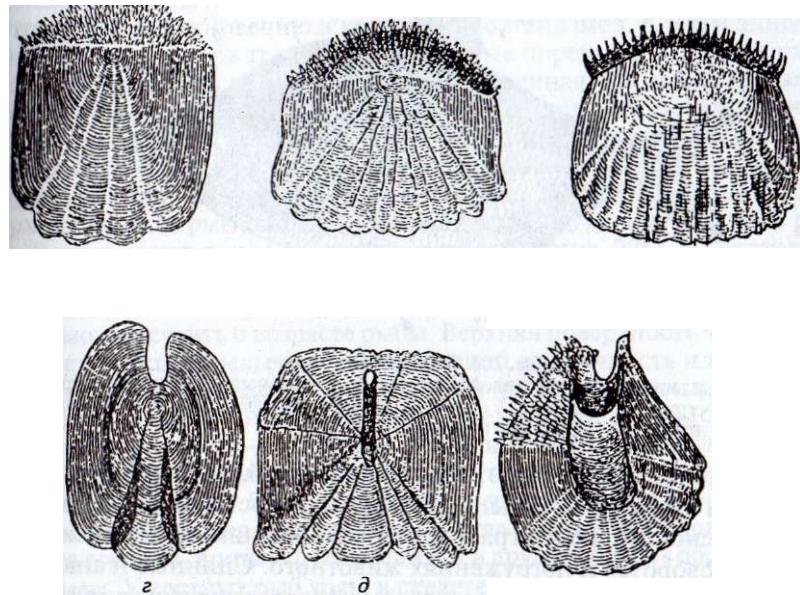


Рис. 203. Чешуи различных костистых рыб:
 л — ктеноидная чешуя судака; б, е — ктеноидная чешуя ерша; в — ктеноидная чешуя бычка; г — циклоидная чешуя щуки; д — циклоидная чешуя красноперки

придонный и донный образ жизни, она может быть отрицательной — 0,05—0,07 у камбал и бычков. Нейтральная плавучесть обеспечивается наличием плавательного пузыря — специального гидростатического органа. Внутренний скелет окостеневший, костные элементы имеются всегда, например кожные кости. Чешуя космоидная, ганоидная, чаще kostная (ктеноидная или циклоидная) (рис 203). Жабры не разделены перегородками, жаберные щели всегда прикрыты жаберными крышками. По этому признаку костных рыб можно легко отличить от хрящевых. Спиральный клапан в кишечнике и артериальный конус в сердце сохраняются только у древних групп рыб. Оплодотворение в основном наружное, хотя есть и живородящие виды. Икра мелкая, лишенная роговых оболочек.

Органы движения. Рыбы передвигаются в воде, изгибая тело, и с помощью парных и непарных плавников (рис. 204). Парных плавников две пары: грудные и брюшные. Парные плавники поддерживают тело рыбы в горизонтальном положении и служат рулями поворота и глубины. Иногда парные плавники видоизменены. Брюшные плавники бычков сращены и образуют присоску, с помощью которой рыбы удерживаются на дне в местах с быстрым течением или сильным прибоем. Очень длинные фудные плавники летучих рыб образуют несущие плоскости, на которых эти рыбы планируют, отделившись после разбега от воды.

Рис. 204. Внешний вид леща:
 1 — спинной плавник; 2 — хвостовой плавник; 3 — анальный плавник; 4 — брюшные плавники; 5 — грудные плавники; 6 — рот; 7 — ноздри; 8 — глаза; 9 — жаберная крышка; 10 — боковая линия; 11 — анальное отверстие

К непарным плавникам относятся хвостовой, один или несколько спинных и один (реже больше) анальный. В поступательном движении рыб основную роль играет хвостовой плавник; он же служит рулем при поворотах и погружениях животного. Спинные и анальные плавники прежде всего стабилизаторы направления движения рыбы, но они участвуют также в поворотах тела. Так, лещ движением длинного анального плавника может наклонять передний конец тела вниз, что облегчает нахождение пищи. У щук спинной и анальный плавники смещены к хвосту, что увеличивает мощность удара хвостом и, следовательно, обеспечивает стремительный бросок на добычу из засады. У живущей на дне морей рыбы-удильщика удлиненный передний луч спинного плавника расположен над верхней губой; колебания этой своеобразной удочки привлекают к хищнику добычу. У рыбы-прилипало спинной плавник превратился в присоску. Иногда плавники представлены колючками.

Покровы костных рыб представлены эпидермисом и дермой (рис. 205). Многочисленные одноклеточные железы эпидермиса выделяют слизь, которая тонким слоем покрывает тело рыбы. Это

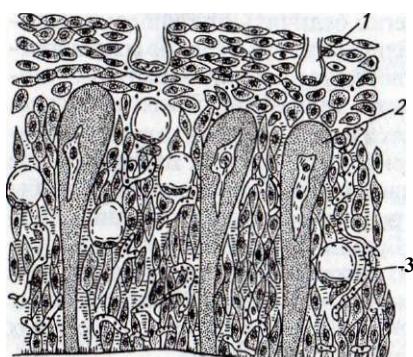


Рис. 205. Строение эпидермиса костистой рыбы:

1 — слизеотделительная одноклеточная железа; 2 — колбовидная одноклеточная железа; 3 — лимфатический сосуд

уменьшает трение при плавании, а бактерицидные свойства слизи препятствуют проникновению в кожу микроорганизмов. Клетки эпидермиса и дермы содержат пигменты, которые определяют окраску рыб. <)краска у большинства рыб покровительственная, что делает их мало-заметными в водной среде. У большинства рыб тело покрыто защитными костными образованиями — костной чешуей (циклоидной или ктеноидной), имеющей вид тонких, черепицеобразно налегающих друг на друга пластинок различной формы (см. рис. 203). Они развиваются в верхних слоях дермы, образуя в большинстве случаев правильные ряды. Чешуи растут в течение всей жизни рыбы (рис. 206). У некоторых иидов неравномерность роста чешуи в разные сезоны года ведет к образованию на них широких летних и узких зимних колец, по числу которых можно судить о возрасте рыбы. Верхняя поверхность чешуи часто имеет сложный рельеф, увеличивающий ее прочность и повышающий гидродинамические свойства тела рыбы. Циклоидная чешуя (сазан) имеет округлый задний край, у ктеноидной чешуи (окунь) задний край несет зубцы — «гребень». У многих рыб в нижних слоях чешуи лежит прослойка кристаллов извести и пигмента гуанина, усиливающих серебристый блеск рыб. При потере чешуи достаточно быстро происходит их полная регенерация. Иногда чешуи рыб видоизменяются, образуя иглы, шипы, костные щитки и другие кожные образования.

Скелет. У костных рыб хрящ в скелете частично или полностью замещается костной тканью, образуются хондральные, или «хрящевые»

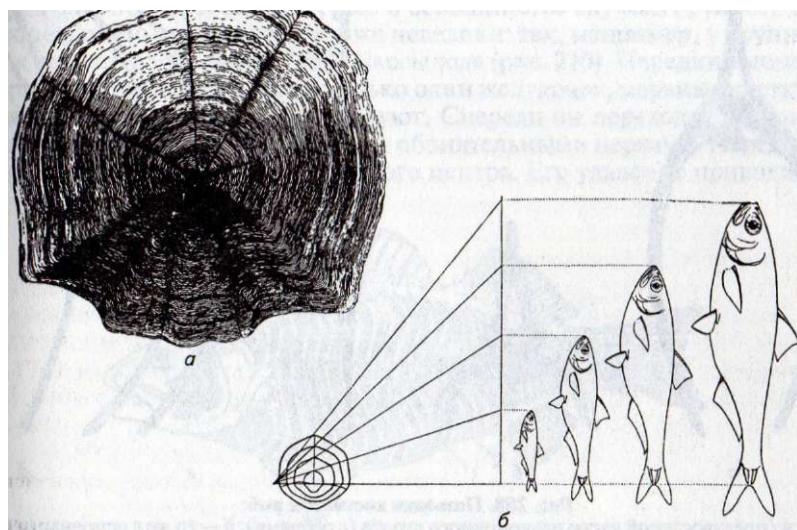


Рис. 206. Чешуя костистой рыбы (а) и соотношение между скоростью роста рыбы и размерами ее чешуи (б)

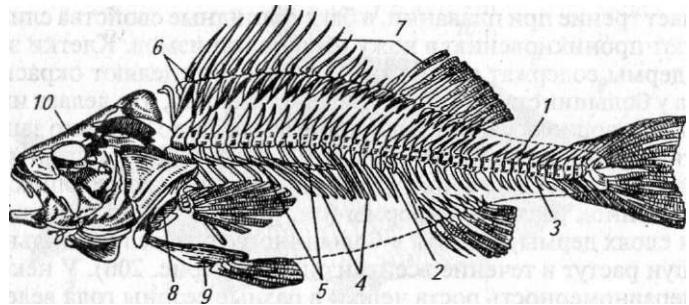


Рис. 207. Скелет костистой рыбы (окуня):
1 — остистые отростки позвонков; 2 — нижние дуги позвонков; 3 — нижние остистые отростки позвонков; 4 — ребра; 5 — мускульные косточки; 6 — основные косточки лучей плавников; 7 — лучи плавников; 8 — кости плечевого пояса; 9 — кости тазового пояса; 10 — череп

кости, возникающие независимо от покровных. Первоначальным же типом окостенения являются кожные, или покровные, кости, имеющие, как правило, вид пластинок. Покровные кости не имеют хрящевых предшественников, и их образование ведет к появлению новых элементов скелета, т. е. к его усложнению. У костистых рыб во взрослом состоянииrudименты хорды остаются только между позвонками, а скелет образован в основном костными элементами (рис. 207). У кис-

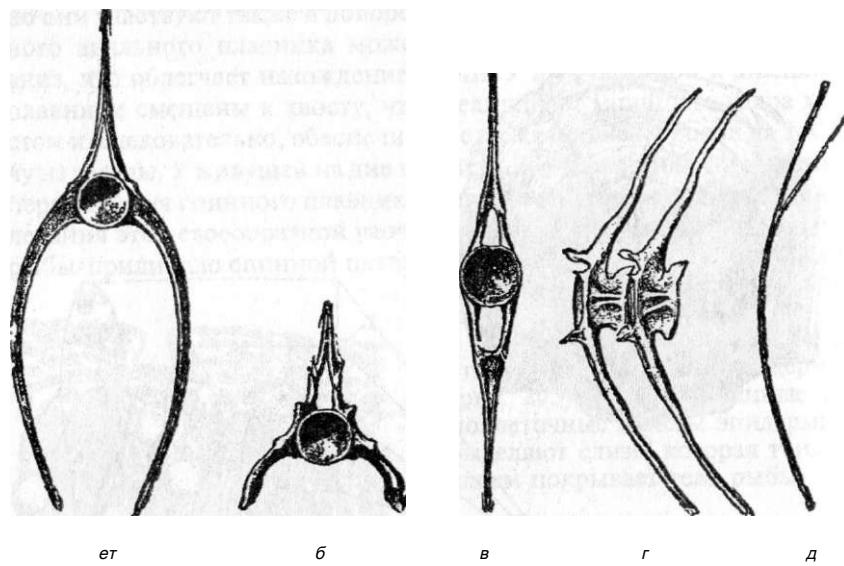


Рис. 208. Позвонки костистых рыб:
а — из предхвостовой части позвоночного столба (с ребрами); б — то же с основными отростками; в — из хвостовой части (с верхней и нижней дугами); г — то же сбоку (два позвонка); д — межмышечная косточка

теперых, двоякодышащих и осетровых рыб в течение жизни сохраняется хорда, а позвонки представлены только хрящевыми дугами.

Скелет рыб слагается из черепа, осевого скелета (позвоночника) и связанных с ним ребер, скелета парных плавников и их поясов и скелета непарных плавников. Череп состоит из мозговой коробки и висцерального скелета, который образован челюстной, подъязычной и пятью жаберными дугами. У большинства рыб ротовая полость вооружена одновершинными зубами, которые расположены не только на челюстях. Позвоночник разделен на туловищный и хвостовой отделы. Позвонки амфицельные, т. е. их тела двояковогнутые (рис. 208). Верхние дуги позвонков образуют спинномозговой канал, а нижние дуги хвостовых позвонков — канал, где проходят крупные кровеносные сосуды. С позвонками туловища связаны ребра, свободно оканчивающиеся в мускулатуре стенок тела. Парные плавники имеют пояса, лежащие в туловище рыбы. Основанием плавников служит внутренний опорный скелет. Наружные лопасти поддерживаются костными или хрящевыми плавниками лучами. Последние бывают жесткими, нерасчлененными или мягкими, членистыми; мягкие лучи иногда ветвятся.

Мускулатура туловища и хвоста рыб состоит из мышечных сегментов сложной формы (рис. 209). Вдоль тела двумя лентами тянутся продольные мышцы, разделенные перегородками (миосептами) на ряд мышечных сегментов (миомеров). Миомеры состоят из спинного и брюшного отделов. В голове, плавниках и их поясах расположены отдельные группы мышц (мускулатура челюстных и жаберных дуг, парных плавников и др.).

Нервная система костистых рыб более совершенна, чем у круглоротых. Головной мозг костистых рыб в большинстве случаев крупнее, чем хрящевых. Но размеры его все же невелики: так, например, у крупных щук он составляет лишь 1/1300 массы тела (рис. 210). Передний мозг не образует полушарий и имеет только один желудочек, нервные клетки в крыше переднего мозга отсутствуют. Спереди он переходит в обонятельные доли с отходящими от них обонятельными нервами. Передний мозг рыб играет роль обонятельного центра. Его удаление приводит к

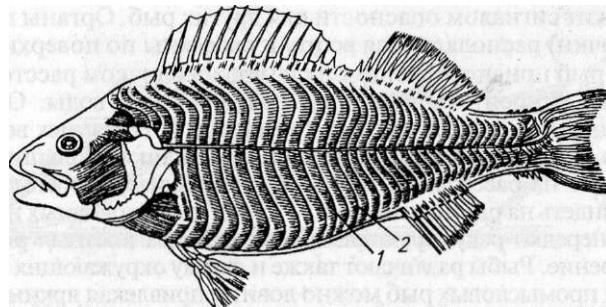


Рис. 209. Мускулатура костистой рыбы (окуня):
1 — миомеры; 2 — миосепты

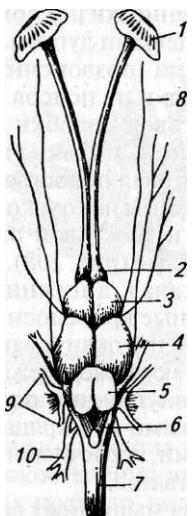


Рис. 210. Головной мозг костистой рыбы (окуния):

1 — обонятельные капсулы; 2 — обонятельные доли; 3 — передний мозг; 4 — средний мозг; 5 — мозжечок; 6 — продолговатый мозг; 7 — спинной мозг; 8—10 — черепные нервы

потере рефлекса на восприятие запахов. Промежуточный мозг достаточно развит. На его крыше расположен эпифиз, на нижней стороне расположена другая железа внутренней секреции — гипофиз. От дна этого отдела мозга отходят зрительные нервы. Средний мозг по размерам превосходит остальные отделы. Сверху выдаются два зрительных бугра, где оканчиваются волокна зрительных нервов. Мозжечок хорошо развит в первую очередь у подвижных видов рыб. У костных рыб, как и у других позвоночных, он является центром регуляции движений и сохранения равновесия. Продолговатый мозг рыб прикрыт сверху эпителиальной пленкой. От головного мозга костных рыб, как и у хрящевых, отходят десять пар головных нервов. Спинной мозг тонок, он тянется до конца позвоночника и имеет такое же строение, как и у хрящевых рыб.

Органы чувств костных рыб приспособлены к функционированию в условиях водной среды. Особенно важную роль играют органы химического чувства (обоняние, вкус), которые дополняют друг друга. Органы обоняния имеют вид пары мешков с хорошо развитыми складками

обонятельного эпителия, открывающихся наружу двумя ноздрями. Обоняние позволяет рыбам хорошо распознавать запахи чужих видов и различных особей своего вида. Так, ослепленный налим находит червя по запаху на расстоянии до 30 см, но не может его обнаружить на расстоянии 1 см, если ему закрыть ноздри. Угорь воспринимает запах при наличии 15—50 тыс. молекул в 1 л. У многих, особенно стайных рыб, в коже содержится «вещество страха». При травме кожи оно попадает в воду и служит сигналом опасности для других рыб. Органы вкуса (кусковые сосочки) располагаются во рту и рассеяны по поверхности тела.

Глаза у рыб приспособлены к видению на близком расстоянии, что обусловлено относительно малой прозрачностью воды. Они имеют плоскую роговицу и шаровидный хрусталик. Подвижных век нет. Аккомодация достигается перемещением хрусталика. Большинство рыб хорошо видит на расстоянии до 1 м, но некоторые, особенно крупные, способны видеть на расстоянии до 10—15 м. У глубоководных и пещерных рыб глаза нередко редуцируются. У большинства костных рыб развито цветовое зрение. Рыбы различают также и форму окружающих предметов. Некоторых промысловых рыб можно ловить, привлекая ярким светом.

Орган слуха и равновесия представлен только внутренним ухом, заключенным в капсулу. Собственно внутреннее ухо костных рыб (пере-

пончатый лабиринт) состоит из трех взаимно перпендикулярных полу-кружных каналов, отходящих от овального мешочка (вестибулярный аппарат) и нижнего круглого мешочка (собственно орган слуха), заполненных эндолимфой. В эндолимфе во взвешенном состоянии находятся мелкие камешки — отолиты. Отолиты прикреплены к вершинам ресничек эпителия, выстилающего внутреннее ухо. При изменении положения тела животного, давление и натяжение ресничек отолитами меняется, что и воспринимается нервными окончаниями. Сравнительно простое строение органа слуха рыб связано с большой звукопроводностью воды. Многие рыбы издают различные звуки зубами, жаберными крышками, трением плавников, с помощью плавательного пузыря и другими способами. Расшифровка этих звуков позволяет использовать их для обнаружения косяков рыбы при их ловле. У некоторых костных рыб перепончатый лабиринт соединен с плавательным пузырем. Перепончатый лабиринт улавливает изменение давления в плавательном пузыре, а последний служит резонатором и увеличивает чувствительность органа слуха. С своеобразным органом чувств рыб является боковая линия (рис. 211). У большинства рыб органы боковой линии имеют вид канала в коже животного, ветвящегося на голове. Через многочисленные поры, пронизывающие или не пронизывающие чешую, этот канал сообщается с внешней средой. В стенах канала расположены окончания ветвей блуждающего нерва. Органы боковой линии воспринимают даже слабые изменения движения и давления воды и инфразвуки. Осязательную функцию у рыб выполняют группы чувствительных клеток, расположенных по всей поверхности тела и образующих скопления на усиках, губах, лучах плавников. Рыбы способны улавливать изменения магнитного и электрического полей, а некоторые имеют специальные электрические органы.

Органы пищеварения у костных рыб более дифференцированы, разнообразнее устроен челюстной аппарат и шире спектр используемых кормов. Пищеварительный тракт рыб делится на три отдела: передний, включающий ротовую полость, глотку и пищевод; средний, состоящий из желудка, тонкой кишки и пищеварительных желез (печени и поджелудочной железы), и задний, представленный задней кишкой (рис. 212).

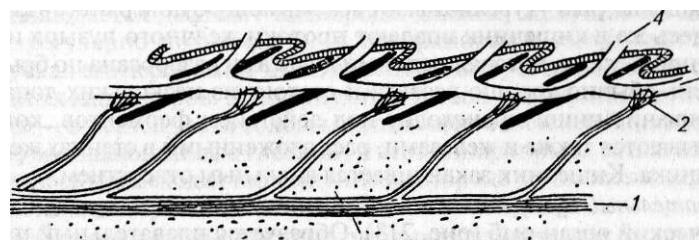


Рис. 211. Продольный разрез через канал боковой линии костистой рыбы:
1 — боковой нерв; 2 — чувствительные клетки органа; 3 — канал органа; 4 — наружное отверстие

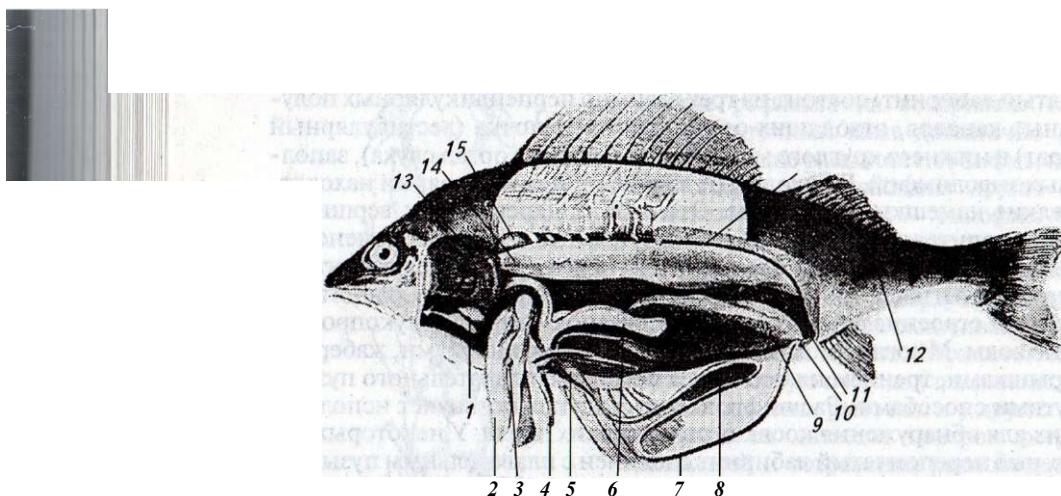


Рис. 212. Внутреннее строение костистой рыбы (окуния):
 1 — сердце; 2 — печень; 3 — желчный пузырь; 4 — пилорические выросты; 5, 7 — кишка;
 6 — желудок; 8 — селезенка; 9 — яичник; 10 — анальное отверстие; 11 — мочевой пузырь;
 12 — органы боковой линии; 13 — жабры; 14 — плавательный пузырь; 15 — почка;
 16 — мочеточник

В ротовой полости имеется язык. Обычно на челюстях и на нёбе расположены зубы, которые подвержены нерегулярной смене. Важную роль в добывании пищи у рыб играют жаберные тычинки — выросты вогнутой стороны жаберных дуг. У планктоноядных сельдей, сигов, толстолобиков длинные и многочисленные жаберные тычинки образуют своеобразный цедильный аппарат, который процеживает проходящую через жабры воду. Строение и подвижность челюстного аппарата и развитие зубов связаны с особенностями питания.

Короткий мускулистый пищевод без резкой границы переходит в желудок, форма и размеры которого разнообразны; иногда он слабо выражен, а у карповых вообще отсутствует. В самом начале кишечника у ряда рыб имеются слепые (пилорические) отростки, число которых может быть различно (у окуня — 3, у разных видов лососевых — до 400). Они выполняют примерно такую же функцию, что и спиральный клапан хрящевых рыб — увеличивают всасывающую поверхность кишечника. Здесь же в кишечник впадают протоки желчного пузыря и поджелудочной железы, которая мелкими дольками разбросана по брыжейке. Печень обычно хорошо развита и состоит из нескольких лопастей. Переваривание пищи происходит под действием ферментов, которые вырабатываются также и железами, расположенными в стенках желудка и кишечника. Кишечник заканчивается анальным отверстием.

Плавательный пузырь имеется у большинства костных рыб; это гидростатический орган рыб (рис. 213). Образуется плавательный пузырь как тонкостенный вырост спинной стороны начальной части пищевода; этот вырост у разных рыб имеет различные форму и строение и заполнен газом. У открытопузырных рыб связь с пищеводом сохраняется

в течение всей жизни (карпообразные, сельдеобразные). У закрытопузырных эта связь нарушается в процессе развития (окунеобразные). Изменяя объем плавательного пузыря, рыба может перемещаться в вертикальном направлении за счет изменения плотности тела и плавучести. Первоначальное заполнение плавательного пузыря газом происходит при заглатывании мальком атмосферного воздуха. В дальнейшем объем пузыря изменяется за счет заглатывания воздуха с поверхности открытопузырными рыбами или его выдавливания при сжатии пузыря. У закрытопузырных рыб изменение объема плавательного пузыря происходит за счет выделения и поглощения газов кровью через особое сосудистое сплетение капилляров в так называемой газовой железе. Плавательный пузырь отсутствует у многих донных рыб и рыб, совершающих быстрые вертикальные перемещения в воде.

Органами дыхания у рыб являются жабры. У костных рыб межжаберные перегородки редуцированы, и жаберные лепестки попарно сидят на жаберных дугах, прикрыты снаружи подвижными костными жаберными крышками. У костных рыб четыре жаберные дуги, на каждой из которых располагаются лепестки двух полужабр. У некоторых видов добавочная полужабра располагается на внутренней поверхности жаберной крышки. Снаружи лепестки покрыты тончайшими складочками, до 15 и более на 1 мм. Ток воды, омывающей жабры, обеспечивается главным образом движениями жаберных крышечек, а при плавании — самотеком. Важную роль в газообмене рыб играет кожа. Так, у годовалых карпов кожное дыхание обеспечивает до 1/8, а у карася и угря даже до 1/3 потребности организма в кислороде, энергично выделяется и диоксид углерода. У костных рыб имеются добавочные органы дыхания; обычно такие органы присутствуют у обитателей пресных водоемов, где наблюдается дефицит кислорода. Выносы, живущие на дне водоемов, регулярно поднимаются к поверхности и заглатывают воздух, пропуская его через кишечник, в стенках которого находится сеть кровеносных капилляров. У аквариумных рыбок — макроподов, лялиусов, гурами — в задней части головы расположен особый лабиринтовый аппарат, сообщающийся с глоткой; в этот аппарат рыба заглатывает воздух, и здесь происходит поглощение кислорода сетью капилляров. В газообмене может участвовать и плавательный пузырь.

Кровеносная система костных рыб в основных чертах сходна с кровеносной системой ланцетника, круглоротых и хрящевых рыб (рис. 214). У всех рыб, кроме двоякодышащих, один круг кровообращения. Сердце двухкамерное и состоит из предсердия и желудочка, имеется венозный

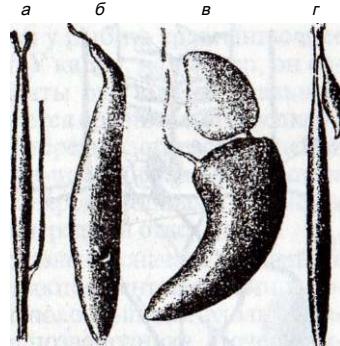


Рис. 213. Форма плавательного пузыря:
а — сельдь; б — сиг; в — карп;
г — многопер

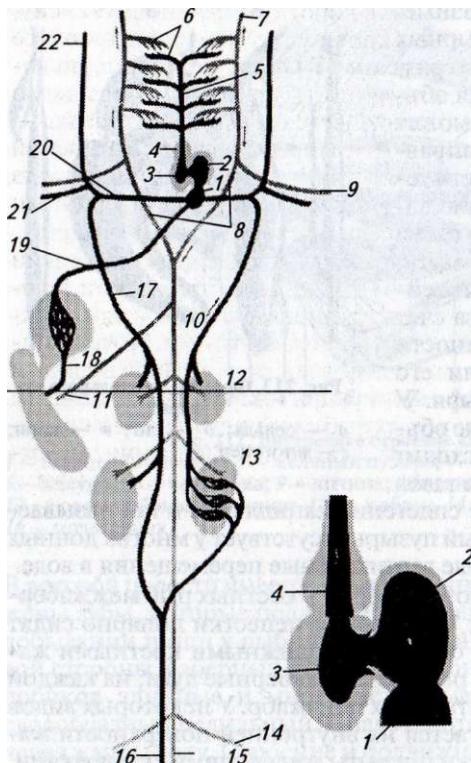


Рис.214. Кровеносная система кости-стой рыбы:

1 — венозный синус; 2 — предсердие; 3 — желудочек; 4 — луковица аорты; 5 — брюшная аорта; 6 — жаберные сосуды; 7 — левая сонная артерия; 8 — корни спинной аорты; 9 — левая подключичная артерия; 10 — спинная аорта; 11 — кишечная артерия; 12 — гонады; 13 — почка; 14 — левая подвздошная артерия; 15 — хвостовая артерия; 16 — хвостовая вена; 17 — правая задняя кардиальная вена; 18 — воротная вена печени; 19 — печеночная вена; 20 — правый юовьеверов проток; 21 — правая подключичная вена; 22 — передняя кардиальная (яремная) вена

синус, из которого кровь поступает в предсердие; сокращаясь, предсердие проталкивает кровь в желудочек. От желудочка отходит брюшная аорта. У костистых рыб имеется луковица аорты, образованная гладкой мускулатурой — утолщенное основание аорты, а артериальный конус редуцирован. От брюшной аорты отходят четыре пары приносящих жаберных артерий, несущих кровь к жабрам. Газообмен происходит в жаберных лепестках, где артерии распадаются на сеть капилляров. Кровь по капиллярам течет навстречу току воды (противоточная система), что усиливает газообмен.

Окисленная в жабрах кровь поступает в парные выносящие жаберные артерии, которые впадают в парные корни спинной аорги. Спереди корни аорты, отделив сонные артерии, соединяются, образуя головной круг, характерный для костных рыб. Сзади корни аорты сливаются и образуют спинную аорту, которая тянется под позвоночником, от нее отходят артерии к различным частям тела, а она переходит в хвостовую артерию.

От переднего конца тела венозная кровь собирается в парные передние кардиальные вены. Из хвостового отдела венозная кровь собирается в непарную хвостовую вену, которая, войдя в полость тела, разделяется на парные задние кардиальные вены; одна из них, проходя через левую почку, образует воротную систему. Передние и задние кардиальные вены, сливаясь, образуют два юовьевера протока, которые впадают в венозный синус. От желудка, селезенки, кишечника венозная кровь собирается в непарную подкишечную вену, входящую в печень; там эта вена распадается на сеть капилляров, образуя воротную систему

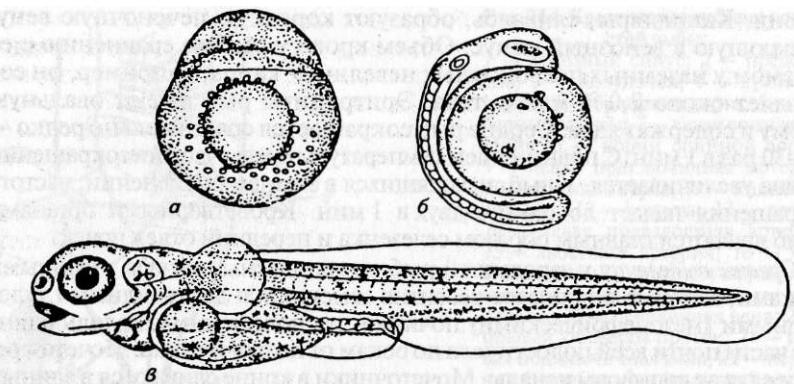
печени. Капилляры, сливаясь, образуют короткую печеночную вену, впадающую в венозный синус. Объем крови у рыб по сравнению с ее объемом у наземных позвоночных невелик. У карпа, например, он составляет около 2,0 % массы тела. Эритроциты рыб имеют овальную форму и содержат ядра. Сердце рыб сокращается сравнительно редко — 20—30 раз в 1 мин. С повышением температуры среды частота сокращений сердца увеличивается. У рыб, находящихся в зимнем оцепенении, частота сокращения падает до одного-двух в 1 мин. Кроветворными органами у рыб являются главным образом селезенка и передний отдел почек.

Органы выделения у зародышей рыб представлены сначала головными почками, которые в процессе развития заменяются лентовидными туловищными (мезонефрическими) почками, расположенными вдоль спинной части почти всей полости тела по бокам от позвоночника. Мочеточниками служат вольфовы каналы. Мочеточники в конце сливаются в единый проток, который открывается в мочевой пузырь или наружу. Конечным продуктом обмена у большинства костных рыб служит аммиак. Почки выполняют осморегуляторную функцию и участвуют в поддержании кислотно-щелочного равновесия. В выделении участвуют также жабры и кожа.

Органы размножения отличаются от органов размножения хрящевых рыб. Половые железы (яичники и семенники) парные (у окуня яичник непарный). У костистых рыб яйца выводятся образующимися яйцеводами, а мюллеровы каналы редуцируются. Яйцеводы одним концом срастаются с яичником, а другим открываются половым отверстием наружу. У некоторых представителей лососевых яйца выпадают из яичников в полость тела, а оттуда выходят наружу через особые половые поры на брюхе. Оплодотворение чаще наружное, у немногих видов — внутреннее. Икра (яйцеклетки) рыб мелкая, размером до нескольких миллиметров, покрыта студенистой оболочкой. Плодовитость рыб очень высокая. Большинство костных рыб раздельнополы, хотя среди них имеются и гермафродиты, например некоторые окунеобразные (*Labroides*).

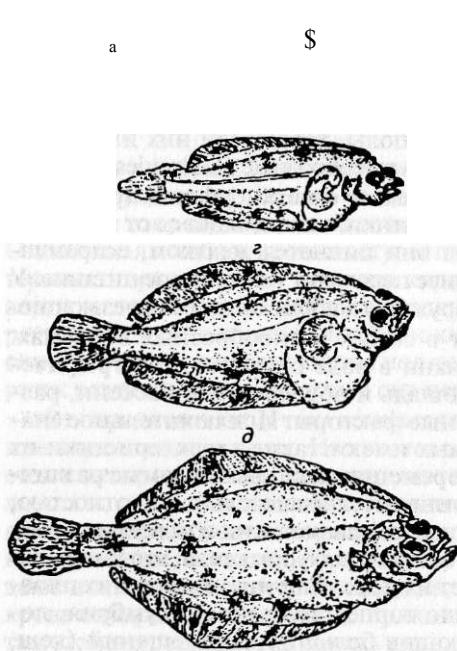
Развитие у большинства рыб протекает с метаморфозом (рис. 215). В этом случае из икринок выходят личинки, отличающиеся от взрослых особей рядом признаков. Сначала они питаются желтком, сохранившимся в желточном мешке, а позднее переходят на активное питание. У некоторых рыб личинки имеют наружные жабры, позднее исчезающие.

Экология рыб. Рыбы обитают в самых разнообразных водоемах. Большое влияние на условия жизни в воде оказывают географическое положение водоема, его площадь и водосборный бассейн, различные природные и хозяйствственные факторы. Исключительное значение для жизни водных животных имеют такие характеристики их среды обитания, как течения и перемешивание, вызываемые различными причинами. Водная среда отличается значительной плотностью, с чем связаны особенности морфологии, физиологии и поведения рыб. Для большинства рыб характерна обтекаемая форма тела; это уменьшает сопротивление воды и облегчает их перемещение. У хороших пловцов, обитателей толщи воды, тело торпедообразное (скумбрия, лосось, тунец). Рыбы, не совершающие больших перемещений (лещ,



а — икринка; б — зародыш; в — личинка с остатками желточного пузьря

сазан, карась) часто имеют высокое, сжатое с боков тело. У донных рыб (морской черт, скат) тело сжато в дорсовентральном направлении, что обеспечивает больший контакт с дном, где находится пища. Плоская форма тела характерна для камбалы, но у камбалы тело сильно сжато с боков, так что функциональной брюшной стороной является боковая (рис. 216). Можно еще отметить змеевидную, лентовидную, шаровидную и другие формы тела у рыб.



Движение рыб вперед обычно осуществляется за счет изгиба тела, в этом принимает участие и хвостовой плавник. Некоторые рыбы плавают с неподвижным телом за счет колебательных движений грудных плавников (скат), спинного (амия) или анального (электрический угорь) или одновременно и спинного, и анального (палтус). Основную функцию рулей

Рис. 216. Развитие камбалы:
а, б — мальки камбалы (имеют обычную для рыб форму тела); в — более взрослая рыбка (тело сплющено, но глаза расположены симметрично); г — еще более взрослая камбала (глаза смещаются на одну сторону); д — камбала, закончившая развитие

глубины выполняют грудные и брюшные плавники. Кожа рыб покрыта слизью, что уменьшает трение тела о воду и турбулентные завихрения. Скорость движения рыбы зависит от ее размеров, состояния, температуры среды и других факторов. Наибольшие скорости способны развивать тунцы, парусники, меч-рыбы; лосось плавает со скоростью около 5 м/с, голубая акула — 10 м/с, тунец — 20 м/с. Некоторые рыбы приспособились к своеобразному полету над водой (летучие рыбы).

Вода имеет особые свойства, обусловливающие неоднородность среды обитания рыб в различных водоемах. Вода относительно мало прозрачна, поэтому днем свет проникает только в верхние слои; чем глубже, тем становится темнее, а в океанских и морских глубинах царит полный мрак. Вода обладает большой теплоемкостью, но малой теплопроводностью. Верхние слои в водоеме могут хорошо прогреваться, в умеренных широтах это происходит летом. С увеличением глубины температура воды понижается, а на больших глубинах колебания температуры минимальны. Зимой в малых водоемах создается обратная слоистость — от 0 °С у ледового покрова до 4 °С в глубине. Рыбы являются пойкилотермными животными, т. е. животными с неизменной температурой тела, зависящей от температуры окружающей среды. Их активность напрямую зависит от температуры воды. В воде содержится меньше кислорода (примерно в 30 раз), чем содержится его в том же объеме воздуха. Диффузия кислорода, поступающего главным образом из атмосферы, происходит в воде крайне медленно, что определяет понижение его концентрации от поверхности в глубину (как летом, так и зимой). Рыбы могут обитать в водоемах с различным содержанием кислорода. У рыб (карп, карась, линь и др.), живущих в водоемах, бедных кислородом или с резкими колебаниями его концентрации, гемоглобин крови способен поглощать кислород даже при малом его количестве, тогда как у рыб (форель, сиг и др.), обитающих в воде, постоянно насыщенной кислородом, гемоглобин может связывать кислород только при значительном его количестве. По потребности в кислороде рыб можно разделить на четыре группы: требующие кислорода до 11 см³/л — форель, голян; до 7 см³/л — голавль, пескарь; до 4 см³/л — плотва, окунь; менее 3 см³/л и живущие даже при 1 см³/л — линь, карась. Как в морских, так и в пресных водах рыбы образуют более или менее обособленные группы, связанные с определенным местом обитания. При всем разнообразии условий в водной среде рыб можно разделить на несколько экологических групп:

- 1) морские — рыбы, постоянно живущие в морских водоемах (форель, луна-рыба);
- 2) пресноводные — рыбы, всю жизнь проводящие в пресных водоемах, таких как озера, пруды, реки (жерех, щука);
- 3) проходные — рыбы, совершающие протяженные нерестовые миграции из морских водоемов в пресные, или наоборот. Рыб, обитающих в море, но для размножения поднимающихся в реки, называют анадромными (лососевые, осетровые); рыб, живущих в пресных водоемах, но для нереста уходящих в море, — катадромными (угорь);

4) существуют также полупроходные рыбы; это рыбы, обитающие в пресных или морских водоемах, но способные жить в эстуариях рек (плотва, речная камбала).

Различаются рыбы и по характеру распределения в водоеме:

1) обитающие в толще воды, — пелагические, или нектонные. Это достаточно многочисленная группа рыб, которые прекрасно плавают и зачастую совершают продолжительные миграции. Одни из них питаются планктоном; это толстолобик, китовая акула, являющиеся мирными кочевниками-пастбищниками; другие поедают рыб — акула, судак; это активные охотники; третий ловят насекомых, упавших в воду и обитают в поверхностных водах. У большинства пелагических рыб окраска сверху, как правило, темная, а снизу светлая, серебристая;

2) обитающие на дне водоемов — донные, или бентосные, (в морях — скаты, камбалы, бычки, в пресных водах — карп, карась, сом, линь, стерлядь и др.). Они характеризуются меньшей подвижностью; это и мирные бентосоеды, и хищники. Одни из них плавают у дна, питаясь преимущественно донными животными, другие лежат на дне — тело у них зачастую сильно уплощено, и верхняя сторона имеет покровительственную окраску — под цвет грунта;

3) обитающие в прибрежной зоне — литоральные. Эта группа рыб характерна для морей и океанов. Многие из них являются донными и придонными рыбами. Это обычно плохие пловцы, не совершающие дальних перемещений, бентософаги, или хищники. Из морских рыб к ним относятся различные бычки, морские собачки и многие другие рыбы. Некоторые из них имеют особые приспособления, чтобы удерживаться на грунте во время прибоя, прилива или отлива.

Рыб, обитающих в реках, особенно в крупных, можно разделить на пять четко различающихся групп. Виды, обитающие в горной части реки — форель, голян; обитающие в предгорной части — хариус, елец; в равнинной части — голавль, сом; в нижнем течении — язь, красноперка; обитающие в устье, эстуарии — лещ, бычки.

Многие рыбы ведут оседлый образ жизни и перемещаются на незначительные расстояния; есть виды, кочующие постоянно (в разных направлениях или по определенному маршруту), а также совершающие разовые протяженные, в основном сезонные миграции. Большое число рыб перед нерестом совершают более или менее далекие миграции: из озер в реки, из глубин на мелководье, из низовьев в верховья рек и т. п. Проходные виды мигрируют из морей в реки или, что реже, из рек в моря. Морские рыбы совершают далекие путешествия с мест кормежки к местам икрометания. Различают нерестовые миграции, кормовые, зимовальные, а также суточные и сезонные.

Половозрелыми рыбы становятся в различном возрасте: гуппи — в 3—4 мес, хамса мечет икру впервые в годовалом возрасте, большинство карловых и окуневых рыб — в возрасте 3—4 лет, осетры и севрюги — на 10—12-м, а белуги на 14—17-м и даже на 20-м году жизни. По времени нереста можно выделить рыб, нерестящихся весной и ранним летом — сомовые, карловые; осенью и зимой — сиговые, тресковые; не имеющих

определенного сезона размножения — тропические виды. Большинство рыб наших пресных вод мечет икру в теплое время года, но щуки не-рестятся ранней весной, а налимы — зимой. К наступлению срока икрометания многие рыбы приобретают брачный наряд — изменяются окраска и форма их тела. Тихоокеанские лососи и некоторые другие виды рыб мечут икру только один раз в жизни и вскоре после этого погибают. Во время хода к местам нереста они перестают питаться, у горбуши во время хода на нерест окраска меняется с серебристой на коричневую с темными пятнами, на спине самцов вырастает горб, челюсти загибаются крючком.

У многих проходных рыб имеются яровая и озимая расы. Первые заходят в реки на нерест весной, вторые — осенью и остаются зимовать, нерестясь в следующем году.

Некоторые рыбы откладывают икру на дно или подводные предметы, к которым прикрепляют ее выделяемой клейкой слизью. Икра таких рыб — осетровых, многих карловых, окуневых — тяжелее воды. У ряда рыб икра легче воды и поэтому после откладки она поднимается в верхние слои водоемов (камбаловые, тресковые). Целый ряд видов проявляют заботу о потомстве. Некоторые рыбы откладывают икру в особое гнездо (колюшка), в мантинную полость моллюсков (горчак), другие закапывают ее в песок или гравий на дне (кета), третьи вынашивают ее во рту (тиляпия), в особых сумках на теле (самцы морского конька), что способствует большей выживаемости икры и мальков. Для большинства рыб характерно наружное оплодотворение. Способность к оплодотворению спермии самцов сохраняют недолго. У дальневосточных лососей (кета, горбуша) спермии сохраняют подвижность в течение 10—15 с; у осетровых (русский осетр, севрюга) — в течение 4—5 мин; у океанической сельди — до суток.

Плодовитость рыб связана с условиями развития икры и молоди. Наиболее плодовиты рыбы, откладывающие плавающую пелагическую икру, на втором месте стоят рыбы, откладывающие икру на растения, рыбы, охраняющие или прячущие икру, наименее плодовиты. Живородящие рыбы рождают лишь единицы или десятки мальков. Лососи мечут всего несколько тысяч икринок, развитие которых происходит в гнезде под слоем песка или гравия. Карловые рыбы, откладывающие икру на дно водоема или водные растения, в теплое время года продуцируют за один нерест до нескольких сотен тысяч икринок. Морские рыбы, дающие плавающую икру, выметывают за нерест миллионы икринок (крупная самка трески — до 9 млн, а луна-рыба — до 300 млн). Высокая плодовитость объясняется массовой гибеллю икры и выплывших из нее мальков. Порционность икрометания и растянутость периода размножения характерны для тропических рыб. В умеренных и холодных широтах большинство рыб выметывают икру единовременно. Плодовитость морских рыб обычно несколько выше, чем пресноводных и проходных.

Развитие икры рыб нашей фауны, нерестящихся весной и летом, длится обычно 3—8 сут (чем теплее вода, тем скорее), а у рыб, мечущих ее осенью и зимой (лососи, налимы и др.) затягивается до весны.

Для многих видов рыб характерен смешанный тип питания, но встречаются и достаточно специализированные виды. У большинства хорошо выражена возрастная, сезонная и даже суточная смена характера питания. Многие виды рыб способны к длительному голоданию. В зависимости от характера питания рот у рыб бывает различных типов: хватательный — когда челюсти вооружены острыми зубами для захвата и удержания добычи (щука); всасывающий, способный вытягиваться для всасывания пищи (лещ); дробящий — с тупыми, хорошо развитыми зубами, способными дробить твердую пищу (зубатка). По расположению различают рот нижний, расположенный на нижней стороне головы (характерен для рыб, добывающих пищу на дне водоема), конечный, расположенный на конце головы, и верхний, когда нижняя челюсть выдается за конец верхней (свойствен рыбам, которые питаются падающими в воду насекомыми и мелкими животными, обитающими в верхних слоях воды). Способы добывания пищи у различных рыб весьма разнообразны. Некоторые (щука) подкарауливают добычу, укрывшись в засаде; судак, жерех преследуют ее; сазан кормится на дне водоема; уклейка питается насекомыми, упавшими на поверхность воды; некоторые питаются планктоном, толстолобик, например, пропускает воду через жаберные щели, и отцеживает планктон жаберными тычинками; белый амур питается высшими растениями и т. д. Молодь рыб питается в основном зоопланктоном, а затем постепенно переходит на пищу, свойственную взрослым особям. В теплое время года рыбы питаются интенсивнее, чем зимой. Многие рыбы нашей страны зимой совсем не питаются, впадая в зимнее оцепенение. У многих рыб наблюдаются значительные миграции в поисках кормовых мест.

Хозяйственное значение рыб. Практическое значение рыб огромно. Это один из важнейших продуктов питания (рис. 217). За счет мяса рыбы мы получаем до 20 % белка животного происхождения. Некоторые виды рыб добывают для получения других видов продукции. Главным образом из отходов рыбной промышленности вырабатывают рыбную муку для свиней, пушных зверей и других сельскохозяйственных животных. Из печени некоторых видов (тресковых, акул) получают лечебный и технический жир. Из плавников акул, плавательного пузыря осетровых и других рыб вырабатывают ценный клей. Из чешуи ряда видов рыб изготавливают искусственный жемчуг. Все шире используются лекарственные препараты, получаемые, в частности, из тканей акул.

Россия принадлежит к числу стран с развитым рыбным промыслом. Основная масса рыбы вылавливается в морях и океанах различных частей земного шара. Море привычно считалось неисчерпаемым источником продовольствия. Однако расчеты биологов показывают, что в Мировом океане в общем можно ежегодно вылавливать не более 100 млн т рыбы, чтобы не нарушать биологического равновесия. В настоящее время подошли к такому пределу. Во всяком случае, в связи с сокращением запасов многих традиционных объектов лова (50 % вылова рыбы составляет 21 вид из почти 25 тыс. видов ихтиофауны) возможность их

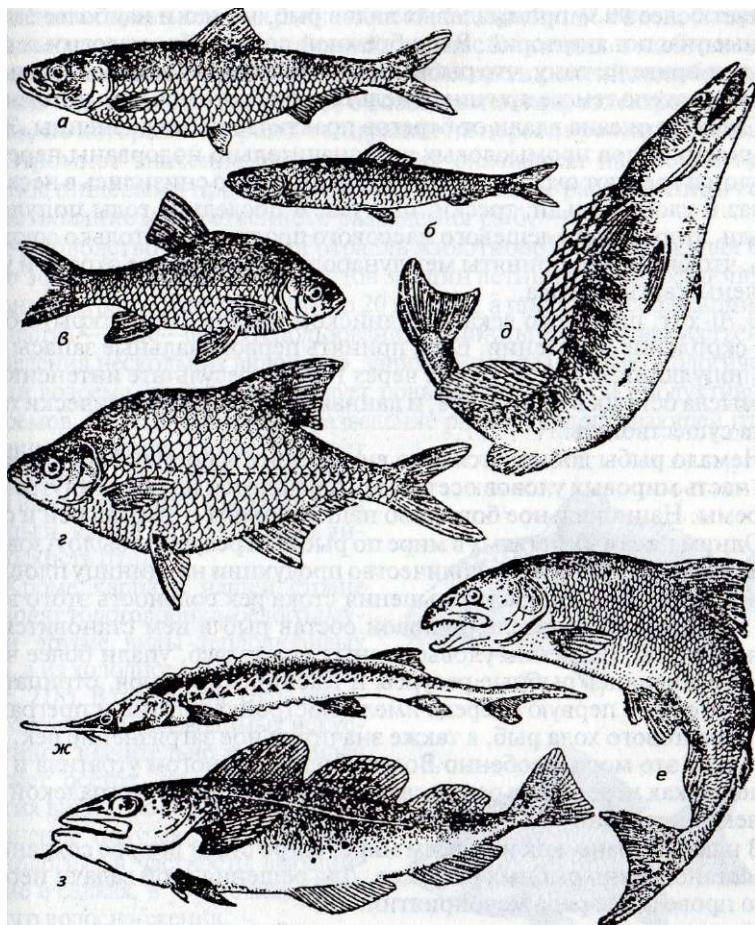


Рис. 217. Ценные объекты промысла:
а — сельдь-черноспинка; б — сельдь мурманская; в — вобла; г — лещ; д — судак; е — лосось; ж — севрюга; з — треска

потенциального использования снизилась на 15—20 %. В Атлантическом океане добыча рыбы начала снижаться уже с 1980 г.

Основные объекты промысла — различные виды трескообразных, сельдеобразных, окунеобразных. В настоящее время Мировой океан разделен на зоны рыболовства. По различным видам рыб устанавливаются квоты, лимиты вылова. В конце 80-х гг. были установлены 200-мильные охраняемые экономические зоны. Таким образом, весь прибрежный шельф оказался поделенным между различными государствами. Это сократило возможность свободного лова рыбы, так как здесь

обитает более 90 % промысловых видов рыб, но это и наиболее загрязненные участки акваторий. В прибрежной полосе сброс ядовитых сточных вод привел к тому, что рыбы иногда оказывались несъедобными. В таких местах по тем же причинам нельзя устраивать морские питомники.

Глубины океана вдали от берегов практически безжизненны. Запасы многих видов промысловых рыб значительно подорваны перепромыслом и требуют охранных мер; соответственно снизились в несколько раз и уловы сельди, трески, палтуса. В последние годы популяция сельди, этого самого дешевого массового продукта, настолько сократилась, что были предприняты международные меры по ее охране и установлены квоты вылова.

В 70-х гг. прошлого века в Индийском океане было открыто большое скопление нототении. Если принять первоначальные запасы данной популяции за 100 %, то уже через 10 лет в результате интенсивного промысла осталось всего 1,5 %, и данная популяция практически перестала существовать.

Немало рыбы добывается и во внутренних водоемах России. Большая часть мировых уловов осетровых приходится на наши внутренние водоемы. Национальное богатство нашей страны — это лососи и сиги.

Одним из самых богатых в мире по рыбным ресурсам было Азовское море, дававшее рекордное количество продукции на единицу площади. В настоящее время из-за уменьшения стока рек соленость этого водоема значительно возросла, видовой состав рыб в нем становится все беднее; уловы, особенно уловы ценных видов рыб, упали более чем в 15 раз. Снизились рыбные ресурсы и Каспийского моря, отрицательное значение в первую очередь имели постройка плотин и преграждение нерестового хода рыб, а также значительное загрязнение рек, впадающих в это море, особенно Волги. Волга во многом утратила и свое значение как нерестовая река в связи с постройкой Волгоградской гидроэлектростанции.

В нашей стране, как и во всем мире, остро стоит вопрос сохранения и восстановления рыбных ресурсов. Для решения этой задачи необходимо проведение ряда мероприятий:

1) охрана нерестилищ и производителей, охрана от вылова молоди и неполовозрелых особей, организация рационального рыболовства. Управление воспроизводством промысловых видов рыб;

2) предохранение водоемов от загрязнения промышленными стоками, нефтепродуктами и пр.;

3) акклиматизация ценных промысловых видов рыб в целях повышения эффективности использования кормовых запасов в водоеме и получения ценного продукта в большем количестве. Успешно была проведена акклиматизация азово-черноморской кефали в Каспии; сазана и сигов в ряде уральских озер; белого и пестрого толстолобиков, белого амура в южных районах европейской части России;

4) искусственное разведение ценных видов промысловых рыб. В этом деле достигнуты значительные успехи по разработке технологии размножения и выращивания, в частности в нашей стране, наибо-

лее ценных промысловых рыб: лососей, белорыбицы, сигов, осетровых и ряда карповых. Но предприятий, осуществляющих подобные мероприятия, явно недостаточно. Работа таких предприятий приобрела особое значение в связи, например, с возведением гидроэлектростанций на реках, по которым совершают миграции на нерест проходные рыбы.

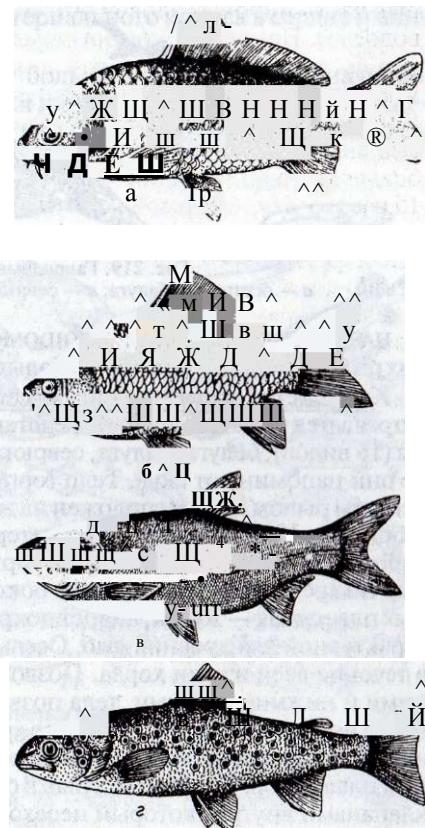
Огромное значение для улучшения снабжения населения нашей страны пищевыми продуктами имеет дальнейшее расширение и совершенствование искусственного разведения рыб.

Рыбоводство — выгодная отрасль животноводства, имеющая большую эффективность. С 1 га прудов за один летний период без дополнительных затрат можно получать до 20 ц рыбы, а при интенсификации производства — до 50—60 ц. Использование поликультуры, т. е. искусственного разведения нескольких видов рыб, занимающих различные экологические ниши водоема, также способствует повышению рыбной продуктивности водоемов. Можно совмещать разведение рыбы и водоплавающей птицы, например карпов и уток, что дает возможность получать повышенный выход не только мяса рыбы, но и птицы и, кроме того, увеличить яи-

Кроме карпа в нашей стране разводят толстолобика, белого амура и другие виды теплолюбивых рыб, а из холодноводных — форель (рис. 218). В последние годы видовой состав рыб, разводимых искусственно, значительно расширился за счет осетра, стерляди, сома и других видов. Все большее распространение получают интенсивные технологии, в частности выращивание в садках, в установках замкнутого водоснабжения.

ПОДКЛАСС ЛУЧЕПЕРЫЕ РЫБЫ (ACTINOPTERYGII). К подклассу лучеперых относится около 95 % известных рыб, объединенных в два надотряда. Характерной чертой этого подкласса является строение скелета парных плавников, образованных веерообразно расположенными хрящевыми или костными лучами.

Рис. 218. Рыбы — объекты рыбоводства:
а — карп; б — карась; в — толстолобик;
г — форель



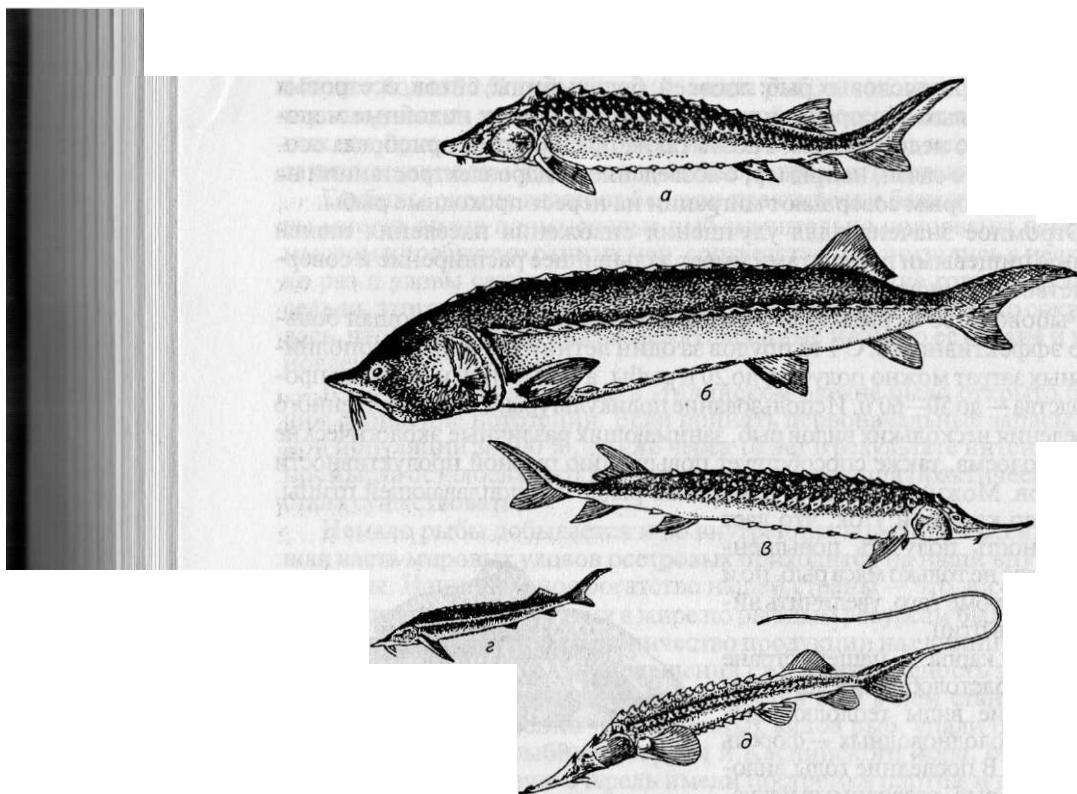


Рис. 219. Ганоидные (осетровые) рыбы:
а — осетр; б — белуга; в — севрюга; г — стерлядь; д — лопатонос

НДДОТРЯД ГАНОИДНЫЕ (GANOIDOMORPHA). Среди ныне живущих лучеперых рыб ганоидные выделяются большим числом архаичных черт строения.

К семейству осетровых (*Acipenseridae*) относится 23 вида, из них 11 — встречается в наших водах. Представителями осетровых являются осетры (16 видов), белуга, калуга, севрюга, стерлядь, лопатоносы и др. Внешне они напоминают акул. Тело торпедообразной формы. Голова с удлиненным рылом. Рот расположен на нижней стороне головы, выдвижной, зубов нет. Хвостовой плавник гетероцеркальный (неравнолопастной), парные плавники расположены горизонтально (рис. 219). У осетровых рыб по хребту, бокам и границе боков и брюха тянутся пять рядов костных пластинок — жучек, а хвост покрыт ромбической ганоидной чешуйей, характерной для древних рыб. Осевым скелетом служит сохраняющаяся в течение всей жизни хорда. Позвонки представлены хрящевыми верхними и нижними дугами, тела позвонков не развиты. Череп, позвоночник и пояса конечностей в основе хрящевые, но хрящевой череп покрыт накладными плоскими костями. Кости входят также в состав пояса грудных плавников. У осетровых рыб в сердце имеется артериальный конус с клапанами внутри, который переходит в брюшную аорту. В кишечнике

находится спиральный клапан. Созревшие яйца также, как и у акул и скатов, выпадают из яичников в полость тела и увлекаются движением ресничек в воронки яйцеводов, которыми являются мюллеровы каналы.

Большинство осетровых обитает в морях; основные запасы сосредоточены в Каспийском море. Но некоторые виды, такие как стерлядь, лопатоносы, постоянно живут в реках и озерах. В последние годы численность осетровых повсеместно снизилась. Некоторые виды являются объектами аквакультуры. Все виды осетровых нерестятся в реках весной и летом. У многих видов проходных осетровых выражены две расы — озимая и яровая. Большинство питается различными донными животными. Мясо и икра этих рыб высоко ценятся.

НАДОТРЯД КОСТИСТЫЕ (TELEOSTEI). К этому надотряду относится большинство лучеперых рыб. Форма тела разнообразна. Тело обычно покрыто костной чешуйей, имеющей вид тонких, налегающих друг на друга пластинок. Верхняя и нижняя лопасти хвоста примерно одинаковой величины и формы (равнолопастный хвост). Грудные плавники обычно вертикальные. Скелет костный. Хорда у взрослых особей в той или иной степени редуцирована. Артериального конуса в сердце и спирального клапана в кишечнике нет.

Многие костиные рыбы имеют большое промысловое значение. Это прежде всего рыбы, относящиеся к семействам сельдевых, лососевых, карповых, сомовых, щуковых, угревых, окуневых, камбаловых и тресковых (рис. 220).

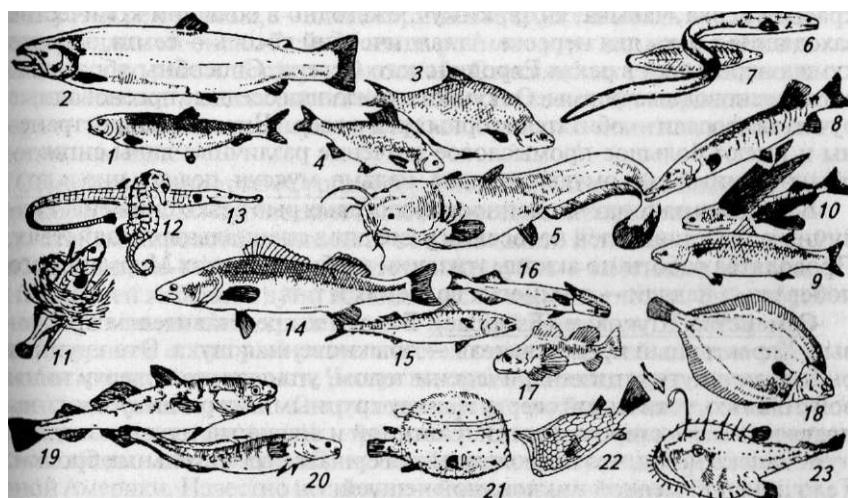


Рис. 220. Костиные рыбы:

/ — сельдь-черноспинка; 2 — обыкновенный лосось; 3 — лещ; 4 — плотва; 5 — сом; 6 — угорь; 7 — личинка угря; 8 — щука; 9 — кефаль; 10 — летучая рыба; 11 — самец колюшки (в гнезде); 12 — морской конек; 13 — морская игла; 14 — судак; 15 — скумбрия; 16 — рыба-прилипало; 17 — бычок колючий; 18 — камбала; 19 — треска; 20 — налим; 21 — иглобрюх; 22 — кузовок; 23 — морской черт

Семейство Сельдевые (Clupeidae). Тело вытянутое, стройное, слабо сжатое с боков, невысокое. Спина темно-синяя или зеленоватая, бока серебристые. Брюшные плавники расположены посередине брюха, над ними находится один спинной плавник. Боковая линия внешне не заметна. На брюшной стороне имеется острый киль из чешуек. Зубы слабые, или их нет. Сельдевые — стайные рыбы, зачастую совершающие продолжительные миграции (атлантическая сельдь). Наибольшее число видов обитает в тропиках.

Питаются сельдевые планктонными животными, пользуясь цедильным аппаратом из жаберных тычинок. Одни из них, как например атлантическая, тихоокеанская сельди, кильки, шпроты, сардины и др., всю жизнь проводят в море. Другие (некоторые сельди Каспийского, Азовского и Черного морей) на нерест поднимаются в реки (пузанок, черноморская, кесслеровская сельди). Сельдевые имеют большое промысловое значение (атлантическая, иваси, килька, сардина, пузанок).

Семейство Лососевые (Salmonidae). Проходные и пресноводные рыбы северного полушария. По внешнему виду сходны с сельдями. Отличаются жировым плавничком на хвостовом стебле. Тело покрыто чешуей, боковая линия хорошо заметна и образует полоску прободенных чешуй. У собственно лососей рот большой с зубами, чешуя мелкая; у сигов рот маленький беззубый, чешуя крупная. Лососевые — очень ценные промысловые рыбы с высокими вкусовыми качествами мяса и икры. Наибольшее промысловое значение имеют проходные тихоокеанские (дальневосточные) лососи рода *Oncorhynchus* — кета, горбуша, красная нерка, чавыча, сима, кижуч, ежегодно в большом количестве заходящие в реки для нереста. Атлантический лосось — семга, а также кумжа нерестится в реках Европейского Севера. Способны образовывать пресноводные формы. От кумжи произошли оседлые пресноводные ручьевые форели — обитатели горных рек и озер. Широко распространены и имеют большое промысловое значение различные виды сигов — обыкновенный сиг, омуль, ряпушка, нельма, муксун, пелядь, чир и др.

Во многих районах численность лососевых рыб резко снижена. Различных представителей лососевых разводят в специальных хозяйствах. Проводятся работы по акклиматизации горбуши в реках Мурманского побережья, пеляди — во многих водоемах и т. п.

Семейство Щуковые (Esocidae). Близки к представителям лососевых. Характерный представитель — обыкновенная щука. Это крупная рыба с вытянутым цилиндрическим телом, уплощенной сверху головой с пастью, усаженной сверху и снизу крупными острыми зубами, направленными в сторону глотки. Спинной и анальный плавники сдвинуты далеко назад, что позволяет ей совершать стремительные броски. Тело покрыто мелкой циклоидной чешуей.

Щуки широко распространены по рекам и озерам нашей страны (на Дальнем Востоке обитает амурская щука). Предпочитают реки с медленным течением и проточными озера с заливами, заросшими камышом и водной растительностью. Кормятся щуки по утрам и под вечер. Это прожорливые, сильные хищники, питающиеся в основном другими ры-

бами. Хищничать щука начинает очень рано, уже при длине 12—15 мм; достигнув длины 5 см, она почти полностью переходит на питание рыбой. Наблюдаются случаи каннибализма. В некоторых водоемах из рыб обитают только щуки. Здесь наблюдается цепочка последовательного каннибализма, когда более крупные щуки питаются более мелкими сородичами. Отличаются большой плодовитостью и долголетием. Местами щук добывают в большом количестве, мясо содержит мало жира.

Семейство Угревые (Anguillidae) представлено в нашей фауне только речным (европейским) утром. Эта рыба отличается длинным змееобразным телом без брюшных плавников, покрытым очень мелкой чешуй. Голова маленькая, коническая. Спинной и анальный плавники слиты с хвостовым. Рот вооружен мелкими зубами. Питается беспозвоночными и мелкой рыбой. Активен ночью, днем чаще всего зарывается в илистый грунт. Взрослые самки угрей обитают во многих пресных водоемах бассейна Балтийского и реже Баренцева и Черного морей. Самцы держатся в море. Живут 6—12 лет, достигая длины 2 м и массы 5 кг, но это бывает редко. Нерестится раз в жизни. Взрослые самки спускаются по рекам к устьям (могут переползать в росистые ночи из водоема в водоем), где к ним присоединяются самцы, и они совместно мигрируют в Саргассово море (Атлантический океан), там и проходит нерест. Мигрирующие уги должны проплыть 4—7 тыс. км. Нерест проходит на глубине около 1000 м. После нереста самки и самцы погибают. Личинки — лептоцефалы — пассивно переносятся течением Гольфстрим, которое здесь начинается, через океан к берегам Европы. В устья рек через 2,5—3 года приплывает уже молодь — стеклянные уги длиной около 7 см. Молодь речного угря входит в реки весной и широко расселяется по водоемам. Стеклянных угрей интенсивно отлавливают в целях расселения и искусственного выращивания. Угорь — ценная промысловая рыба, мясо высоко ценится, особенно в копченом виде.

Семейство Карповые (Cyprinidae). Внешне похожи на сельдевых, но отличаются от них хорошо заметной боковой линией, а от лососевых — отсутствием жирового плавника. Рот выдвижной и лишен зубов, пища дробится одним—тремя рядами глоточных зубов — костными выростами последней жаберной дуги, вдающимися в полость глотки. Зубы обращены вершинами к находящемуся на верхней поверхности глотки жерновку, трением о который размельчается пища. Усиков одна-две пары, или они могут отсутствовать. Чешуя циклоидная или отсутствует, плавники с мягкими лучами. Открытопузирные рыбы, плавательный пузырь большой, состоит из двух - трех камер. Семейство карповых — самое богатое видами семейство рыб, населяющих воды Евразии, Африки и Северной Америки. Известно несколько десятков родов и около 1,5 тыс. видов в основном пресноводных рыб; в водоемах России обитает около 80 видов, среди них густера, золотой карась, пескарь, верховка, уклейка, чехонь, подуст, елец, гольян, красноперка, рыбец, шемая.

Большинство карповых — теплолюбивые рыбы. Осеню они постепенно прекращают питаться и на зиму впадают в оцепенение. Размно-

жаются в конце весны и летом. У серебряного карася есть популяции, где отсутствуют самцы. Когда самки нерестятся, то развитие икры идет при наличии спермиев других близких видов рыб (золотой карась, линь, сазан). Проникая в яйцеклетку, они не оплодотворяют ее, а только стимулируют развитие (гиногенез). Из икринок развиваются только самки. Мальки карповых питаются обычно планктоном. Взрослые особи поедают преимущественно различных донных беспозвоночных животных и водные растения, хотя среди карповых имеются планктоноядные (толстолобик) и хищные виды (жерех).

Карповые рыбы служат важнейшими объектами рыбного промысла наших внутренних водоемов. Наибольшее промысловое значение из проходных форм имеют подвиды плотвы — вобла и тарань, а из пресноводных — сазан, карась, лещ, щипак, жерех, голавль, линь, усач и др. Основным объектом промышленного разведения является карп (одомашненная форма сазана). Разводят также толстолобиков, белых амуров и другие виды. К карповым относятся такие объекты аквариумистики, как пунтиусы, расборы, данио и золотые рыбки (комета, шубункин, вуалехвост, телескоп и др.); все они — результат одомашнивания и селекции серебряного карася.

Семейство Сомовые (Siluridae). Обитают в пресных водах Европы и Азии, отсутствуют в бассейне Северного Ледовитого океана. В европейской части РФ широко распространен обыкновенный (европейский) сом. Отличается коротким вальковатым туловищем, лишенным чешуи, сплюснутым с боков хвостом с очень длинным анальным плавником, слегка приплюснутой сверху крупной головой с большой пастью и усами. Зубы хорошо развиты. Живут более 30 лет, достигая 300—400 кг при длине тела 5 м. Озерные сомы всегда мельче речных. Активны обычно в вечернее время и рано утром. Ведут оседлый образ жизни, предпочитая держаться в омутах. Зимой лежат в ямах. Всеядные хищники, жидают иногда в лиманах. Растут быстро — к 5-летнему возрасту достигают нередко 1 м. Кожа раньше использовалась вместо стекол — так называемый рыбий пузырь. Является объектом аквакультуры. В бассейне Амура обитают более мелкий амурский сом и сом Солдатова.

Семейство Окуневые (Percidae) объединяет большую группу морских и пресноводных закрытопузирных рыб, у которых в плавниках имеются колючие твердые лучи. Спинных плавников два (колючий и мягкий) или, что реже, один, состоящий из колючей и мягкой частей. В анальном плавнике два колючих луча. Рот большой с зубами, у некоторых есть клыки. Кости жаберной крышки зазубренные. К этому семейству относятся окунь, ерши, судаки, морские судаки и другие виды. Многие из них являются объектом промысла. В наших водоемах объектом промысла является обыкновенный судак, достигающий 12 кг и более. Обычны окунь, которые ведут оседлый образ жизни и могут достигать 1—1,5 кг.

Семейство Бычковые (Gobiidae) отличается от других тем, что у его представителей брюшные плавники срастаются, образуя присасывающую воронку, с помощью которой онидерживаются на дне. Мор-

ские и реже речные рыбы. Многие виды являются объектом промысла. Самый многочисленный в наших водоемах — бычок-кругляк, ценный промысловый объект.

Семейство Камбаловые (Pleuronectidae) — морские рыбы с уплощенным с боков телом, окаймленным спинным и анальным плавниками. Ведут донный образ жизни, лежа на одном боку, причем этот бок светлый, а верхний — темный. Оба глаза у взрослых рыб смешены на одну, обычно правую, сторону головы. Плавательный пузырь отсутствует. Обитают в прибрежных водах до глубины 1 тыс. м. Питаются рыбами и донными беспозвоночными. Икра пелагическая, развивается в верхних слоях воды. Мальки камбалы имеют симметричное тело и глаза по обе стороны головы. Подрастая, постепенно опускаются в более глубокие слои воды, тело их уплощается, один глаз перемещается на другую сторону головы, они начинают плавать на боку и переходят к донному образу жизни. Различные виды камбал (желтоперая, морская, полярная) и палтусов (атлантический белокорый может достигать в К длину 4,5 м и массы 300 кг) являются важными объектами тралового промысла.

Семейство Тресковые (Gadidae). Холоднолюбивые морские рыбы, широко распространенные в арктических водах. Тело покрыто чешуей, на подбородке находится непарный усик, брюшные плавники расположены впереди грудных, имеют (кроме налинов) три спинных плавника и два анальных. Лучи всех плавников мягкие. Многие совершают длительные миграции и образуют большие скопления. К семейству относятся треска, пикша, минтай, сайды, навага. Лишь речной налим обитает в пресных водах, и это единственный вид из наших пресноводных рыб, нерестящийся зимой (декабрь—январь). Тресковые являются важными объектами промысла. Атлантическая треска — самая крупная (до 40 кг) и в уловах наиболее многочисленная из тресковых. Обитает в морях Европейского Севера и Дальнего Востока. Один из ее отличительных признаков — изогнутая белая боковая линия. В Баренцевом море промышляют пикшу (боковая линия у нее черная, прямая). Объектами промысла являются навага (небольшая рыба длиной около 30 см), минтай, путассу. У тресковых ценятся мясо и жирная печень, из которой приготовляют консервы и медицинский рыбий жир.

ПОДКЛАСС ЛОПАСТЕПЕРЫЕ РЫБЫ (SARCOPTERYGH). Лопастеперые известны с середины раннего девона. Они сочетают в своем строении архаичные и прогрессивные черты. Уже в начале формирования этой группы рыб их специализация пошла по двум разным направлениям. Чешуя космоидная или костная. В течение всей жизни сохраняется хорда, окруженная плотной соединительнотканной волокнистой эластичной оболочкой. Парные плавники с покрытой чешуей мясистой лопастью у основания. У двоякодышащих и кистеперых рыб (рис. 221) передний мозг крупнее остальных отделов; он разделяется на правое и левое полушария. В сердце имеется артериальный конус, а в кишечнике — спиральный клапан. Как выросты брюшной стороны начального отдела пищевода образуются один-два пузыря, выполняющие функции

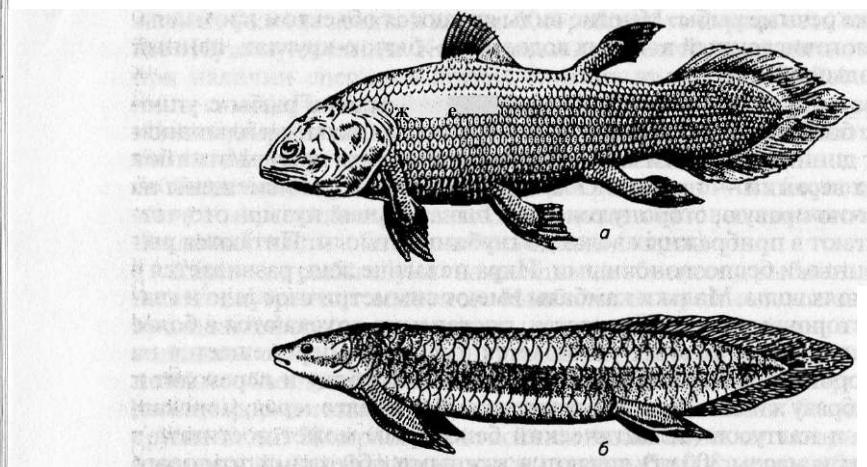


Рис. 221. Кистеперые и двоякодышащие рыбы:
а — латимерия; б — неоцератод

цию легких. У кистеперых (ископаемых) и двоякодышащих (кроме рогозуба) плавательный пузырь — легкое — парный. Он снабжается кровью по ответвлению последней (пятой) выносящей жаберной артерии и в течение всей жизни сохраняет связь с пищеводом, открываясь на его брюшной стороне щелевидным отверстием, снабженным замыкающим мускулом; у глубоководной латимерии легкие сохраняются в видеrudimenta. Стенки легких ячеистые, что увеличивает их внутреннюю поверхность. Они функционируют как добавочные органы дыхания, а у зарывшихся в грунт при засухе протоптеров и лепидосиренов — как основной орган дыхания.

НАДОТРЯД ДВОЯКОДЫШАЩИЕ (DIPNOI). Двоякодышащие рыбы — очень древняя группа. По-видимому, это боковая, сильно специализированная ветвь, обособившаяся от каких-то примитивных кистеперых еще в нижнем девоне. Скелет двоякодышащих в основном хрящевой. Хорда сохраняется в течение всей жизни. Чешуя космоидная или kostная. Передний мозг образует полушария. Зубы обычно сливаются в две-три пары мощных зубных пластинок. Имеются сквозные ноздри. У современных видов есть одно или два легких, соединяющихся с брюшной стороной начального отдела пищевода, есть клоака. В связи с наличием легких у них имеется два круга кровообращения. В предсердии присутствует неполная перегородка, продолжающаяся и в желудочке. Таким образом, у двоякодышащих появляются изменения, которые напоминают особенности кровеносной системы земноводных. Мочеполовая система близка к таковой хрящевых рыб и амфибий.

Двоякодышащие рыбы, впадающие в оцепенение при высыхании водоемов (протоптер), в активном состоянии выделяют аммиак, а в оцепенении — мочевину, накапливающуюся в организме.

В течение долгого времени двоякодышащие были известны только по окаменелым остаткам, и только в 1835 г было установлено, что обитающий в Африке протоптер — двоякодышащая рыба. Всего же, как оказалось, до наших дней дожили представители шести видов этой группы: австралийский рогозуб (*Neoceratodus forsteri*) из отряда однолетючих, американский чешуйчатник (*Lepidosiren paradoxa*) — представитель отряда двулегочных и четыре вида африканского рода *Protopterus*, также из отряда двулегочных. Все они, видимо, как и их предки, пресноводные рыбы. Обитают в пересыхающих водоемах. Когда водоемы пересыхают полностью, протоптеры и лепидосирены зарываются в ил и впадают в состояние оцепенения, дыша воздухом с помощью легких. В природе такая спячка обычно длится около 6 мес. Неоцератоды при полном пересыхании водоема погибают

НАДОТРЯД КИСТЕПЕРЫЕ (CROSSOPTERYGI). Палеонтологи давно обнаружили в древних слоях осадочных пород, относящихся к палеозойской эре развития Земли, останки своеобразных рыб с особым строением парных плавников и рядом других, как правило, примитивных признаков. У кистеперых рыб основанием каждого из парных плавников служит мясистая, вытянутая, покрытая крупной чешуей лопасть, на конце центральной оси которой располагаются лучи плавника. Расположение скелетных элементов внутри основной лопасти плавника несколько напоминает расположение костей пятипалых конечностей наземных позвоночных. К тому же ископаемые палеозойские представители этого надотряда обладали помимо жаберного также легочным дыханием, легкие возникли как мышцы брюшной стороны кишечника. Имелись внутренние ноздри — хоаны. Все это позволяет предполагать близость древних кистеперых рыб к предкам четвероногих наземных позвоночных.

Их считали вымершими миллионы лет назад. Первую рыбку поймали в 1938 г. Позднее было добыто еще несколько особей этой замечательной рыбы (всего менее ста экземпляров), получившей название латимерия (*Latimeria*); оказалось, что они сохранились на небольшой акватории у трех Коморских островов в западной части Индийского океана.

По морфологическим особенностям латимерия, видимо, мало отличается от ископаемых целакантов мезозоя. Тело покрыто космоидной чешуей. Хвост дифицеркальный с дополнительной средней лопастью. Плавники с мощными, но довольно короткими основаниями и удлиненными лопастями. Головной мозг занимает не более 1/100 объема мозговой коробки, заполненной в основном жироподобной массой. У них сохраняется хорда, в сердце — артериальный конус, в кишечнике — спиральный клапан. Хоаны отсутствуют. Плавательный пузырь подобно легким двоякодышащих рыб отходит от брюшной стороны начальной области пищевода, заметно редуцирован и имеет вид трубки длиной 5—8 см, переходящей в окруженный жиром тяж. У половозрелой самки массой 78 кг в правом яичнике (левыйrudimentарен) было обнаружено 19 икринок диаметром 8—9 см и массой около 300 г каждая; яйцекивородящи.

В ходе наблюдений выяснилось, что латимерия ведет ночной образ жизни, опускаясь для охоты на глубину до 700 м и более. С наступле-

нием дня рыбы возвращаются в подводные пещеры, расположенные на глубине 150—200 м.

И еще одно совершенно невероятное событие: в 1998 г. в 10 тыс. км от Коморских островов в Целебесском море у берегов Индонезии также была выловлена латимерия. Это была рыба средних размеров — длиной 124 см и массой 29,2 кг (наиболее крупные особи с Комор достигают в длину 180 см и весят около 95 кг). Внешне она ничем не отличалась от представителей коморской популяции, только цвет ее тела был не синевато-стальной, а коричневый. Выполнив анализ митохондриальной ДНК, американские исследователи пришли к выводу, что расхождение между двумя популяциями началось 5—7 млн лет назад и индонезийский целакант, скорее всего, может рассматриваться как новый вид.

НАДКЛАСС ЧЕТВЕРОНОГИЕ, ИЛИ НАЗЕМНЫЕ ПОЗВОНОЧНЫЕ (*Tetrapoda*)

Надкласс объединяет позвоночных, которые освоили наземную среду обитания. В строении этих животных прослеживаются изменения, отражающие приспособленность к наземной, воздушно-сухой среде. В процессе эволюции представители данной группы приобретали способность дышать атмосферным воздухом, частично или полностью утратив связь с водной средой, а иногда вторично перейдя к водному образу жизни; при этом у них сохранились легочное дыхание и другие признаки наземных позвоночных (рис. 222).

Качественно новые условия обитания, действие гравитации, неоднородность среды обитания и ее разреженность, более разнообразное и широкое воздействие абиотических факторов привели к значительным морфологическим и физиологическим изменениям организма в целом.

В этих условиях получили развитие парные конечности, которые стали служить для передвижения и опоры тела (рис. 223). Само тело на-

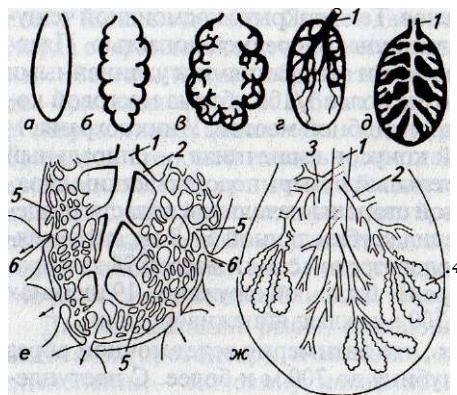


Рис. 222. Схема строения легких у позвоночных:

a, б — хвостатые амфибии; *в* — бесхвостые амфибии; *г* — чешуйчатые рептилии; *д* — крокодилы и черепахи; *е* — птицы; *ж* — млекопитающие; *1* — бронхи; *2,3* — брюшные и спинные ветви бронха; *4* — легочные альвеолы; *5* — легочные трубочки; *6* — сообщение легких с воздушными мешками

Рис. 223. Скелет передних конечностей различных позвоночных животных (гомологичные органы):
а — лягушка; б — ящерица; в — птица; г — обезьяна; д — лошадь; е — кит; ж — кошка;
и — летучая мышь

земных позвоночных стало более дифференцированным, например выделился шейный отдел. Увеличение подвижности головы позволило более эффективно использовать органы чувств и лучше ориентироваться в окружающей обстановке. Это дало возможность, оставаясь самому неподвижным и менее заметным, обследовать окружение за счет движения только головы; отражать нападение и захватывать пищу в разных направлениях.

Воздействие гравитации и других абиотических факторов привело к усилению мышечной активности и повышению активности самих животных. Повышенная активность требовала дополнительного поступления энергии и, как следствие, интенсификации обмена веществ, что привело к качественным изменениям строения и функционирования всех систем организма. Эта группа животных включает земноводных, пресмыкающихся, птиц и млекопитающих.

КЛАСС ЗЕМНОВОДНЫЕ, ИЛИ АМФИБИИ (*Amphibia*)

Немногочисленная группа первых наиболее примитивных наземных пойкилотермных позвоночных (около 4,5 тыс. видов), сохранивших значительную связь с водной средой. У большинства яйца не имеют плотных оболочек и могут развиваться только в воде. Характерно легочное дыхание, два круга кровообращения и парные пятипалые конечности. Большинство обитает в зависимости от стадии жизненного цикла то в воде, то на суше. Личинки ведут водный образ жизни. В течение жизни они претерпевают метаморфоз, превращаясь из чисто водных организмов во взрослые формы, обитающие большей частью **вне** воды. Однако у взрослых особей степень приспособления к жизни **на** суше в общем невелика.

Амфибии являются первичноназемными позвоночными, предки которых жили в воде. В эпидермисе кожи этих животных имеется большое число многоклеточных слизистых желез. Череп соединяется с единственным шейным позвонком двумя мышцами, крестец также образован одним позвонком. Конечности, хотя и построены по типу пятипалых, развиты слабо и не могут удержать тело в приподнятом положении. Ноздри сквозные, носовая полость сообщается с ротовой внутренними ноздрями — хоанами, среднее ухо с одной слуховой косточкой — стременем. Передний мозг имеет два полушария. Легкие развиты слабо, в качестве дополнительного органа дыхания достаточно большую роль играет еще и кожа. Органы дыхания личинок — жабры, а взрослых — легкие. Имеются два круга кровообращения. Сердце трехкамерное и состоит из двух предсердий и одного желудочка с артериальным конусом. Трехкамерное сердце не обеспечивает полного разделения артериальной и венозной крови, поэтому в большей части тела по артериям течет смешанная кровь. Почки туловищные.

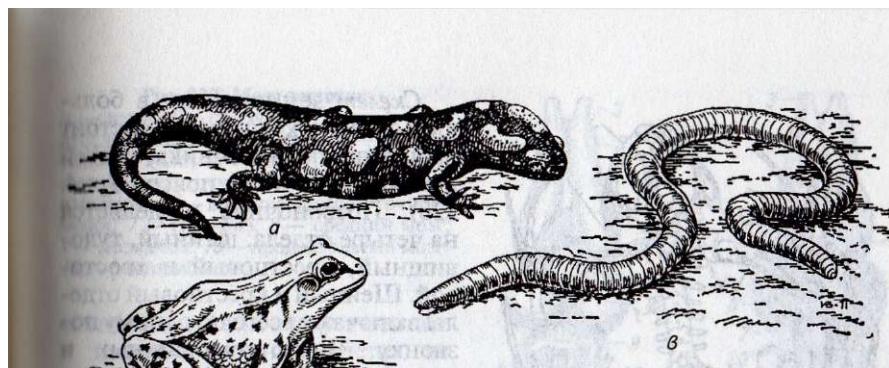
Основная масса представителей размножается в воде. Оплодотворение наружное, развитие с метаморфозом. Взрослые земноводные после метаморфоза становятся наземными дышащими легкими животными с двумя кругами кровообращения. Только немногие земноводные проводят всю жизнь в воде, сохраняя жабры и некоторые другие личиночные признаки.

Амфибии служат цennыми объектами лабораторных экспериментов. При их изучении было сделано много выдающихся открытий. Так, И. М. Сеченов в опытах на лягушках открыл рефлексы головного мозга. Земноводные интересны как животные, филогенетически связанные, с одной стороны, с древними рыбами, а с другой — с примитивными пресмыкающимися.

Наибольшее число видов обитает в регионах с теплым, влажным климатом. К северу и югу от экватора число видов земноводных и их численность снижаются. Не встречаются они и в соленных водоемах.

Строение и жизненные отправления. Внешний вид земноводных разнообразен (рис. 224). У хвостатых амфибий тело удлиненное, ноги короткие, примерно одинаковой длины, всю жизнь сохраняется длинный хвост. У бесхвостых амфибий тело короткое и широкое, задние ноги прыгательные, значительно длиннее передних, хвост у взрослых особей отсутствует. У безногих амфибий тело длинное червеобразное, без ног. У всех амфибий шея не выражена или выражена слабо. Поэтому можно выделить так же, как и у рыб, три отдела — голова, туловище, хвост. В отличие от рыб голова у них сочленяется с позвоночником подвижно.

Покровы. Кожа земноводных тонкая, голая, обычно покрыта слизью, выделяемой многочисленными кожными железами (рис. 225). У личинок слизистые железы одноклеточные, у взрослых — многоклеточные. Выделяемая слизь смачивает кожу, что важно для кожного дыхания. Слизь обладает бактерицидными свойствами, у



б

Рис. 224. Представители разных отрядов земноводных:
а — огненная саламандра; б — травяная лягушка; в — червяга

некоторых земноводных секрет кожных желез ядовит и может быть смертельно опасен даже для человека. Степень ороговения эпидермиса у разных видов земноводных далеко не одинакова. У личинок **и** тех видов, которые ведут в основном водный образ жизни, орогование поверхностных слоев кожи развито слабо, но у жаб ороговевшей может быть до 60 % всей поверхности кожи на спинной стороне.

Кожа земноводных как бы свободно накинута на тело (особенно это выражено у бесхвостых), прикрепляясь в отдельных точках и образуя пазухи и полости, заполняемые водой и лимфой.

Кожа — важный орган дыхания земноводных, о чем свидетельствует отношение длины капилляров кожи к длине этих сосудов в легких; у тритона оно равно 4:1, а у жаб, имеющих более сухую кожу, **1 : 3**.

Окраска земноводных обычно носит покровительственный характер. Некоторые, как, например, древесная квакша, способны изменять ее, у ядовитых представителей она яркая, предупреждающая.

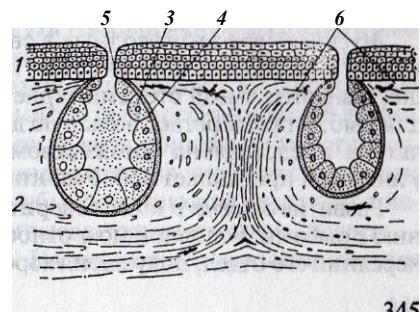


Рис. 225. Схема строения кожи лягушки (поперечный срез):

1 — эпидермис; 2 — дерма; 3 — железистые клетки кожной железы; 4 — мускульный покров железы; 5 — выводной проток кожной железы; 6 — пигментные клетки

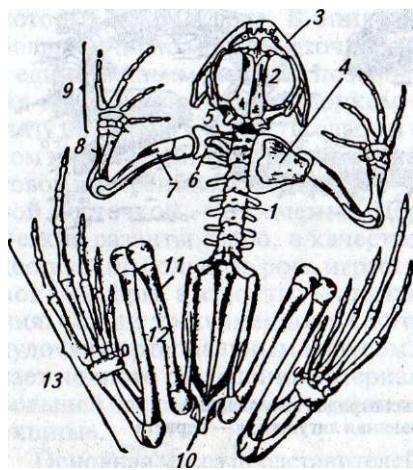


Рис. 226. Скелет лягушки:
 1 — позвоночник; 2 — черепная коробка;
 3 — челюсти; 4 — лопатка; 5 — ключица;
 6 — коракоид (воронья кость); 7 — плечевая кость; 8 — кость предплечья; 9 — кости кисти; 10 — таз; 11 — бедренная кость; 12 — кость голени; 13 — кости стопы

вичных) костей. С переходом от жаберного к легочному дыханию водных предков амфибий к легочному висцеральный скелет изменился. Подъязычный аппарат состоит из элементов скелета черепа. Верхняя часть подъязычной дуги — подвесок, к которому у низших рыб прикрепляются челюсти, у амфибий в связи со срастанием первичной верхней челюсти с черепом превратилась в маленькую слуховую kostochku — стремя, расположенную в среднем ухе. Характерно широкое основание черепа, что повышает эффективность как захвата пищи, так и дыхания.

Скелет конечностей и их поясов складывается из элементов, характерных для пятипалых конечностей наземных позвоночных животных. Пояс передних конечностей лежит свободно в толще мускулатуры. Число пальцев на ногах неодинаково у разных видов.

Мускулатура земноводных в связи с более разнообразными движениями и развитием конечностей, приспособленных к движению по суше, в значительной степени теряет метамерное строение, характерное для рыб, и приобретает большую дифференцировку. Скелетная мускулатура представлена множеством отдельных мышц, число которых у лягушки превышает 350. Развита мощная мускулатура конечностей.

Развитие головного мозга говорит о значительном прогрессе по сравнению с рыбами. Головной мозг относительно крупнее (рис. 227), особенно передний его отдел, полушария хорошо выражены и разделены. Прогрес-

Скелет земноводных в большей степени хрящевой и состоит из черепа, позвоночника, костей конечностей и их поясов (рис. 226). Позвоночник разделяется на четыре отдела: шейный, туловищный, крестцовый и хвостовой. Шейный и крестцовый отделы включают всего по одному позвонку. Число туловищных и хвостовых позвонков различно. У бесхвостых амфибийrudименты хвостовых позвонков срастаются в длинную kostochku — уrostиль. У некоторых хвостатых земноводных позвонки двояковогнутые, и между ними сохраняются остатки хорды. У большинства же амфибий они либо выпуклые спереди и вогнутые сзади, либо наоборот — вогнутые спереди и выпуклые сзади. Ребра отсутствуют, соответственно отсутствует и грудная клетка.

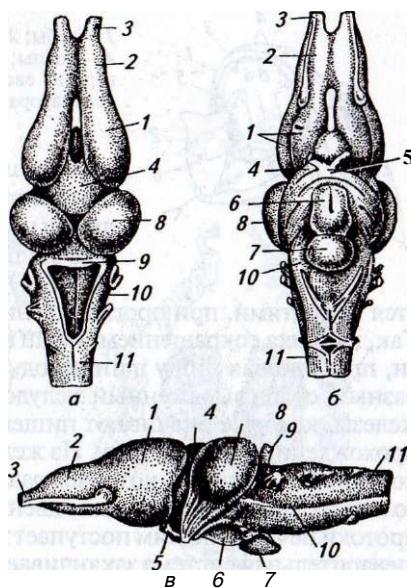
Череп в основном хрящевой с небольшим числом накладных (вторичных) и основных (пер-

Рис. 227. Мозг лягушки:
 а — сверху; б — снизу; в — сбоку; 1 — полушария переднего мозга; 2 — обонятельные доли; 3 — обонятельный нерв; 4 — промежуточный мозг; 5 — зрительные нервы; 6 — воронка промежуточного мозга; 7 — гипофиз; 8 — средний мозг; 9 — мозжечок; 10 — продолговатый мозг; 11 — спинной мозг

сивными чертами следует считать и наличие нервных клеток не только в дне и боковых стенках, но и в крыше полушарий. Промежуточный мозг сверху имеет придаток — эпифиз, а от дна промежуточного мозга отходит воронка, с которой связан гипофиз. Средний мозг и мозжечок развиты слабо. От центральной нервной системы отходят нервы ко всем органам тела. Черепных нервов 10 пар. Спинномозговые нервы образуют плечевое и пояснично-кресцовое сплетения, иннервирующие передние и задние конечности.

Органы чувств у амфибий получили в процессе эволюции прогressive развитие. Зрение в воздушной среде играет более важную роль, поэтому у земноводных произошли изменения в строении глаз. Образовались подвижные веки и мигательная перепонка, защищающие и увлажняющие глаза. Роговица глаза стала выпуклой, хрусталик приобрел линзовидную форму. Аккомодация происходит за счет перемещения хрусталика. Зрение устроено так, что земноводные реагируют на движение. В связи с тем что воздушная среда хуже проводит звуковые волны, органы слуха земноводных представлены как внутренним, так и средним ухом (барабанной полостью) со слуховой косточкой — стременем. Среднее ухо снаружи ограничено от внешней среды барабанной перепонкой и сообщается с глоткой каналом (евстахиевой трубой), что позволяет уравновешивать давление воздуха в нем с давлением воздуха во внешней среде. Полость среднего уха гомологична брызгалцу рыб. Органы обоняния — ноздри — сквозные (внутренние ноздри — хоаны) и служат также для дыхания. У личинок и постоянно живущих в воде земноводных сохранились характерные для рыб органы боковой линии.

Органы пищеварения. Широкий рот ведет в обширную ротовую полость (рис. 228): у многих земноводных на челюстях, а обычно также на нёбе, расположены мелкие зубы, помогающие удерживать добычу. В ротовую полость открываются внутренние ноздри — хоаны, в глотку — евстахиевые трубы. На дне ротовой полости имеется



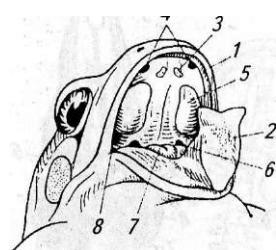


Рис. 228. Ротовая полость лягушки:

1 — зубы; 2 — язык; 3 — сошник с сошниковыми зубами; 4 — хоаны; 5 — просвечивающее глазное яблоко; 6 — отверстие евстахиевой трубы; 7 — гортань; 8 — отверстие резонатора

язык различной формы; у многих бесхвостых он прикреплен к передней части нижней челюсти и может выбрасываться изо рта; животные пользуются этим для ловли насекомых.

При отсутствии такого языка пища захватывается челюстями, при проглатывании добычи используются еще и глаза. Так, лягушка сокращением мышц втягивает глаза в глубь ротовой полости, проталкивая пищу в пищевод, которая поступает далее в мешкообразный, слабо выраженный желудок. У земноводных имеются слюнные железы, которые смачивают пищевой ком и способствуют его лучшему прохождению по пищеводу. Из желудка пища поступает в сравнительно короткий кишечник, который разделен на отделы (тонкий, средний и толстый). От желудка отходит двенадцатиперстная кишка, куда впадают протоки печени (по ним поступает желчь) и поджелудочной железы. Пищеварительная система заканчивается клоакой, куда открываются также мочеточники, канал мочевого пузыря и половые протоки (рис. 229).

Органы дыхания изменяются с возрастом животного. Личинки земноводных дышат наружными или внутренними жабрами. У взрослых амфибий развиваются легкие, хотя у некоторых хвостатых амфибий жабры сохраняются пожизненно. Легкие просто устроены и имеют вид тонкостенных эластичных мешков, иногда со складками на внутренней поверхности (см. рис. 222). Отношение поверхности легких к поверхности тела равно примерно 2:3. Поскольку земноводные не имеют грудной клетки, воздух в легкие поступает путем накачивания, при этом дно ротовой полости выступает как поршень насоса. При опускании дна ротовой полости воздух втягивается в нее через ноздри, затем ноздри закрываются, а дно ротовой полости, поднимаясь, проталкивает воздух в легкие. Выдох осуществляется за счет изменения объема внутренней полости земноводных и спадания легких. В дополнение к легочному большую роль играет кожное дыхание: именно через кожу выделяется до 80 % диоксида углерода и поступает до 50 % кислорода. Кожное дыхание является приспособлением для обитания в воде; во время спячки, которая у многих амфибий проходит в водоемах, эти животные также пользуются кожным дыханием. При нырянии легкие земноводных играют роль, сходную с ролью плавательного пузыря рыб.

Кровеносная система. Поскольку амфибии дышат легкими, у них имеется два круга кровообращения (рис. 230). Сердце у амфибий трехкамерное, оно состоит из двух предсердий и желудочка. Левое предсердие принимает кровь из легких, а правое — венозную кровь со всего тела, а также артериальную, идущую от кожи. Предсердия проталкивают кровь в желудочек через общее отверстие с клапанами. В желудочке

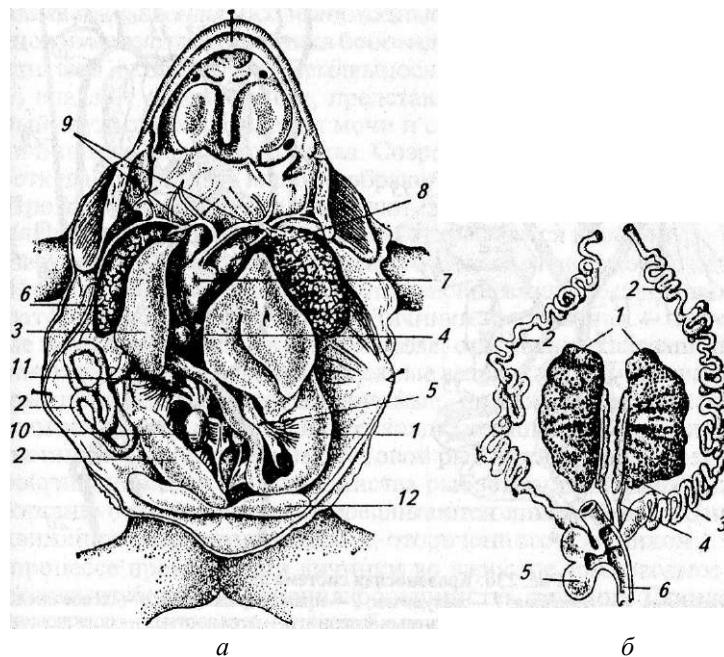


Рис. 229. Внутреннее строение лягушки:

a — вскрытый самец: 1 — желудок; 2 — кишка; 3 — печень; 4 — желчный пузырь; 5 — поджелудочная железа; 6 — легкие; 7 — желудочек сердца; 8 — предсердие; 9 — кровеносные сосуды, отходящие от сердца; 10 — семенник; 11 — селезенка; 12 — мочевой пузырь; *b* — органы размножения и выделения самки: 1 — яичники; 2 — яйцеводы; 3 — почка; 4 — мочеточники; 5 — мочевой пузырь; 6 — клоака

имеются карманы, которые не дают крови полностью смешиваться. От правой части желудочка отходит артериальный конус, за которым следует короткая брюшная аорта. У бесхвостых амфибий аорта делится на три пары симметрично отходящих сосудов, это видоизмененные (три из четырех) жаберные артерии рыб — предков амфибий. Первая пара — сонные артерии, несут кровь к голове. Вторая пара — дуги аорты, огибая сердце, сливаются в спинную аорту, от которой отходят артерии, несущие кровь к разным органам и участкам тела. Третья пара — легочные артерии, по ним кровь течет в легкие. По пути к легким от легочных артерий ответвляются большие кожные артерии, направляющиеся в кожу, где они разветвляются на множество капилляров, обусловливая кожное дыхание, имеющее у амфибий большое значение. В артериальном конусе имеется клапан, который смещается при сокращении желудочка и возрастании давления, открывая последовательно легочные артерии, дуги аорты и в последнюю очередь сонные артерии. Артериальный конус отходит от правой части желудочка, поэтому вначале **в** конус поступает кровь с наименьшим содержанием кислорода (в ле-

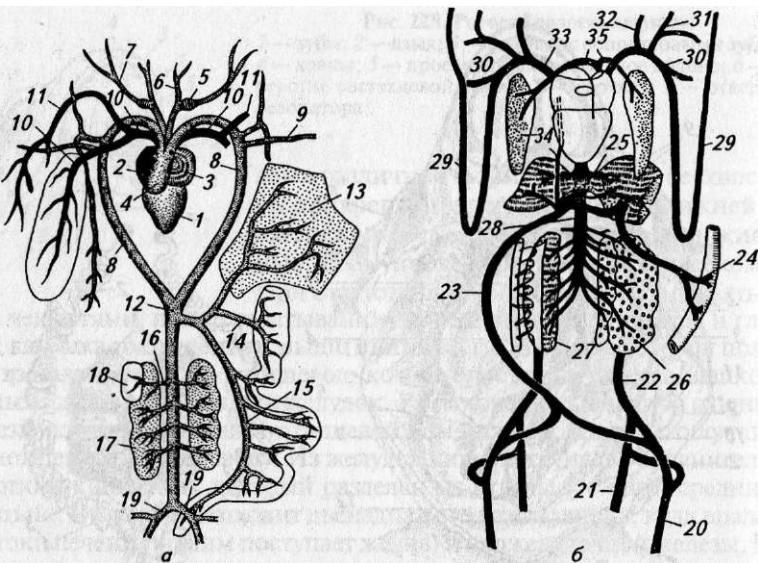


Рис. 230. Кровеносная система лягушки:
 а — артериальная; б — венозная; 7 — желудочек; 2 — правое предсердие; 3 — левое предсердие; 4 — артериальный конус; 5—7 — ветви сонных артерий; 8 — дуги аорты; 9 — подключичная артерия; 10 — легочная артерия; 11 — большая кожная артерия; 12 — спинная аорта; 13 — печень; 14 — желудочная артерия; 15 — кишечная артерия; 16 — почечная артерия; 17 — почки; 18 — семенник; 19 — подвздошные артерии; 20 — бедренная вена; 21 — седалищная вена; 22 — подвздошная вена; 23 — брюшная вена; 24 — воротная вена печени; 25 — печеночная вена; 26 — яичник; 27 — почка; 28 — задняя полая вена; 29 — большая кожная вена; 30 — подключичная вена; 31, 32 — яремные вены; 33 — передняя полая вена; 34 — легкое; 35 — легочные вены

точные артерии и далее к легким) и в последнюю очередь — с наибольшим (в сонные артерии и далее к голове).

Венозная кровь от задней части тела частично проходит в почки, где почечные вены распадаются на капилляры, образуя воротную систему почек. Вены, выходящие из почек, образуют непарную заднюю (нижнюю) полую вену. Другая часть крови от заднего отдела тела течет по двум сосудам, которые, сливаясь, образуют брюшную вену. Она, минуя почки, направляется в печень, где участвует в образовании воротной системы печени. По выходе из печени печеночные вены впадают в заднюю полую вену, а она — в венозный синус сердца. В венозный синус впадает и передняя полая вена, собирающая кровь от головы, передних и х конечностей и кожи. Из венозного синуса кровь изливается в правое предсердие. Органы выделения у взрослых амфибий представлены туловищными почками (см. рис. 229). От почек отходит пара мочеточников. Выводимая ими моча сначала попадает в клоаку, оттуда — в мочевой пузырь. При сокращении мочевого пузыря моча вновь оказывается в клоаке, а из нее выделяется наружу. У зародышей амфибий функционируют головные почки.

Органы размножения. Все земноводные раздельнополы (см. рис. 229). У самцов имеется два семенника бобовидной формы, расположенные в Полости тела около почек. Семявыносящие канальцы, пройдя через почку, впадают в мочеточник, представленный вольфовым каналом, который служит для выведения мочи и семени. У самок большие парные яичники лежат в полости тела. Созревшие яйца выходят в полость тела, откуда попадают в воронкообразные начальные отделы яйцеводов. Проходя по яйцеводам, икринки покрываются прозрачной толстой слизистой оболочкой. Яйцеводы открываются в клоаку.

Развитие у земноводных проходит со сложным метаморфозом (рис. 231). Из икринок выходят личинки, отличающиеся как по строению, так и по образу жизни от взрослых особей. Личинки земноводных — настоящие юодные животные. Обитая в водной среде, они дышат жабрами. Жабры у личинок хвостатых амфибий наружные ветвистые; у личинок бесхвостых амфибий жабры сначала наружные, но вскоре становятся внутренними вследствие обраствания их складками кожи. Кровеносная система личинок амфибий сходна с таковой рыб и имеет только один круг кровообращения. Как и у большинства рыб, у личинок амфибий имеются органы боковой линии. Передвигаются личинки в основном за счет движения уплощенного хвоста, отороченного плавником.

В процессе превращения личинки во взрослое земноводное у нее происходят глубокие изменения большинства органов. Появляются парные пятипалые конечности, у бесхвостых амфибий редуцируется

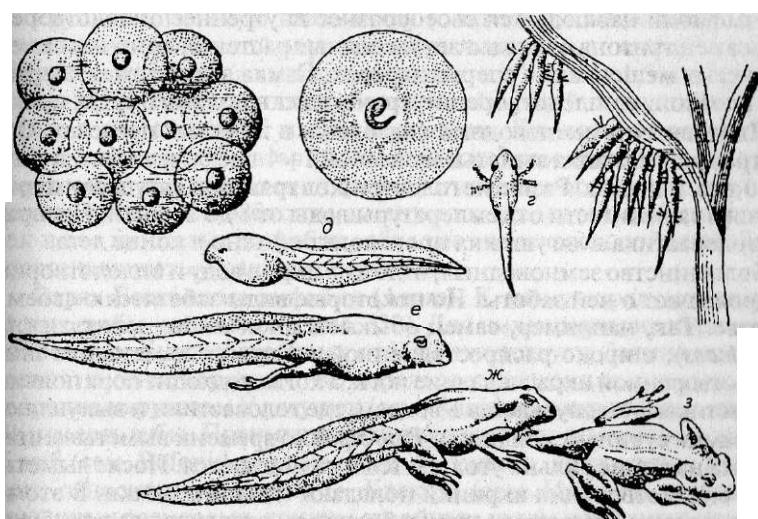


Рис. 231. Развитие лягушки:
а — икра; б — зародыш внутри оболочки; в, г — головастики с наружными жабрами;
д — головастик с внутренними жабрами; е, ж — головастики с конечностями; з — лягушонок с остатками хвоста

хвост. Жаберное дыхание заменяется легочным; жабры обычно исчезают. Вместо одного круга кровообращения развиваются два — большой и малый. При этом первая пара жаберных артерий превращается в сонные артерии, вторая пара становится дугами аорты, третья в той или иной степени редуцируется, а четвертая преобразуется в легочные артерии. У мексиканской амфибии амбистомы наблюдается неотения — способность размножаться на стадии личинки, т. е. достигать половой зрелости при сохранении личиночных черт строения. Личинок амбистом называют аксолотлями.

Экология и хозяйственное значение земноводных. Места обитания земноводных разнообразны, но большинство видов придерживается влажных мест, а некоторые проводят в воде всю жизнь, не выходя на сушу. Тропические земноводные червяги ведут подземный образ жизни. С своеобразной амфибией балканский протей обитает в подземных водоемах; глаза у него редуцированы, а кожа лишена пигмента. Земноводные относятся к группе холоднокровных (пойкилотермных) животных, т. е. температура их тела непостоянна и зависит от температуры окружающей среды. Уже при 10 °С их движения становятся вялыми, а при 5—7 °С они обычно впадают в оцепенение. Зимой в условиях умеренного и холодного климата жизнедеятельность амфибий почти замирает. Лягушки зимуют обычно на дне водоемов, а тритоны — на суще в норах, во мху, под камнями.

Размножаются земноводные в большинстве случаев весной. Самки лягушек, жаб и многих других бесхвостых земноводных выметывают икру в воду, где ее оплодотворяют самцы, поливая семенем. У хвостатых амфибий наблюдается своеобразное внутреннее оплодотворение. Так, самец тритона откладывает на водные растения комочки семени в слизистых мешочках — сперматофорах. Самка захватывает сперматофор клоакой, и оплодотворение происходит в половых путях самки.

Плодовитость земноводных колеблется в широких пределах. Обычная травяная лягушка выметывает весной 1—4 тыс., а зеленая лягушка — 5—10 тыс. икринок. Развитие головастиков травяной лягушки в икринке длится в зависимости от температуры воды от 8 до 28 дней. Превращение головастика в лягушонка происходит обычно в конце лета.

Большинство земноводных, отложив икру в воду и оплодотворив ее, не проявляют о ней заботы. Но некоторые виды заботятся о своем потомстве. Так, например, самец обыкновенной жабы-повитухи (*Alytes obstetricans*), широко распространенной в Европе, наматывает шнуры оплодотворенной икры на задние ноги, а когда подходит пора появиться головастикам, он спускается в водоем, где головастики и выплываются

У самки суринамской пипы (*Pipa papa*) ко времени выметывания икры кожа на спине сильно утолщается и размягчается. После выметывания и оплодотворения икринки попадают на спину самки. В этом помогает и самец, укладывая икру и брюшком вдавливая ее в разбухшую кожу, где и происходит развитие молодняка.

Питаются земноводные мелкими беспозвоночными животными, и первую очередь насекомыми. Они поедают много вредителей культур

ных растений, принося тем самым большую пользу растениеводству. Подсчитано, что одна травяная лягушка (*Rana temporaria*) за лето может съесть около 1200 насекомых — вредителей сельскохозяйственных растений. Еще более полезны жабы, поскольку они охотятся ночью и поедают массу ночных насекомых и слизней, малодоступных для птиц. В Западной Европе жаб часто выпускают в оранжереи и парники для истребления вредителей. Тритоны полезны тем, что поедают личинок комаров.

СИСТЕМАТИЧЕСКИЙ ОБЗОР ЗЕМНОВОДНЫХ. Класс *Amphibia* делится на три отряда: Хвостатые амфибии (*Caudata*, или *Urodea*), Бесхвостые амфибии (*Anura*), Безногие амфибии (*Apooda*).

Отряд Хвостатые амфибии (*Caudata*). Наиболее древняя группа земноводных, представленная в современной фауне саламандрами, углозубами, протеями и др. (рис. 232). Тело у хвостатых амфибий удлиненное, вальковатое. Хвост сохраняется всю жизнь. Передние и задние конечности примерно одинаковой длины, поэтому хвостатые амфибии либо ползают, либо медленно ходят. От водных предков у хвостатых амфибий сохранились кардиальные вены. Некоторые формы сохраняют жабры всю жизнь. Оплодотворение внутреннее.

В нашей стране из хвостатых амфибий широко распространены тритоны (*Triturus*). Наиболее часто встречаются крупный гребенчатый тритон (у них самцы черные с оранжевым брюхом) и более мелкий обыкновенный тритон (*T. vulgaris*).

Весной тритоны живут в воде, где и размножаются, а зиму проводят на суше в состоянии оцепенения. В Карпатах можно встретить довольно крупную огненную саламандру (*Salamandra salamandra*), которую легко узнать по черной окраске с оранжевыми или желтыми пятнами. Гигантская японская саламандра достигает в длину 1,5 м. К семейству протеев (*Proteidae*) относится балканский протей, живущий в водоемах пещер и сохраняющий жабры всю жизнь. Его кожа не содержит пигмента, а глазаrudиментарны, так как животное живет в темноте. В лабораториях для проведения физиологических опытов разводят личинок американских амбистом, именуемых аксолотлями. Эти животные, как и все хвостатые амфибии, обладают замечательной способностью восстанавливать утраченные части тела.

Отряд Бесхвостые амфибии (*Anura*). К этому отряду относятся лягушки, жабы, квакши (см. рис. 232). Для них характерно короткое, широкое тело. Хвост у взрослых особей отсутствует. Задние ноги значительно длиннее передних, что определяет движение скачками. Оплодотворение наружное. У лягушек (сем. *Ranidae*) кожа гладкая, слизистая. Во рту есть зубы. Преимущественно дневные и сумеречные животные. У жаб (сем. *Bufoidae*) кожа сухая, бугристая, во рту зубов нет, задние ноги относительно короткие. Квакши (сем. *Hylidae*) отличаются небольшими размерами, тонким стройным телом и лапами с присосками на концах пальцев. Присоски облегчают передвижение по деревьям, где квакши охотятся за насекомыми. Окраска квакш обычно ярко-зеленая, может меняться в зависимости от окружающего фона.

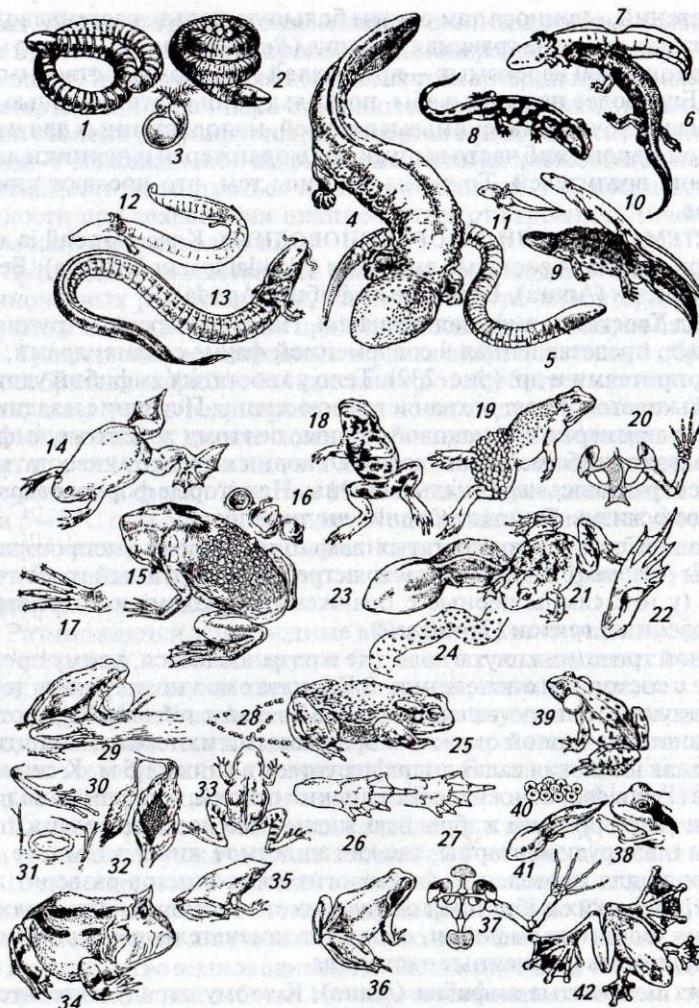


Рис. 232. Амфибии:

1 — кольчатая червяга; 2 — рыбозмей; 3 — личинка рыбозмея; 4 — гигантская саламандра; 5 — амфиума; 6 — амбистома; 7 — аксолотль (личинка амбистомы); 8 — огненная саламандра; 9 — гребенчатый тритон (самец); 10 — гребенчатый тритон (самка); 11 — личинка тритона; 12 — протей; 13 — сирен; 14 — шпорцевая лягушка; 15 — пипа; 16 — спинные ячейки пипы; 17 — передние лапы пипы; 18 — жерлянка; 19 — повитуха (самец с намотанной на ногах икрой); 20 — плечевой пояс жерлянки; 21 — чесночница; 22 — задние лапы чесночницы; 23 — головастик чесночницы; 24 — икра чесночницы; 25 — серая жаба; 26 — задняя лапа серой жабы; 27 — задняя лапа зеленой жабы; 28 — икра жабы; 29 — квакша; 30 — задняя нога квакши; 31 — гнездо квакши кузнеца; 32 — гнездо квакши филомедузы; 33 — сумчатая квакша; 34 — рогатка; 35 — ринодерма; 36 — бурая лягушка; 37 — плечевой пояс лягушки; 38 — остромордая лягушка; 39 — зеленая лягушка; 40 — икра зеленой лягушки; 41 — головастик лягушки; 42 — летающая лягушка

Отряд Безногие амфибии (Apoda). К этому отряду принадлежат тропические земноводные, ведущие подземный образ жизни (червяги, рыбозмеи). Имеют длинное, цилиндрическое тело с коротким хвостом (см. рис. 232). В связи с роющим образом жизни ноги и глаза у них подверглись редукции. Оплодотворение внутреннее. Питаются почвенными беспозвоночными животными.

ПРОИСХОЖДЕНИЕ ЗЕМНОВОДНЫХ

Изучение земноводных представляет интерес для выяснения эволюции позвоночных животных, перехода их из водной среды на сушу. При переходе, хотя и частичном, к наземному существованию у древних земноводных сохранился ряд признаков рыбообразных предков, но вместе с тем появились новые приспособления: из парных плавников образовались пятипалые конечности, жаберное дыхание у взрослых заменилось легочным, произошли и другие глубочайшие изменения.

Амфибии произошли от одной из ветвей кистеперых рыб. Древнейшие амфибии появились уже в девонский период. Они были тогда в основном водными животными, по-видимому, лишь ненадолго покидавшими водоемы. Об этом свидетельствует, например, то, что у них были органы боковой линии. Самые ранние амфибии — ихтиостегиды (*Ichthyostegidae*). В карбоне возникает несколько ветвей амфибий, в том числе и предки современных. Хвостатые и безногие, по-видимому, берут начало от лепосpondиляй (*Lepospondyli*), а бесхвостые — от лабиринтодонтов (*Labyrinthodontia*). Эту группу амфибий иногда называют стегоцефалами. От современных амфибий они отличались, в частности, щитом из накладных костей на черепе. Расцвет амфибий приходился на каменноугольный период, позже большинство видов вымерло.

КЛАСС ПРЕСМЫКАЮЩИЕСЯ, ИЛИ РЕПТИЛИИ (*Reptilia*)

Все основные черты высших наземных позвоночных наглядно выражены у пресмыкающихся.

Пресмыкающиеся относятся к высшим позвоночным, или настоящим наземным позвоночным животным, часть из которых вторично перешла к водному образу жизни. Рептилии представляют наиболее пизкоорганизованных высших позвоночных из группы *Amniota*. Способность к терморегуляции у них невелика, и температура тела непостоянна, это все еще пойкилтермные животные. Во время бодрствования температура может колебаться в значительных пределах, например у некоторых ящериц от 14 до 32 °C.

Ряд черт характеризует пресмыкающихся как типичных наземных животных. В связи с сухопутным образом жизни тело их расчленено на отделы в большей мере, чем у амфибий и рыб. Шея хорошо выражена. Конечности пятипалые, но у некоторых они частично или полностью атрофировались. Кожа сухая, с сильным ороговением эпидермиса.

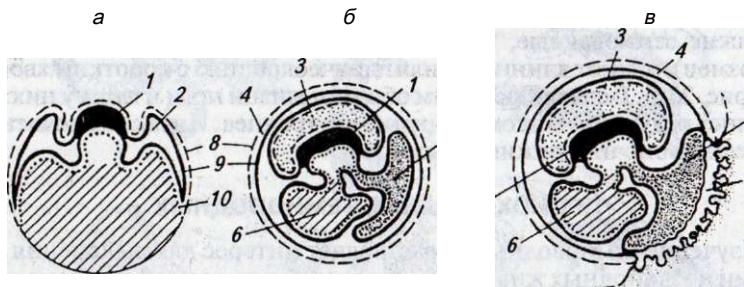


Рис. 233. Образование зародышевых оболочек у амниот:
 а — начало формирования зародышевых оболочек; б — зародышевые оболочки у рептилий и птиц; в — то же у млекопитающих; 1 — зародыш; 2 — внутризародышевая полость; 3 — амнион; 4 — сероза; 5 — аллантоис; 6 — желочный мешок; 7 — плацента; 8 — эктодерма; 9 — мезодерма; 10 — энтодерма

Поверхность тела покрыта обычно роговыми чешуями или щитками. Череп сочленяется с позвоночником одним мышцелком. Позвонки туловища несут ребра, которые составляют грудную клетку. Полушария переднего мозга хорошо развиты, в их крыше имеется серое мозговое вещество, образующее зачатки коры мозга. В течение всей жизни дышат легкими; вдох и выдох производятся путем расширения и сжатия грудной клетки. Сердце у большинства видов трехкамерное, но с неполной перегородкой в желудочке. Артериального конуса нет, три артерии отходят непосредственно от желудочка. Органами выделения у взрослых служат тазовые почки. Оплодотворение внутреннее. У эмбрионов развиваются оболочки амнион и аллантоис (рис. 233).

По сравнению с земноводными пресмыкающиеся более многочисленный и разнообразный класс. Число видов современных пресмыкающихся превышает 6500. Ныне живущих объединяют в четыре отряда, которые составляют три подкласса.

- Класс Пресмыкающиеся (Reptilia)
 - Подкласс Анапсиды (Anapsida)
 - Отряд Черепахи (Testudines)
 - Подкласс Лепидозавры (Lepidosauria)
 - Отряд Клювоголовые (Rhynchocephalia)
 - Отряд Чешуйчатые (Squamata)
 - Подотряд Ящерицы (Lacertilia)
 - Подотряд Змеи (Serpentes)
 - Подкласс Архозавры (Archosauria)
 - Отряд Крокодилы (Crocodylia)

В географическом отношении рептилии распространены гораздо шире, чем амфибии. Максимальное число видов обитает в тропиках и субтропиках, но рептилии не избегают пустынь и даже там весьма многочисленны.

Строение и жизненные отправления. *Внешний вид.* Рептилии живут в разнообразных условиях, и поэтому их внешний вид и форма тела так-

же довольно разнообразны. Наиболее характерны следующие три типа внешнего строения (рис. 234):

1) ящерицеобразный тип свойствен большинству рептилий. Выражены все отделы тела. Конечности хорошо развиты. Хвост, как правило, длинный и у некоторых видов цепкий. Сюда относятся ящерицы, хамелеоны, крокодилы и клювоголовые;

2) змееобразный тип характеризуется цилиндрическим телом, отсутствием конечностей и обособленной шеи. Туловищный и хвостовой отделы переходят друг в друга постепенно. К этому типу относятся змеи и безногие ящерицы. Между первым и вторым типами есть переходные формы: амфисбены имеютrudименты только передних конечностей, а удавы имеют зачаточные задние конечности;

3) черепахообразный тип характеризуется более или менее уплощенным телом, заключенным между спинным (карапакс) и брюшным (Пластрон) костными щитами. Конечности часто укороченные, у сухопутных форм столбчатообразные, у морских — ластообразные. Шея длинная и подвижная. К этому типу относятся черепахи.

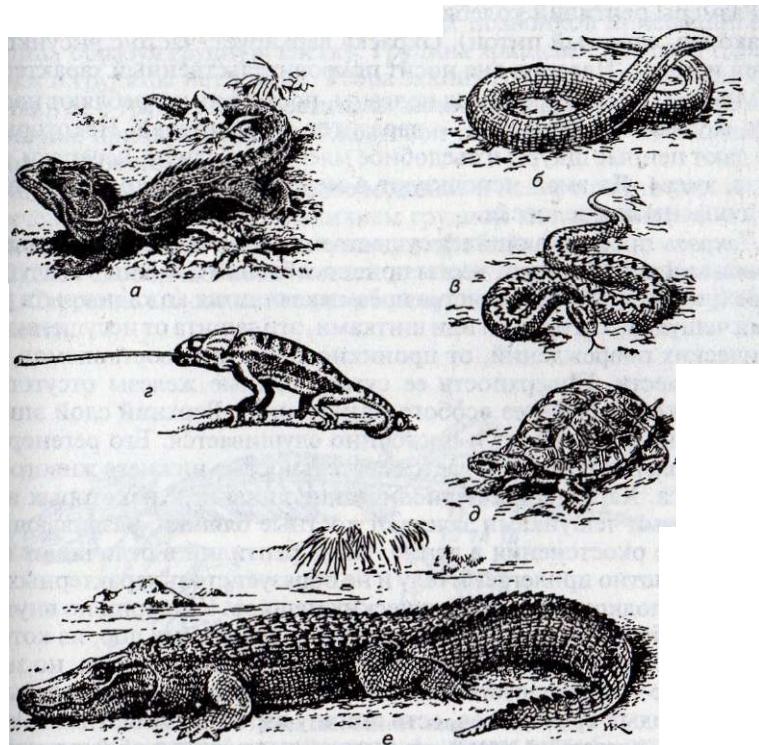


Рис. 234. Представители разных отрядов пресмыкающихся:
а — гаттерия; б — желтопузик (безногая ящерица); в — гадюка; г — хамелеон; д — степная
черепаха; е — нильский крокодил

Рис. 235. Продольный разрез кожи ящерицы:
1 — эпидермис; 2 — дерма; 3 — роговой слой эпидермиса; 4 — пигментные клетки в мальпигиевом слое эпидермиса; 5 — костные образования в дерме

У рептилий нередок лазательный тип конечностей, который хорошо выражен у хамелеонов. У гекконов, лазающих по гладкой поверхности камней, пальцы несут на концах своеобразные присоски. У летающих драконов на боках тела между передними и задними конечностями расположена кожистая складка, в формировании которой участвуют и ребра; эта складка используется для планирования при прыжках.

Размеры рептилий колеблются от нескольких сантиметров до 11 м (анаконда, сетчатый питон). Окраска варьирует, часто с рисунком из пятен и полос. Нередко она носит покровительственный характер.

Многие пресмыкающиеся полезны, поскольку истребляют насекомых, моллюсков, грызунов — паразитов и вредителей. Некоторые из них дают ценные шкуры и съедобное мясо (крокодилы, черепахи, ящерицы, змеи). Яд змей используют в медицине. Некоторые ядовитые змеи опасны для человека.

Покровы пресмыкающихся существенно отличаются от кожных покровов амфибий и имеют черты приспособления к жизни в воздушной среде (рис. 235). У большинства пресмыкающихся кожа покрыта роговыми чешуями, бугорками или щитками, это защита от иссушения, механических повреждений, от проникновения микроорганизмов, ядовитых веществ. Поверхность ее сухая, кожные железы отсутствуют (кроме некоторых желез особого назначения). Верхний слой эпидермиса сильно ороговевает и постоянно слущивается. Его регенерация (восстановление) обеспечивается деятельностью нижнего живого слоя эпидермиса. Характерны периодические линьки. У некоторых видов под роговыми чешуйками залегают костные бляшки, развивающиеся как кожные окостенения в дерме. Кожа рептилий в отличие от кожи амфибий плотно прилегает к телу и не образует столь характерных (как у лягушек) подкожных лимфатических мешков. У ящериц по внутреннему краю бедер имеется ряд отверстий — бедренных пор, из которых в период размножения выделяется вязкая нитевидная масса, но значение этих пор пока неизвестно. Немногочисленные кожные железы развиты у молодых крокодилов, есть они и у взрослых животных. Расположены эти железы на спине, на нижней челюсти и в области клоаки. Относительно хорошо кожные железы развиты у некоторых черепах. Зачатки кожных желез имеются у змей.

Скелет пресмыкающихся почти полностью образован костными элементами (рис. 236). Череп состоит как из окостеневших хрящей черепа эмбриона, так и из большого числа кожных костей, формирующих крышу, бока и дно черепа, длинные челюсти.

Позвоночный столб включает пять отделов — шейный, грудной, поясничный, крестцовый и хвостовой. Позвонки процельные, у низших форм тела позвонков амфицельные. Шея длинная, в шейном отделе восемь позвонков. Первый шейный позвонок (атлас, или атлант) представляет собой костное кольцо, разделенное связкой на нижнюю и верхнюю половины. Верхнее отверстие служит для соединения головного мозга со спинным, в нижнее отверстие заходит зубовидный отросток второго шейного позвонка (эпистрофея), который служит осью вращения головы. Таким образом, атлант, вращаясь вокруг зубовидного отростка эпистрофея, обеспечивает значительную подвижность головы; значительная подвижность обеспечивается также за счет соединения черепа с помощью одного мышелка. Наличие двух первых шейных позвонков — атланта и эпистрофея — характерно для всех тетрапод.

Грудные позвонки пресмыкающихся несут по паре хорошо развитых ребер, но только ребра пяти первых позвонков присоединены к грудине, образуя грудную клетку. Грудина хрящевая. Полной грудной клетки и грудины нет у змей. Ребра задних грудных позвонков не соединяются с грудиной. Крестцовых позвонков два, к их поперечным отросткам причленяется таз. Хвостовой отдел состоит из нескольких десятков позвонков.

Конечности и их пояса более мощные и более прочно укреплены, а плечевой пояс в связи с наличием грудной клетки связан с осевым скелетом, а не лежит свободно, как у амфибий.

Скелет свободных конечностей не имеет существенных особенностей. Строение его соответствует общему плану строения пятипалой конечности.

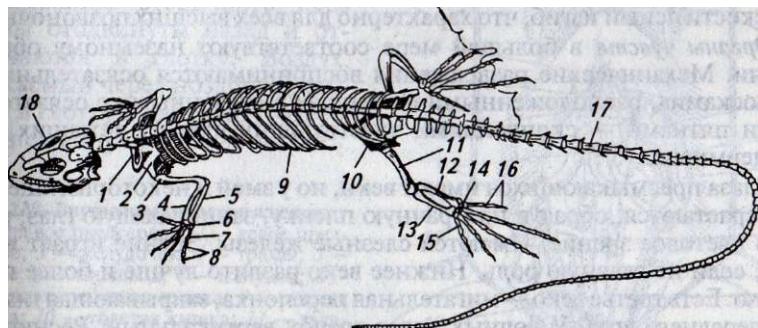


Рис. 236. Скелет ящерицы:
1 — ключица; 2 — лопатка; 3 — плечевая кость; 4 — лучевая кость; 5 — локтевая кость; 6 — запястье; 7 — пясть; 8 — фаланги пальцев; 9 — ребра; 10 — таз; 11 — бедренная кость; 12 — большая берцовая кость; 13 — малая берцовая кость; 14 — предплюсна; 15 — плюсна; 16 — фаланги пальцев; 17 — позвоночник; 18 — череп

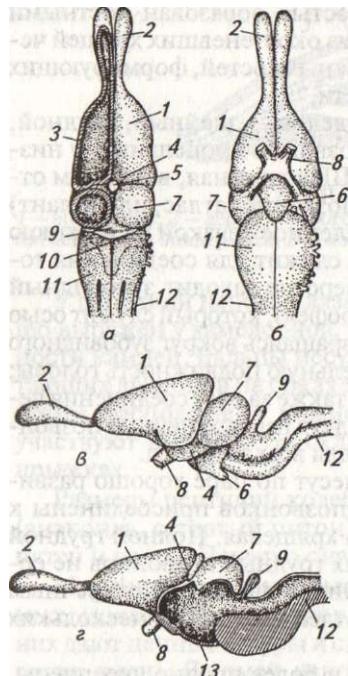


Рис. 237. Головной мозг ящерицы:
 а — сверху; б — снизу; в — сбоку; 1 — полушария переднего мозга; 2 — обонятельные доли; 3 — полосатое тело; 4 — промежуточный мозг; 5 — теменной орган; 6 — гипофиз; 7 — средний мозг; 8 — зрительные нервы; 9 — мозжечок; 10 — четвертый желудочек; 11 — продолговатый мозг; 12 — спинной мозг; 13 — третий желудочек

Мускулатура. У рептилий почти не сохраняется метамерное расположение мускулатуры, которое свойственно низшим позвоночным. Достаточно хорошее развитие пятипалых конечностей, появление шейного отдела, большая расчлененность тела — все это привело к сложной дифференцировке мышечной системы. Появилась межреберная мускулатура, играющая важную роль в механизме дыхания у всех высших позвоночных. Хорошо также развита шейная и жевательная мускулатура.

Нервная система более совершенна, чем у амфибий. Головной мозг значительно больше, полушария переднего мозга относительно крупнее, они

имеют кору из серого мозгового вещества (рис. 237). Но кора развита еще слабо. Хорошо выражен архипаллиум и есть зачатки неопаллиума. Хорошо развит теменной орган, эпифиз и обонятельный центр. Мозжечок довольно большой, что соответствует сложности движений большинства рептилий. Продолговатый мозг образует в вертикальной плоскости ясный изгиб, что характерно для всех высших позвоночных.

Органы чувств в большей мере соответствуют наземному образу жизни. Механические раздражения воспринимаются осязательными «волосками», расположенными на чешуях и связанными с осязательными пятнами — скоплениями чувствующих клеток, лежащих под эпидермисом.

Глаза пресмыкающихся имеют веки, но у змей и некоторых ящериц они срастаются, образуя прозрачную пленку, защищающую глаз; развито цветовое зрение. Имеются слезные железы. Зрение играет важную, если не главную роль. Нижнее веко развито лучше и более подвижно. Есть третье веко — мигательная перепонка, закрывающая глаз из его переднего угла. У ночных видов зрачок вертикальный. Ресничная мышца образована поперечно-полосатой мускулатурой и не только перемещает хрусталик, но и несколько меняет его форму. Это в условиях наземной среды имеет большое значение для различения предметов на разных расстояниях. У гремучих змей есть термолокатор — до 0,001 °С.

У многих рептилий на темени располагается своеобразный теменной орган (теменной глаз), связанный с промежуточным мозгом. Его строение сходно со строением глаза, и он может воспринимать световые раздражения.

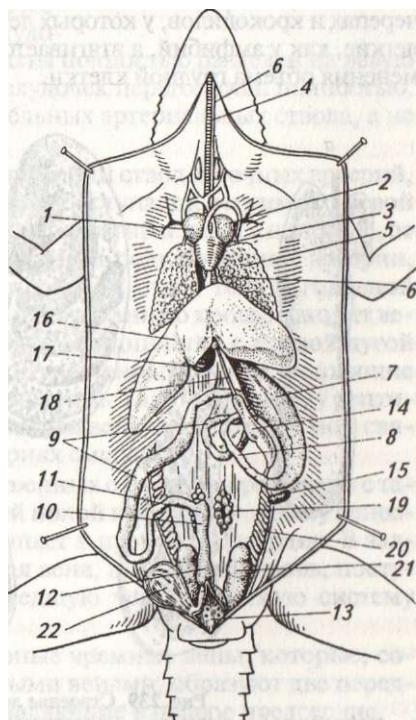
Орган слуха состоит из внутреннего и среднего уха, в котором находится одна слуховая косточка — стремя. В перепончатом лабиринте обособляется улитка, представляющая собой мешкообразный выступ. Имеются также органы осязания, обоняния и вкуса.

Орган обоняния отчетливо подразделяется на нижний — дыхательный, и верхний — собственно обонятельный отделы.

Для рептилий характерен якобсонов орган — извитая и слепо заканчивающаяся полость, отходящая от крыши рта. Многие рептилии как бы ощупывают языком предметы, перенося мельчайшие частицы в рот, где с помощью якобсонова органа воспринимают их запахи и запахи пищи.

Органы пищеварения устроены сложнее, чем у амфибий. Пищеварительный тракт пресмыкающихся (рис. 238) начинается ротовой полостью, в которой находятся язык и зубы. Зубы свойственны большинству рептилий. Они прирастают к краям соответствующих костей и только у крокодилов они сидят в альвеолах. На дне ротовой полости расположен подвижный мускулистый язык, способный далеко выбрасываться. Форма языка весьма различна. У змей и многих ящериц он тонкий и часто раздвоенный на конце. У хамелеонов, напротив, на конце язык расширен. У черепах и крокодилов носоглоточные ходы отделены от ротовой полости вторичным костным нёбом. Благодаря образованию вторичного нёба хоаны отодвинуты назад и открываются в глотку. Воздух, вдыхаемый через ноздри, поступает в глотку и далее по трахее в легкие, минуя ротовую полость.

Рис. 238. Внутреннее строение ящерицы:
1 — правое предсердие; 2 — левое предсердие; 3 — желудочек; 4 — трахея; 5 — легкое; 6 — пищевод; 7 — желудок; 8 — двенадцатиперстная кишка; 9 — тонкая кишка; 10 — толстая кишка; 11 — зачаточный слепой вырост кишки; 12 — прямая кишка; 13 — клоака; 14 — поджелудочная железа; 15 — селезенка; 16 — печень; 17 — желчный пузырь; 18 — желчный проток; 19 — яичник; 20 — яйцевод; 21 — почка; 22 — мочевой пузырь



Это позволяет животному дышать при заглатывании пищи. Ротовые слюнные железы относительно хорошо развиты, и слюна содержит пищеварительные ферменты.

Ротовая полость четко ограничена от глотки. Пищевод ведет в хорошо развитый желудок. Желудок четко выражен, снабжен сильной мускулатурой. Кишечник ясно подразделяется на более длинную тонкую и относительно короткую толстую кишку. Между тонкими и толстыми кишками расположена зачаточная слепая кишка. Она хорошо развита только у растительноядных черепах. Поджелудочная железа имеет свой проток. У рептилий имеется желчный пузырь, проток которого впадает в кишечник рядом с протоком поджелудочной железы. Имеется клоака. У пресмыкающихся значительно расширен спектр кормов. Они могут расчленять пищевой объект, способны долго голодать.

Органами дыхания пресмыкающихся в течение всей жизни служат легкие (у змей имеется только одно правое), кожное дыхание отсутствует (рис. 239). Газообмен у зародыша, развивающегося в яйце, осуществляется с помощью сосудов аллантоиса и желточного мешка. Характерна дифференцировка дыхательных путей. От глотки начинается трахея (дыхательное горло), которая делится на два бронха, ведущие в мешковидные легкие. В полости легкого располагаются множество складок и мелких ячеек, увеличивающих поверхность газообмена. Особенно хорошо это выражено у черепах и крокодилов, у которых легкие губчатые. Воздух не нагнетается в легкие, как у амфибий, а втягивается и выталкивается обратно за счет изменения объема грудной клетки.

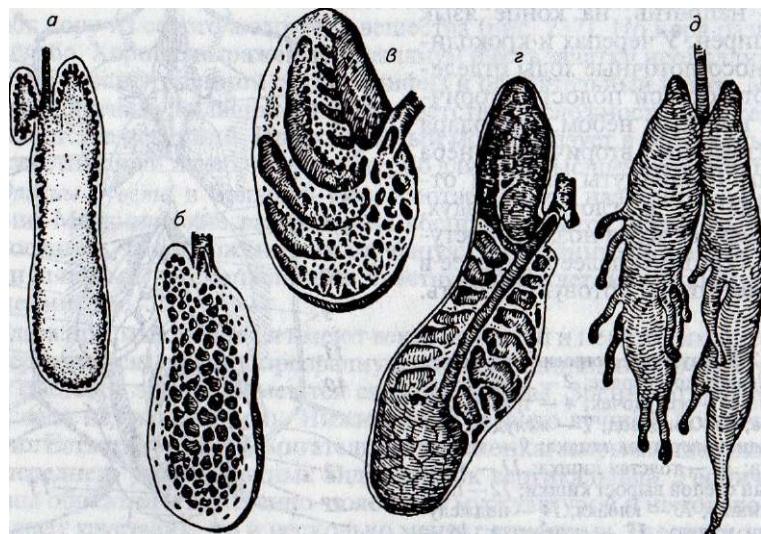


Рис. 239. Строение легких пресмыкающихся:
а — амфисбена; б — гаттерия; в — варан; г — аллигатор; д — хамелеон (внешний вид легких)

Рис. 240. Кровеносная система ящерицы:
 1—левая дуга аорты; 2—правая дуга аорты; 3—сонные артерии; 4—спинная аорта; 5—внутренностная артерия; 6—кишечная артерия; 7—межпозвоночная артерия; 8—артерии задних конечностей; 9—левая брюшная артерия; 10—мезентериальная артерия; 11—легочная артерия; 12—легкое; 13—яремные вены; 14—вены задних конечностей; 15—тазовые вены; 16—брюшина вена; 17—почки; 18—задняя полая вена; 20—кишка; 21—воротная вена печени; 22—печень

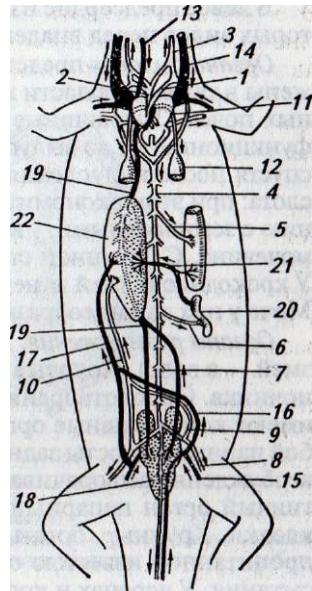
Кровеносная система пресмыкающихся по сравнению с кровеносной системой земноводных имеет ряд черт, лучше соответствующих наземному образу жизни (рис. 240).

Сердце трехкамерное, перегородка между предсердиями всегда полная, и каждое самостоятельно открывается в соответствующую часть желудочка. Кроме того, неполная перегородка имеется и в желудочке. За счет этой перегородки желудочек на короткое время во время диастолы полностью разделен на левую и правую половины. У крокодилов желудочек перегорожен полностью. Из желудочка выходят три самостоятельных артериальных стволов, а не артериальный конус, как у амфибий.

От правой части желудочка отходит общий ствол легочных артерий, по которым практически венозная кровь поступает к легким. От левой части желудочка отходит правая дуга аорты (загибающаяся направо), от которой, в свою очередь, отходят сонные и подключичные артерии, снабжающие артериальной кровью передний отдел тела и головной мозг. От середины желудочка (содержит смешанную кровь) отходит левая дуга аорты, которая, обогнув сердце, соединяется с правой дугой аорты и образует спинную аорту. Многочисленные сосуды, отходящие от нее, несут кровь к различным органам тела. Таким образом, у рептилий более полно разделены артериальный и венозный потоки, но в связи с двумя дугами аорты кровь в артериях смешанная.

Венозная система не имеет существенных отличий в сравнении с таковой бесхвостых амфибий. По задней полой вене — основному венозному сосуду туловища — кровь поступает в правое предсердие. В заднюю полую вену впадает и печеночная вена, выносящая кровь, поступившую туда от кишечника и прошедшую через воротную систему сосудов печени.

От головы кровь собирается в парные яремные вены, которые, соединившись с парными подключичными венами, образуют две передние (левую и правую) полые вены, впадающие в правое предсердие.



В левое предсердие изливают кровь легочные вены, которые у некоторых видов перед впадением в сердце объединяются в один сосуд.

Органы дыхания представлены тазовыми почками, которые расположены в тазовой области и по микроструктуре отличаются от туловищных почек. Туловищные почки возникают как зародышевый орган и функционируют до вылупления животных из яйца или некоторое время спустя после вылупления. Конечным продуктом является мочевая кислота; при этом экономится примерно в 200 раз больше воды по сравнению с земноводными, у которых конечным продуктом обмена является мочевина. С брюшной стороны в клоаку открывается мочевой пузырь. У крокодилов, змей и некоторых ящериц мочевой пузырь недоразвит. Моча у них кашицеобразная и состоит в основном из мочевой кислоты.

Органы размножения. Размножение происходит на суше (у морских змей — в воде). Половые железы лежат в полости тела по бокам позвоночника. Оплодотворение внутреннее. Все рептилии, кроме гаттерии, имеют копулятивные органы. У ящериц и змей они представляют собой парные выросты задней стенки клоаки, которые в период полового возбуждения выворачиваются наружу. У крокодилов и черепах копулятивный орган непарный и также представляет собой вырост стенки клоаки. Крупные, богатые желтком яйца покрыты пергаментной или пропитанной известью оболочкой, защищающей содержимое от высыхания. У черепах и крокодилов появляется белковая оболочка. Развитие пресмыкающихся прямое без метаморфоза. В связи с наземным образом жизни у зародышей пресмыкающихся появляются две зародышевые оболочки — амнион и аллантоис. Амнион — это мешок, заполненный амниотической жидкостью, в которой плавает зародыш (см. рис. 194). Аллантоис выполняет роль органа дыхания и зародышего мочевого пузыря. Отмечено партеногенетическое размножение (некоторые агамы, гекконы), у них популяции состоят только из самок. Есть случай гермафродитизма — остроногая змея ботропс.

Экология пресмыкающихся. Пресмыкающиеся широко распространены по земному шару, но большинство обитает в странах с жарким и теплым климатом. Это обусловливается зависимостью их жизнедеятельности, как животных с непостоянной температурой тела, от температуры окружающей среды.

Пресмыкающиеся, как правило, наземные животные. Однако ряд видов ведет водный образ жизни. Так, в морях, реках и озерах обитают крокодилы, некоторые змеи и черепахи. Все водные пресмыкающиеся дышат воздухом и размножаются на суше (исключение составляют живородящие морские змеи). В умеренных и северных широтах пресмыкающиеся проводят зиму в глубоком оцепенении, укрывшись в различные убежища.

Большинство пресмыкающихся размножаются путем откладывания яиц. Их яйца по строению сходны с яйцами птиц. Самки обычно откладывают яйца в углубления на почве или в трещины скал, под кору деревьев, в навоз. Срок инкубации зависит от температуры окружающей среды. Некоторые пресмыкающиеся, например крокодилы, охра-

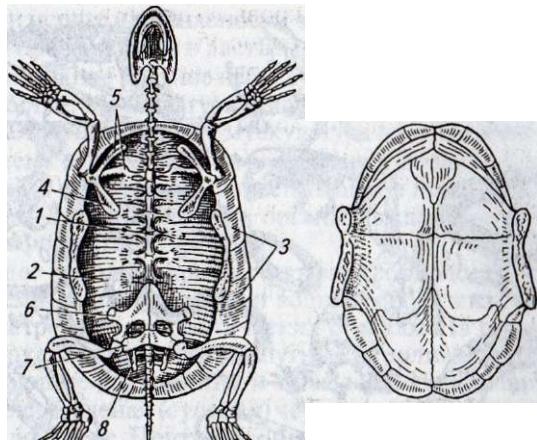


Рис. 241. Скелет болотной черепахи:
1 — карапакс; 2 — туловищный отдел позвоночного столба; 3 — реберные пластинки;
4 — краевые пластинки; 5 — коракоид; 6 — лопатка; 7 — подвздошная кость;
8 — лобковая кость; 9 — седалищная кость; 10 — пластрон

няют кладку яиц. У других наблюдается яйцеживорождение. В этом случае оплодотворенные яйца остаются в теле матери, и в них развиваются молодые животные, которые выходят из яиц тотчас же после их откладки.

Мелкие пресмыкающиеся питаются преимущественно беспозвоночными, главным образом насекомыми. Крупные нападают на позвоночных животных. Ядовитые змеи убивают жертву, вонзая в нее ядовитые зубы, по каналам или бороздкам которых в ранку стекает яд, вырабатываемый специальными, видоизмененными слюнными железами. Имеются также и растительноядные пресмыкающиеся, поедающие траву и плоды (сухопутные черепахи), и всеядные.

В России и других странах существуют специальные питомники ядовитых змей, от которых получают яд, используемый для приготовления ряда лекарств и сывороток.

ПОДКЛАСС АНАПСИДЫ (ANAPSIDA). **Отряд Черепахи (Testudines)** — наиболее специализированная группа пресмыкающихся. Уплощенное широкое тело черепах покрыто прочным панцирем (рис. 241), образованным двумя щитами — более выпуклым спинным (карапакс) и уплощенным брюшным (пластрон). С карапаксом сливаются ребра и большая часть позвоночника, с пластроном — грудина и ключицы. Свободный шейный и хвостовой отделы позвоночника, лопатки, коракоиды и конечности. Сверху панцирь прикрыт роговыми щитками, а у кожистых черепах — мягкой кожей эпидерmalного происхождения. Спереди между щитками выступают голова и передние ноги, сзади — хвост и зад-

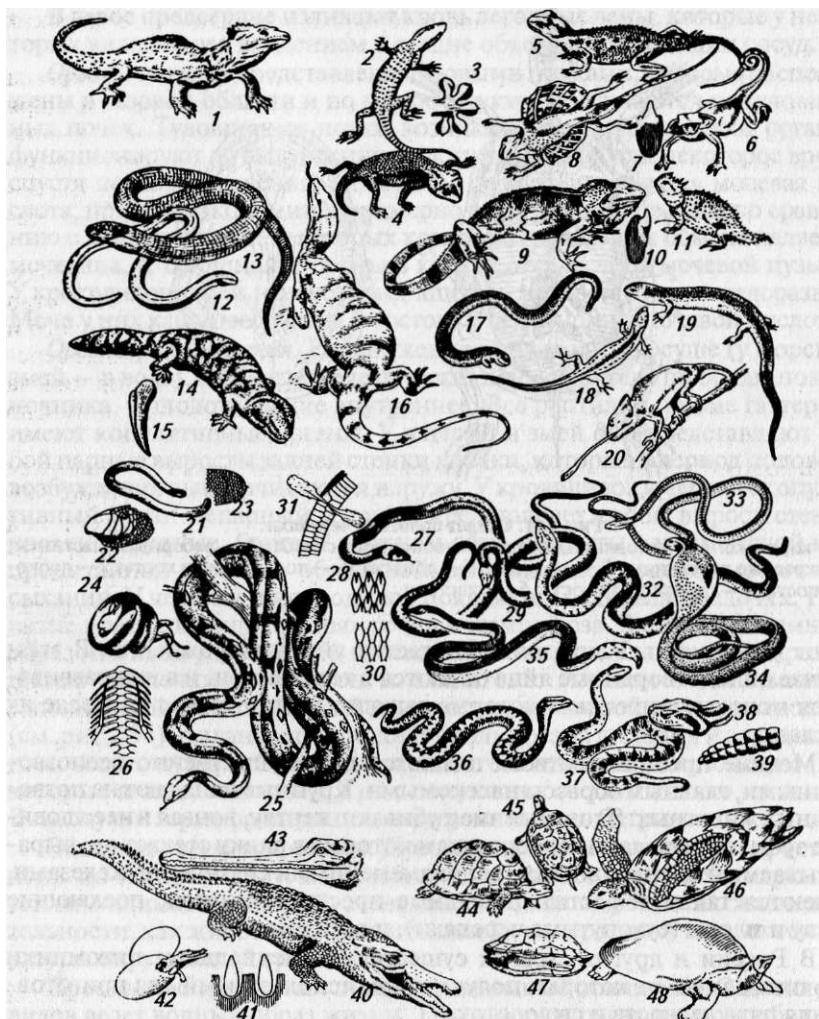


Рис. 242. Рептилии:
 1—гаттерия; 2—серый геккон; 3—лапа широкопалого геккона; 4—сцинковый геккон; 5—степная агама; 6—ушастая круглоголовка (в позе угрозы); 7—зуб агамы; 8—летучий дракон—игуана; 10—зуб игуаны; 11—фринозома рогатая; 12—веретеница; 13—желтопузик; 14—ядозуб; 15—зубядозуба; 16—серый варан; 17—хирот; 18—пряткая ящерица; 19—длинноногий сцинк; 20—хамелеон; 21—слепозмейка; 22—голова слепозмейки; 23—задний конец тела слепозмейки; 24—песчаный удавчик; 25—обыкновенный удав; 26—скелет пояса задних конечностей удава; 27—обыкновенный уж; 28—чешуя ужа; 29—мядянка; 30—чешуя медянки; 31—копулятивные органы змей; 32—узорчатый полоз; 33—стрела-змея; 34—cobra в позе угрозы; 35—морская змея пеламида; 36—обыкновенная гадюка; 37—щитомордник; 38—голова гремучей змеи; 39—конец хвоста гремучей змеи; 40—крокодил; 41—зубы крокодила; 42—яйца крокодила; 43—гавиал; 44—степная черепаха; 45—болотная черепаха; 46—каретта; 47—змеешейная черепаха; 48—трионикс

1 ше ноги. При испуге или опасности черепахи втягивают их. У морских черепах ноги превратились в ласты.

Развито вторичное костное нёбо, зубов нет, но есть роговые чехлы на челюстных костях. Мускулатура туловища в связи с наличием панциря развита слабо. Хорошо развита мускулатура шеи, конечностей и хвоста.

Легкие имеют сложное губчатое строение. Воздух попадает в них путем засасывания в результате движения шеи и ног и опускания дна ротовой полости. Дополнительные органы дыхания — снабженные капиллярами выросты глотки и клоаки.

В настоящее время в мире насчитывается около 250 видов черепах, ведущих наземный, полуводный и водный образ жизни, большинство из которых встречается в тропических странах. На островах Тихого и Индийского океанов водятся сухопутные исполинские слоновые черепахи, достигающие 2 м в длину и массы 200 кг. Объектом промысла служила морская зеленая (суповая) черепаха (*Chelonia mydas*) длиной до 2 м и массой до 400 кг. Почти все они истреблены.

В болотах и озерах южных районов нашей страны водится болотная черепаха (*Emys orbicularis*), питающаяся водными животными (рис. 242, 45). Ноги этих черепах имеют плавательные перепонки. Близка к ней по биологии каспийская черепаха (*Mauremis caspica*). На Дальнем Востоке обитает дальневосточная черепаха (рис. 242, 48) — трионикс (*Trionyx sinensis*). Для этого вида характерно наличие длинного подвижного хоботка, на конце которого открываются ноздри. Плавает каспийская черепаха быстро и может долго оставаться под водой (2—10 ч). Из сухопутных черепах встречается средиземноморская черепаха (*Testudo graeca*). В террариумах часто содержат среднеазиатскую черепаху (*Agrionemys horsfieldi*), обитающую, в частности, в Средней Азии и Казахстане. В природе они откладывают яйца в апреле—мае, инкубация продолжается 70—80 дней. Летом взрослые могут впадать в спячку, закапываясь в грунт. Летняя спячка может переходить в зимнюю, т. е. длиться до 8 мес. Мясо и яйца этих черепах съедобны.

ПОДКЛАСС ЛЕПИДОЗАВРЫ (LEPIDOSAURIA). Подкласс объединяет около 6 тыс. различных видов ящериц, змей и хамелеонов (см. рис. 242). Тело их покрыто роговыми чешуйками, бугорками или щитками. Костные образования в коже встречаются редко. Отверстие клоаки в виде попечерной щели. У многих представителей отряда наблюдается частичная редукция или даже полное исчезновение конечностей.

Отряд **Клювоголовые (Rhynchocephalia).** Этот отряд включает древнейшую группу рептилий, из которых до наших дней сохранился на небольших островах Новой Зеландии один вид — гаттерия. Это живое исскопаемое, по внешнему виду напоминающее ящерицу; длина ее тела достигает 50 см (см. рис. 242). Однако у гаттерии сохранился ряд примитивных черт строения. Сверху туловище и голова покрыты мелкими зернистыми чешуйками. По хребту тянется киль из треугольных роговых пластинок. Позвоночник образован двояковогнутыми позвонками, между телами которых всю жизнь сохраняются остатки хорды. На брюхе под кожей расположены брюшные ребра, которые, по-видимо-

му, являютсяrudиментамибрюшногокостногощита палеозойских амфибий — стегоцефалов. Есть нёбные и сошниковые зубы, развит теменной глаз (роговица, хрусталик, сетчатка). Это малоподвижное животное ведетночнойобразжизни, живетвнорах буревестниковидругихптиц. Питаетсячервями, насекомыми и улитками. Гаттерия строгоохраняется.

Отряд Чешуйчатые (Squamata). Наиболее многочисленная группа современных пресмыкающихся. Роговая чешуя налегает друг на друга, как чешуйки в еловой шишке, клоака в виде поперечной щели. Копулятивные органы в виде парных мешковидных выпячиваний клоаки. Распространены по всем континентам. Отряд делится на два подотряда: Ящерицы (*Lacertilia*) и Змеи (*Serpentes*).

Подотряд Ящерицы (*Lacertilia*). К этому подотряду относятся животные с продолговатым телом и обычно длинным хвостом. Внешний вид и окраска разнообразны. Ноги, как правило, хорошо развиты, но у некоторых форм они редуцированы или совсем отсутствуют (веретеница, желтопузик); в последнем случае в теле сохраняютсяrudиментыпоясов конечностей (см. рис. 242). Внешне такие ящерицы похожи на змей, но в отличие от последних у безногих ящериц сохраняются грудина и пояса конечностей. Многим видам свойственна автотомия — обламывание части хвоста. В последующем хвост восстанавливается, но его скелет не окостеневает.

Глаза у большинства видов снабжены подвижными веками, но у гекконов, гологлазов и некоторых других ящериц они срастаются и превращаются в прозрачные пленки на глазах. Имеются барабанные перепонки.

Большинство ящериц размножаются, откладывая яйца, но некоторые виды яйцеживородящи (веретеница, живородящая ящерица). Питаются мелкими животными, в том числе различными насекомыми и моллюсками, чем приносят пользу сельскому и лесному хозяйствам. Ядовитых видов среди ящериц нашей фауны нет.

В РФ наибольшее число видов ящериц обитает на юге страны. Но некоторые из них, как, например, живородящая и прыткая ящерицы (*Lacerta vivipara*, *L. agilis*), распространены далеко на севере.

Ящурки (*Eremias*) довольно обычные представители, обитающие в степной зоне и более южных районах страны. Размеры до 20 см.

Гекконы (сем. Gekkonidae) — мелкие и наиболее примитивныеочные ящерицы. Хорошо лазают по стенам, скалам, деревьям. Могут бегать по вертикальным гладким поверхностям и по потолку.

Агамы (сем. Agamidae) — мелкие и средние ящерицы с гибким и неломким хвостом, ведущие наземный и древесный образ жизни. В пустынях Средней Азии обычны круглоголовки (*Phrynocephalus*), отличающиеся округлой головой на подвижной шее: тело их покрыто мелкими роговыми бугорками.

Вараны (*Varanus*) — очень крупные (до 4 м) стройные ящерицы. Бегают быстро, держа тело высоко поднятым над землей. Наиболее крупный — обитающий на о. Комodo (Индонезия) комодский дракон.

Хамелеоны (сем. Chamaeleonidae) — высокоспециализированные **реп**тилии, приспособленные в основном к древесному образу жизни; **длина** тела составляет от 3 до 60 см (см. рис. 242). Несколько десятков видов хамелеонов обитают в субтропиках и тропиках (обыкновенный, щеломоносный, Джексона). Тело может сильно раздуваться за счет легочных мешков. На конечностях первые два пальца срослись между собой и противопоставлены также сросшимся трем остальным пальцам. Такими клещеобразными лапами хамелеоны крепко охватывают ветку **Мрева**, по которой передвигаются. Хвост длинный, цепкий. Глаза могут двигаться несогласованно. Окраска хамелеонов может меняться в зависимости от фона окружающей среды. Язык длинный, до половины длины тела; выбрасывая его, хамелеоны ловят насекомых.

Подотряд Змеи (Serpentes). Относящиеся к этому подотряду рептилии **Имеют** длинное цилиндрическое тело со слабо выраженным делением на **Голову**, шею, туловище и хвост. Самые крупные представители — питоны и удавы (длиной до 11 м). Ползают изгибая тело. Щитки, покрывающие **брюхо**, налегают друг на друга задними краями и препятствуют скольжению тела назад.

Характерно отсутствие подвижных век, барабанных перепонок, **Плечевого** пояса, парных конечностей и их скелета. Только у удавов **Имеются**rudименты задних ног, бедер и подвздошных костей. Тело **Мей** покрыто роговой чешуйей и щитками. Кожных желез нет (есть **лишь** у некоторых видов ужей). При линьке старый поверхностный **взор** кожи отделяется на челюстях и постепенно сходит с тела, выворачиваясь, как перчатка.

Глаза змей покрыты прозрачной пленкой сросшихся век. Барабанный перепонки нет. Кости обеих челюстей и нёба соединены связками, **И пасть** змей может открываться настолько широко, что они могут **зачитывать** крупную добычу целиком. В связи с вытянутой формой тела у змей развито только одно правое легкое, а левое, если оно есть,rudиментарно. Мочевого пузыря нет, почки и гонады сильно вытянуты.

Позвоночник представлен большим числом (200—450) однообразных позвонков, которые помимо обычных сочлененных отростков, **Имеют** на верхних дугах по срединному выступу. Позвонки несут **Одно** заканчивающиеся ребра, которые наружными концами упираются в слой мускулатуры. Движение ребер обеспечивает более совершенное перемещение тела, особенно в узких пространствах.

¹ Питаются исключительно животной пищей — от мелких насекомых **до** мелких копытных. Добычу заглатывают целиком, этому способствует подвижное сочленение левой и правой половин челюстного аппарата. Некоторые змеи (ужи) заглатывают живую добычу, другие (удавы) **душат** ее кольцами тела, третья (ядовитые змеи) убивают ядом. У ядовитых змей несколько передних зубов имеют более крупные размеры, они располагаются на верхней челюсти (рис. 243). У одних ядовитых змей (гадюки, гремучие змеи) яд стекает по каналу внутри зуба, **у других** (кобра, аспиды) — по борозде на его поверхности. Яд вырабатывается особыми верхнечелюстными железами. Укус широко распро-

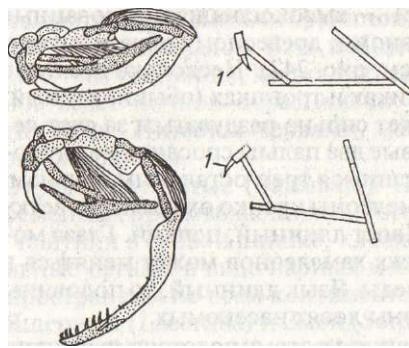


Рис. 243. Череп гремучей змеи с закрытой И
открытой пастью:
1 — ядовитые зубы

странной у нас обыкновенно!) гадюки (*Vipera berus*) болезнен, и ни не смертелен. Распространены во многих районах страны меди п \leq ка и ужасные полозы ядовитых зубов не имеют.

Змеи наиболее распространены в тропиках и субтропиках, хотя гадюка обитает и за полярным

кругом. Некоторые виды живут на деревьях, в воде и под землей. Размножаются змеи, откладывая яйца, для некоторых характерно яйцеж и ворождение. На зиму змеи нашей фауны впадают в спячку.

ПОДКЛАСС АРХОЗАВРЫ (ARCHOSAURIA). Отряд Крокодилы (Crocodylia). Это наиболее высокоорганизованная группа крупных пресмыкающихся (примерно 25 видов), отдельные представители которой могут достигать 8—10 м в длину (гребнистый и нильский крокодилы); все крокодилы ведут полуводный образ жизни.

Отличаются крокодилы уплощенным туловищем, вытянутой головой, короткими лапами и мощным длинным хвостом, сжатым с боков (см. рис. 242). Передние лапы с пятью свободными пальцами, задние несут четыре пальца, соединенных перепонкой. Тело покрыто роговыми пластинами, под которыми развиваются костные щитки. Имеют, хотя и немногочисленные, кожные железы, расположенные на хребте, под нижней челюстью и в области клоаки. Пасть снабжена многочисленными крупными зубами, сидящими в альвеолах. Сердце четырехкамерное. Однако наряду с правой дугой аорты, выходящей из левого (arterialного) желудочка, имеется и левая дуга, выходящая из правого (венозного) желудочка. Объемистые легкие имеют губчатое строение. Вторичное нёбо отделяет носоглоточный ход от ротовой полости. От заднего края нёба свешивается мускульная складка — нёбная завеса, которая, закрывая глотку, позволяет крокодилам хватать добычу под водой, изолирует ротовую полость от глотки, давая возможность дышать, когда рот в воде раскрыт, а наружу выставлен лишь конец морды с ноздрями. У ноздрей есть клапаны, закрывающиеся в воде. Большую часть времени крокодилы проводят в воде; они хорошо плавают и ныряют, но нередко вылезают на берег погреться на солнце. Яйца зарывают в прибрежный песок, где и проходит их развитие. Нередко самка охраняет кладку.

Питаются крокодилы как водными животными, так и приблизившимися к водоему. Крупные крокодилы опасны для человека.

В ряде стран ведется промысел крокодилов, в первую очередь ради ценной шкуры и мяса. Во многих странах крокодилов разводят с этой целью на специальных фермах.

ПРОИСХОЖДЕНИЕ ПРЕСМЫКАЮЩИХСЯ

Десятки миллионов лет назад, в мезозойскую эру, на Земле господствовали многие виды пресмыкающихся. Они были представлены разнообразием форм и на суше, и в воде. Рептилии ведут род гип и повную от древней палеозойской группы котилозавров, которая, в свою очередь, связана происхождением со стегоцефалами — панцирь-мешковыми земноводными. Подобно последним котилозавры тоже имели цельночелюстными, имели массивное тело и пятипалые конечности. Формирование котилозавров относится к концу каменноугольно-

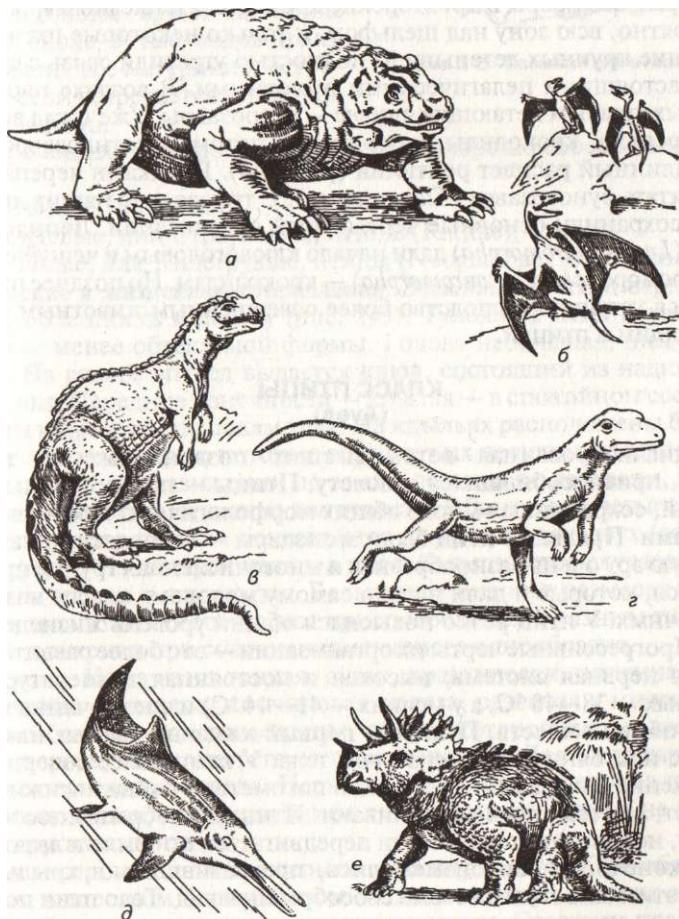


Рис. 244. Ископаемые пресмыкающиеся мезозоя:
а — котилозавр; б — птерозавр; в — цератозавр; г — псевдоузухий; д — ихтиозавр; е — триитиопонс

го периода палеозойской эры и связано с изменением климатических условий на больших пространствах суши: сменой теплого и влажного климата на более засушливый континентальный. Первые пресмыкающиеся возникли в конце палеозоя, но по-настоящему распространялись в триасе, а расцвет их наблюдался в юрском и меловом периодах. По берегам водоемов бродили гигантские бронтозавры. В водоемах около берегов жили огромные диплодоки, длина тела которых достигала 25 м. На суше паслись трицератопсы. Верхушками деревьев лакомились ходившие на задних ногах колоссальные игуанодоны. Расти тельноядными питались хищные цератозавры. В море носились косяки похожих на дельфинов ихтиозавров, плавали огромные плезиозавры мезозавры, мозазавры и др. Морские крокодилы и плезиозавры освоили, вероятно, всю зону над шельфом, и только некоторые ихтиозавры (рожавшие крупных детенышей) полностью утратили связь с сушей и стали настоящими пелагическими животными. В воздухе парили на кожистых крыльях летающие ящеры — птерозавры. Уже тогда встречались черепахи, крокодилы, ящерицы — потомки котилозавров. Это был подлинный расцвет рептилий (рис. 244). Предками черепах приято считать эунотозавра (*Eunotosaurus*). С триаса и до наших дней чешуякоши сохранили основные черты своей организации. Лепидозавроморфы (*Lepidosauromorpha*) дали начало клювоголовым и чешуйчатым, а архозавроморфы (*Archosauromorpha*) — крокодилам. Но позднее пресмыкающиеся уступили господство более совершенным животным — млекопитающим и птицам.

КЛАСС ПТИЦЫ (*Aves*)

Специализированная ветвь высших позвоночных из группы Amniota, приспособившихся к полету. Птицы — прогрессивная ветвь рептилий, сохранившая много общих морфологических особенностей с предками. Предками птиц были архозавры — господствующая в мезозойскую эру очень разнообразная и многочисленная группа пресмыкающихся, которые и дали начало самому молодому классу наземных позвоночных. У птиц резко повысился общий уровень жизнедеятельности. Прогрессивные черты их организации — это более развитая центральная нервная система; высокая и постоянная температура тела (у крупных — 38—40 °C, а у мелких — 41—44 °C) и чрезвычайно интенсивный обмен веществ. Птицы — первый класс изучаемых нами животных с постоянной температурой тела. У птиц более совершенно размножение и выраженная забота о потомстве — яйца насиживают и охраняют, а потомство выкармливают. Птицы приобрели способность к полету, не утратив способности передвигаться по земле и лазать. Передние конечности видоизменились, превратившись в крылья, что придает этим животным весьма своеобразный вид. Тело птиц покрыто перьями — черта, характерная только для представителей данного класса. Часть костей плюсны и предплюсны срослись и образовали единую кость — цевку. Череп сочленяется с позвоночником одним мы

Щелком. В полушариях мозга имеются кора, поверхность которой редкая. Мозжечок хорошо развит.

Все губчатые, соединены с системой воздушных мешков. Сердце трехкамерное. Имеется только один дуга аорты. Размножаются, складывая яйца. Способность к Полету наложила отпечаток как на **Фрсные** птиц, так и на их биологии, у птиц все подчинено полету.

В настоящее время на Земле существует около 9 тыс. видов птиц, населяющих все материками и острова. В России встречается примерно 730 видов птиц.

Современных птиц подразделяют на три хорошо обоснованных надстрида:

Лингины (Impennes);

Бескилевые, или Страусовые, птицы (Ratitae);

Типичные, или Кильгрудые, птицы (Neognathae, или Carinatae).

Строение и жизненные отправления. Внешний вид птиц отражает их приспособленность к полету (рис. 245). Туловище птиц компактное, более или менее обтекаемой формы. Голова небольшая, шея тонкая, гибкая. На голове вперед выдается клюв, состоящий из надклювья и подклювья. Передние конечности — крылья — в спокойном состоянии вложены и прижаты по бокам тела. На крыльях расположены большие упругие маховые перья, которые образуют их несущие плоскости. Вся масса тела при передвижении по земле, лазанью по деревьям, взлете и посадке приходится на задние конечности. Обычно они четырехпалые, но иногда число пальцев сокращается до трех и даже двух (африканский страус). Из четырех пальцев ног в большинстве случаев три направлены вперед, а один назад. Большие вариации в размерах и форме отделов тела птиц обеспечивают приспособления к разным типам питания и движения при сохранении внешнего однообразия.

Покровы. Кожа птиц тонкая, сухая, практически лишенная желез. Исключение составляет копчиковая железа, развитая у многих видов птиц, особенно у водоплавающих, секрет которой используется для мытья перьев, что препятствует их намоканию. Для птиц характерно наличие перьевого покрова. Перья эволюционно развились из чешуй пресмыкающихся, они характерны для всех видов птиц и не встречаются у других животных.

Перо — производное эпидермиса кожи (рис. 246). Оно образовано роговым веществом — кератином. Отдельное перо состоит из очина (часть, погруженная в кожу), стержня и опахала. Стержень представляется собой плотную роговую трубку с рыхлой роговой сердцевиной. Опахало образовано отходящими от стержня в обе стороны бородками пер-



Рис. 245. Внешний вид птицы (луна)

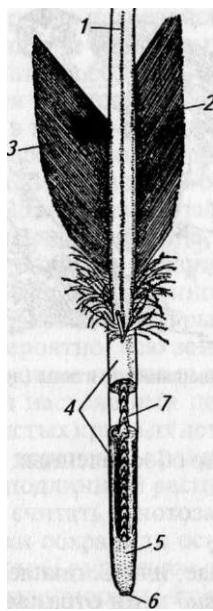


Рис. 246. Строение пера птицы:
1 — стержень; 2 — наружное опахало; 3 — внутреннее опахало; 4 — ствол; 5 — очин; 6 — отверстие очина; 7 — дужка

ными маховыми, а меньшие по размерам и не столь упругие перья, соединенные с костями предплечья, — второстепенными маховыми (рис 247). Рулевые перья, слагающие хвост и направляющие полет птиц, отличаются большими размерами, упругостью и симметрией опахал. Пол контурными перьями располагаются пуховые (опахало без бородок второго порядка — рассучено) и пух, а также имеются нитевидные перья и

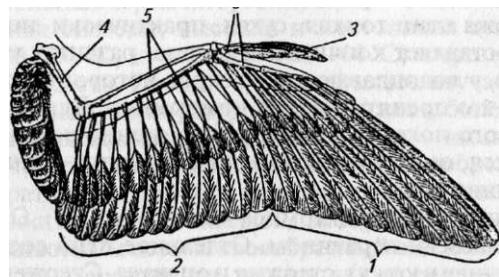


Рис. 247. Крыло птицы:
1,2 — маховые перья; 3 — крылышко; 4 — плечевая кость; 5 — кости предплечья; 6 — кости недоразвитой кисти

щетинки. Пуховые перья и пух, согревающие организм, особенно развиты у водоплавающих птиц.

Перья расположены в направлении от головы к хвосту, причем расположены не на всей поверхности тела Птицы, а на некоторых определенных участках, чередующихся с участиями и без перьев (рис. 248). Это касается в первую очередь контурных Перьев. Участки с перьями называются птерилиями, а без перьев — апториями. У пингвинов и бескилевых илптерий нет. Аптерии расположены по средней линии груди, в подмышечной области, вдоль лопаток, т. е. в тех местах тела, где кожа над мышцами напрягается при полете. Такое свое положение перьев улетающих птиц позволяет «закрыть» все тело меньшим числом перьев, а также является приспособлением для уменьшения трения отдельных участков тела при полете и увеличивает подвижность перьев на теле. Аптерии прикрываются соседними контурными Перьями. По-видимому, возникновение перьев (или пуха, подобного эмбриональному у современных видов) было вызвано требованиями терморегуляции у эндотермных животных при уменьшении их размеров.

Роль перьевого покрова в жизни птиц велика и разнообразна. Перьевий покров придает телу птиц обтекаемую форму, что облегчает полет. Маховые и рулевые перья образуют большую часть несущей поверхности крыльев и хвоста, следовательно, они обеспечивают сам полет. Благодаря высоким теплозащитным свойствам перьев и воздушных прослоек между ними перьевой покров способствует сохранению тепла и таким образом участвует в терморегуляции. Он также защищает птицу от различных механических воздействий. Перья могут нести и сигнальную функцию, как визуальную (павлин), так и акустическую (бекас). Разнообразные пигменты перьев придают птицам ту или иную окраску, которая часто носит покровительственный характер.

Для птиц характерна регулярная частичная или полная смена перьевого покрова путем линьки. При этом старые перья выпадают, а на их месте развиваются новые (иногда другой окраски). Число линек варьирует от одной до нескольких, обычно один или два раза в год.

Для большинства птиц характерна медленная и постепенная смена перьевого покрова, благодаря чему они сохраняют способность к полету (орлы), но у водоплавающих птиц линька протекает столь быстро, что они временно не могут летать (гуси).

Большую роль играют сезонные линьки для зимующих птиц: так, у **(Шнога)** вида синицы летом насчитывается 1100 перьев, а зимой — 1700. У крупных видов птиц число перьев больше, чем у мелких, например: у **Килибри** — 1 тыс.; у чаек — 5—6; у уток — 10—12; у лебедей — 25 тыс.

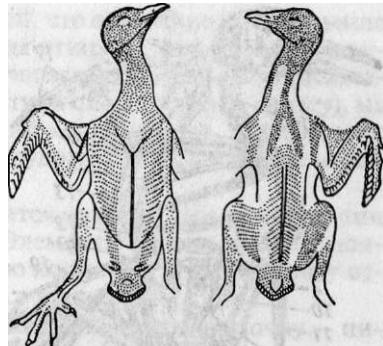


Рис. 248. Птерилии и аптерии птиц (голубя)

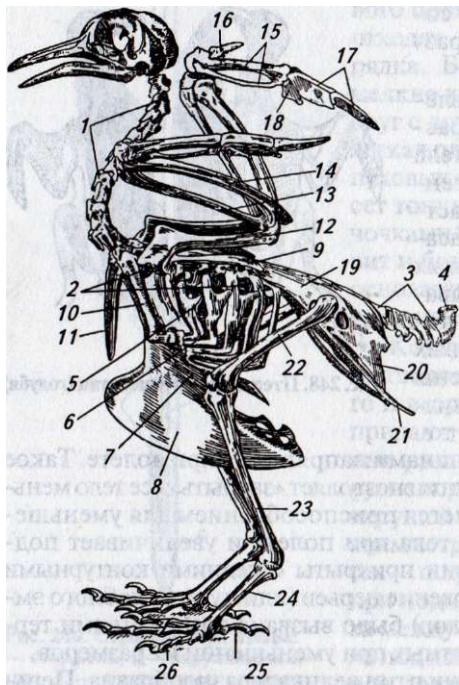


Рис. 249. Скелет птицы (голубя):
 1 — шейные позвонки; 2 — грудные позвонки; 3 — хвостовые позвонки; 4 — копчиковая кость; 5, 6 — ребра; 7 — грудинка; 8 — киль грудины; 9 — лопатка; 10 — коракоид; 11 — ключица (вилочка); 12 — плечевая кость; 13 — лучевая кость; 14 — локтевая кость; 15 — пясть; 16—18 — фаланги пальцев; 19—21 — тяжелые кости; 22 — бедренная кость; 23 — кость голени; 24 — цепка; 25, 26 — фаланги пальцев

К производным кожи относятся также роговой чехол клюва, когти, роговые щитки на цевке.

Для костей скелета птиц характерна прочность и легкость, что важно для полета (рис. 249). Легкость его достигается тем, что у птиц кости тонкие и трубчатые, а их полости заполнены воздухом. Прочность скелета в значительной степени обусловлена срастанием отдельных костей. Его относительная масса со-

ставляет — 10 % массы тела, как и у млекопитающих, поскольку у птиц очень длинные конечности — в 2—3 раза длиннее туловища. Рост костей у птиц ограничен.

Череп птиц характеризуется большой тонкостенной мозговой коробкой, огромными глазничными впадинами и беззубыми челюстями. Замещение зубов роговым клювом — рамфотекой — связано, по-видимому, с необходимостью облегчения конструкции для полета и с особенностями питания птиц. У взрослых птиц кости черепа полностью срастаются, что обеспечивает его прочность. Череп сочленяется с первым шейным позвонком одним мышцелком.

Позвоночник птиц состоит из пяти отделов. Высокая подвижность, столь необходимая для шейных позвонков, у грудных и поясничных отсутствует, туловищный отдел укорочен, часть поясничных и хвостовых позвонков функционирует как крестцовые. Грудные, поясничные крестцовые и передние хвостовые позвонки у взрослых птиц срастаются между собой, создавая жесткую опору работающим мышцам. Позвонки срастаются также с тонкими подвздошными костями таза, который служит прочной основой для ног. Ребра нижними концами прикрепляются к большой грудной кости: на заднем крае они имени крючковидные отростки, которые налегают концами на ребра следующей пары; это придает грудной клетке прочность. Каждое ребро соединяется с

ИТ из двух подвижно сочлененных частей, что обеспечивает **изменение объема** грудной клетки и дыхание, когда птица не летает, **за счет** поднятия и опускания грудины. Грудина птиц всегда окостеневает (**исключение** составляют птицы, которые утратили способность к полету), на Передней поверхности она несет высокий костный киль, к которому с **Обеих** сторон прикрепляются мощные грудные и подключичные **мышцы**, приводящие в движение крыло.

При ходьбе масса всего тела приходится на кости таза, поэтому они Очень прочно сочленены между собой. Элементы скелета тазового пояса* срастаются у взрослых птиц в единую кость. Таз остается снизу открытым, что позволяет им откладывать очень крупные яйца.

Задние хвостовые позвонки срастаются и образуют косточку — **пигментиль**. Она служит опорой для рулевых перьев.

Плечевой пояс состоит из трех пар костей: саблевидных лопаток, лежащих вдоль позвоночника, тонких ключиц, которые срастаются нижними концами в вилочку, расширяющую основания крыльев, и кора-Юидов — массивных костей, соединяющихся одним концом с лопатками и основаниями плечевых костей, а другим — с грудиной.

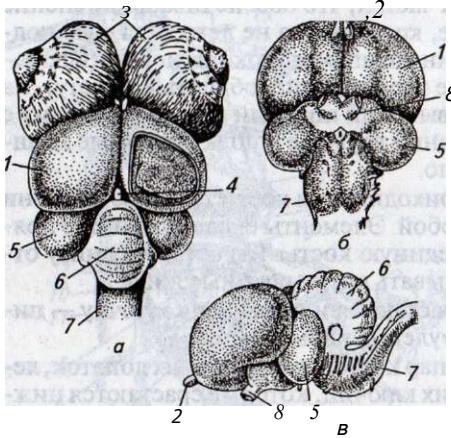
Скелет передних конечностей подвергся изменениям, из которых первые заметные — срастание костей пясти и сокращение числа фаланг Иищцев II, III и IV. Пальцы I и V редуцированы, палец II имеет только один фаланг, которая служит опорой для обособленного пучка перьев Ий внешнем крае крыла, так называемого крыльышка. Крыло до конца не расправляется, его отделы всегда расположены под углом друг к Другу и подвижны относительно друг друга только в горизонтальной Яйос кости, а в вертикальной сочленены жестко, что важно в полете для Цбгспечения опоры крыла о воздух.

Дни птиц характерна вертикальная постановка ног. В скелете задних (Винчестеров) малая берцовая кость сильно редуцирована и приращена к большей берцовой кости. В процессе онтогенеза к нижнему концу голени прирастают косточки основного ряда предплосны. Остальные косточки предплосны и три косточки плюсны сливаются в единую вытянутую рн'п. — цевку. К нижнему концу цевки прикрепляются фаланги пальцев.

Мускулатура птиц весьма своеобразна. Мышцы птиц хорошо разработаны на конечностях, в шейном и хвостовом отделах тела. Наиболее развитые мышцы и соответственно наиболее тяжелые (например, грудные) расположены на самом теле, от них к конечностям идут длинные и прочные сухожилия. Особенно развиты грудные и подключичные мышцы, приводящие в движение крылья. У хороших летунов 114'и фудных мышц может достигать 18—25 % массы тела. Мощные четырехглавые мышцы ног (бедра), выполняющие большую работу при хождении птицы и передвижении по веткам деревьев, при взлете и посадке. Овывание подкожные мышцы управляют положением перьев.

* Нервная система, особенно центральный отдел, у птиц имеет более яркое сформирование, чем у рептилий, что соответствует более высокому (Копию жизнедеятельности. В головном мозге хорошо развиты полулярные переднего мозга, хотя кора гладкая, без извилин; мозжечок

Рис. 250. Головной мозг птицы (голубя):
 а — сверху; б — снизу; в — сбоку; / полушиария переднего мозга; 2 — обонятельные доли; 3 — глазные яблоки, 4 — эпифиз; 5 — зрительные доли; 6 — мозжечок; 7 — продолговатый мозг; 8 — зрительные нервы



очень большой, сильно развиты зрительные бугры среднего мозга, обонятельные доли малы (рис. 250). Центр рассудочной деятельности локализуется у птиц в полосатых телях, а не в коре головного мозга. Сильное развитие зрительные бугры среднего мозга, несущих зрительную функцию, обусловлено доминированием в жизни птиц зрения. Мозжечок велик и имеет сложное строение. Его средняя часть — червь — передним краем почти соприкасается с полушариями, а задним концом прикрывает продолговатый мозг. Червь покрыт характерными поперечными бороздами. Большой мозжечок соответствует высокой подвижности и сложности движений птиц. От головной мозга отходит 12 пар нервов.

Органы чувств развиты у птиц в различной степени. Наибольшее значение в их жизни имеет зрение, поэтому большинство птиц — живота и с дневной активностью. Глаза велики, у некоторых видов составляют до 4 % массы тела и сложно устроены. Аккомодация происходит путем изменения кривизны хрусталика, изменения расстояния между хрусталиком и роговицей и изменения кривизны роговицы. Поле зрения велико; поле бинокулярного зрения значительно лишь у некоторых, например сои, веслоногих. У соколообразных разрешающая способность глаза в 4—8 раз превышает таковую у человека. Сокол видит добычу размерами с галку и на расстоянии 1 км. Соры видят при освещенности в сотни раз меньшей, чем человек. Располагаясь в черепе, глазные яблоки практически соприкасаются друг с другом и лишены мышц, поэтому они малоподвижны или не подвижны, зато очень подвижна голова. Слух развит хорошо, например соры при его помощи охотятся даже в темноте. Ошибка пассивной локации источника звука у сов не превышает 2°, чему способствуют, вероятно, наличие кожной складки и особая структура из перьев (лицевой диск), также присущая некоторым из видов асимметрия ушных отверстий. В среднем ухе имеется одна kostочка — стремя, барабанная перепонка и сколько углублена, хорошо развит наружный слуховой ход. У некоторых птиц хорошо развито и обоняние (гриф, киви).

Органы пищеварения начинаются ротовой полостью. Зубы у современных птиц отсутствуют, их частично заменяют острые края рогового чехля

Клюва, которым птица захватывает, удерживает и иногда размельчает пищу. Отсутствие зубов породило ряд других особенностей пищеварительной системы (рис. 251). Для птиц характерны пищевод и зоб (расширенная часть пищевода, свойственная многим птицам), где пища увлажняется, набухает и размягчается. Из пищевода пища попадает в железистый (идет желудка, где смешивается с пищеварительными соками. Химическая и обработка начинается уже в зобе и продолжается в железистом отделе желудка. Из него пища переходит в мышечный отдел желудка. Стенки его образованы мощными мышцами, а в полости, выстланной твердой оболочкой, обычно находятся мелкие камешки (гастролиты), проглоченные Птицей. Эти камешки и складки стенок желудка при сокращении мышц отсюда перетирают пищу. Таким образом, желудок птиц состоит из двух отделов. Сила мышечного желудка так велика, что могут перетираться самые жесткие плоды с твердой кожурой. Так, у кур давление в мускульном отделе достигает 100—150 мм рт. ст., у гусей — 265—286 мм рт. ст.

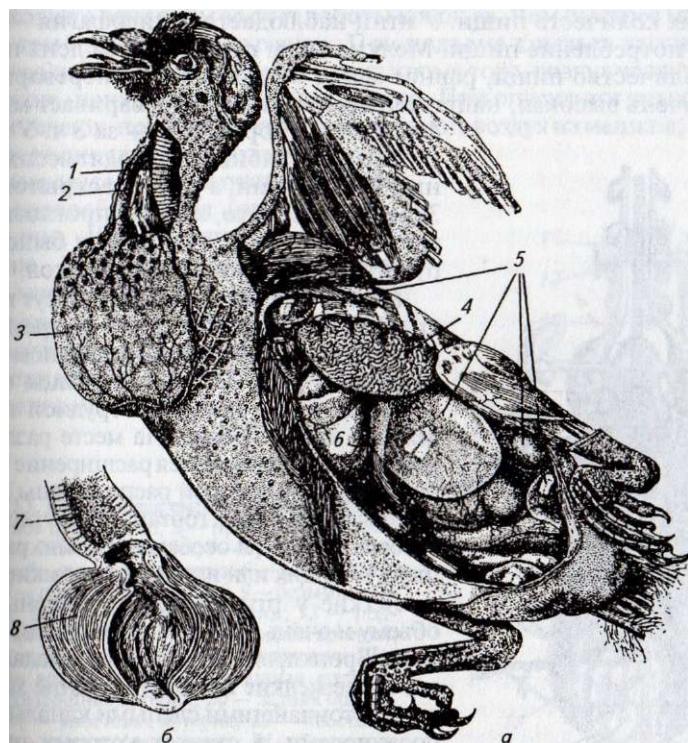


Рис. 251. Внутреннее строение птицы:
» в крытый голубь; б — разрез желудка голубя; 1 — трахея; 2 — пищевод; 3 — зоб;
4 — кос; 5 — воздушные мешки; 6 — сердце; 7 — железистый желудок; 8 — мускульный
МНУ/ПК

Кишечник птиц относительно короткий за счет уменьшения длины толстого отдела. Длина кишечника в 3—12 раз больше длины тела. Ни границе тонкого и толстого отделов от кишечника отходят два слепых выроста. Прямая кишка не развита, поэтому экскременты не накапливаются в кишечнике, что облегчает массу птицы. Заканчивается кишечник расширением — клоакой, в которую открываются мочеточники и протоки половых желез. Секрет двулопастной печени и поджелудочной железы, поступающий в двенадцатиперстную кишку, способствует перевариванию пищи. Печень у птиц большая, ее масса может составлять 2—8 % массы тела; это позволяет запасать энергетические резервы углеводов (в первую очередь гликогена), ведь полет требует по вышенного в 2—10 раз расхода энергии. Энергия запасается и в виде жировых отложений, по способности накапливать жир (запас энергии) птицы в 2 раза превосходят любых других животных. Во времена миграций запасы жира могут составлять 30—50 % массы тела. Затрат птицами во время полета огромного количества энергии и высокий уровень обмена веществ обусловливают необходимость поглощении больших количеств пищи. У птиц наблюдается гиперфагия — избыточное потребление пищи. Мелкие виды птиц могут за день потреблять количество пищи, равное массе тела. Скорость переваривания пищи очень высокая, например домовый сыч переваривает мышь за 4 ч, а серый сорокопут — за 3 ч. У свиристики ягоды рябины проходят весь кишечник за 8—10 мин, а у утки, вскрытой через 30 мин после того, как она проглотила кмра длиной 6 см, уже нельзя было обнаружить в кишечнике его остатков.

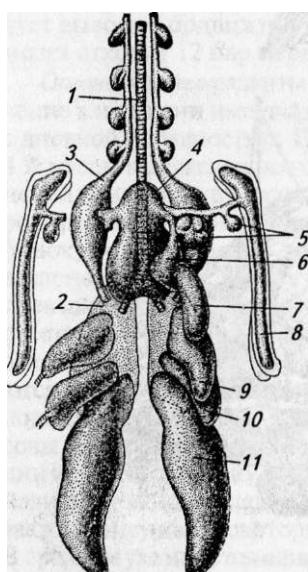


Рис. 252. Органы дыхания птицы (голубя):
1 — трахея; 2 — легкие; 3—11 — воздушные мешки

Органы дыхания птиц также несут признаки приспособления к полету, во время которого организм нуждается в усиленном газообмене (рис. 252). От глотки птицы отходит длинная трахея, которая в грудной полости делится на два бронха. На месте разделения трахеи на бронхи имеется расширение — нижняя гортань, в которой расположены голосовые связки. Нижняя гортань играет роль голосового аппарата и особенно сильно развита у птиц, поющих или издающих громкие звуки.

Легкие у птиц губчатые, маленькие по объему и очень большие по площади газообмена. Бронхи, входя в легкие, распадаются на все более мелкие ветви. Последние заканчиваются тончайшими слепыми канальцами — бронхиолами, в стенках которых проходит кровеносные капилляры.

Часть разветвлений бронхов проникает в легкие, продолжаясь в тонкостенные

воздушные мешки, которые располагаются между органами и мышцами, под кожей и в полостях трубчатых костей крыльев. Эти мешки играют большую роль в дыхании птицы во время полета. У сидящей птицы дыхание осуществляется путем расширения и сжатия грудной клетки и результате движения грудины. В полете же, когда движущимся крыльям нужна твердая опора, грудная клетка остается почти неподвижной и прохождение воздуха через легкие обусловливается в основном расширением и сжатием воздушных мешков. Этот процесс получил название «двойное дыхание», поскольку отдача кислорода в кровь происходит как при вдохе, так и при выдохе. Полнота извлечения кислорода из воздуха при этом способе дыхания такова, что гуси способны достигать высоты 10 км. Воздушные мешки также участвуют в терморегуляции, уменьшают трение между органами и мышцами, способствуют акту дефекации. Путем изменения объема воздушных мешков ныряющие птицы регулируют свою плавучесть.

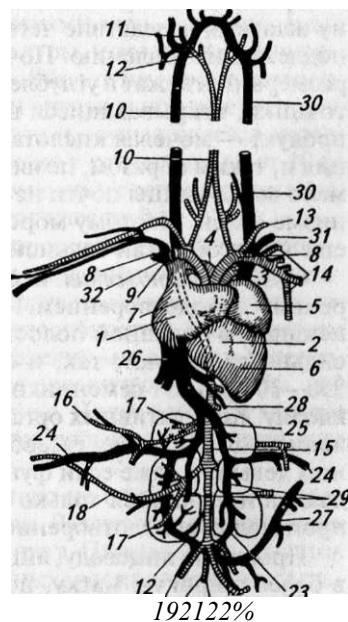
Объем воздушных мешков примерно в 10 раз больше объема легких, Мо газообмен осуществляется только в легких. Само дыхание в полете Происходит в основном за счет работы крыльев. Чем быстрее машущий Полет, тем интенсивнее дыхание. При подъеме крыльев легкие растягиваются и воздух засасывается в них; кроме того, воздух частично, ми-Нум легкие, попадает в воздушные мешки. При опускании крыльев про-исходит выдох, причем через легкие проходит воздух из мешков, что спо-Юбствует окислению крови в легких.

Кровеносная система птиц имеет два круга кровообращения (рис. 253; см. также рис. 192). Сердце птиц очень большое, у колибри оно составляет 1,7 % массы тела, у человека — 0,42 %.

Сердце у птиц четырехкамерное, **поэтому** кровь полностью разделяется на артериальную и венозную. От каждого желудочка отходит по одному со-

ГиI 253. Кровеносная система птиц (голубя):

- I — правое предсердие; 2 — правый желудочек;
- левая легочная артерия; 4 — правая легочная Ци^{нги}я; 5 — левое предсердие; 6 — левый желу-
чичгк; 7 — правая дуга аорты; 8, 9 — безымянные Пи^ии; 10—12 — сонные артерии; 13 — под-
чечничная артерия; 14 — левая грудная артерия;
- J — порта; 16 — правая бедренная артерия; 17 —
Почекчая артерия; 18 — седалищная артерия;
- IV — подвздошная артерия; 20 — задняя брыже-
кни^и; 21 — хвостовая артерия; 22 — хвостовая
ltMii; 23 — воротная вена почек; 24 — бедренная
Ци^и; 25 — подвздошная вена; 26 — задняя полая
Ци^и; 27 — кишечная вена; 28 — надкишечная ве-
ни^и 29 — почечная вена; 30 — яремная вена; 31 —
(Иимючинская вена; 32 — передняя полая вена



СУДУ — общий ствол легочных артерий от правого желудочка и правая дуга аорты от левого. В эритроцитах сохраняются ядра.

У зародышей птиц, как и у зародышей пресмыкающихся, закладываются и левая, и правая дуги аорты, но в процессе эмбрионального развития у птиц левая дуга атрофируется. Артериальная кровь, идущая от легких по легочной вене, попадает в левое предсердие, а оттуда в левый желудочек, из которого уходит в аорту. Венозная кровь со всего тела поступает в правое предсердие, а из него в правый желудочек, затем по легочным артериям к легким.

Правая дуга аорты изгибается вправо и поворачивает назад. На изгибе от нее отходят большие парные безымянные артерии, которые разделяются на сонные артерии, несущие кровь к голове, и грудные и подключичные артерии, идущие к грудным мышцам и крыльям. Дуга продолжается стволов аорты, тянущимся под позвоночником. От ствола аорты отвечаются артерии к различным частям туловища птиц и к ногам. Венозная система птиц в основном сходна с венозной системой пресмыкающихся.

Высокая активность обменных процессов у птиц делает необходимым быструю и обильную доставку питательных веществ и кислорода во все части организма, поэтому кровь по сосудам течет с большой скоростью, что обеспечивается энергичной работой сердца. Частота сокращений сердца высокая, например пульс у серебристой чайки в полете 130—250, в машущем полете 625 ударов в 1 мин; у воробья в покое 460, а в полете до 1000 ударов в 1 мин.

Органы выделения птиц также приспособлены к интенсивному обмену веществ, вследствие чего увеличивается объем продуктов распада, подлежащих удалению. Почки у птиц тазовые, отличаются большими размерами и лежат в углублениях тазовых костей. От них отходят мочевые протоки, открывающиеся в клоаку. Мочевого пузыря нет, конечный продукт — мочевая кислота, почти не требующая воды для ее выведения и, таким образом, позволяющая экономить воду. Выделяя с мочой мало воды, птицы почти не способны избавиться и от избытка в организме солей. Поэтому морские птицы, пьющие соленую воду, имеют еще и другой орган соляной экскреции — носовую железу.

Органы размножения. Все птицы — яйцекладущие животные с внутренним оплодотворением. Птицы раздельнополы. Парные семенники, лежащие в брюшной полости, имеют бобовидную форму. Размеры их сильно варьируют; так, в сезон размножения они увеличиваются и на 250—300 раз. От семенников отходят семяпроводы, открывающиеся в клоаку. Копулятивных органов нет у многих видов, за редким исключением (бескилевые, гусеобразные). Яичник функционирует в оспойном левый, и даже если функционируют оба (попугаи), то яйца попадают в имеющийся только левый яйцевод, в верхней части которого происходит оплодотворение.

Пройдя по яйцеводу, яйцо приобретает белковую оболочку, а поит в более широкую матку, покрывается известковой скорлупой. Череп конечный отдел половых путей самки — влагалище — яйцо попадает и

клоаку, а оттуда выводится наружу. Плодовитость невысокая — 1—20 яиц в кладке. Хотя некоторые виды за сезон размножения откладывают значительно больше яиц, **тих как** делают несколько кладок (**воробыи, синицы**). При непрерывной яйцекладке промежуток между внесением яиц может колебаться от **1 до 3** сут, причем большую часть времени занимает образование скорлупы.

Яйцо птиц по сравнению с размерами самки очень крупное, поскольку содержит много питательных веществ в виде желтка и белка (рис. 254). По своему строению яйцо птиц практически не отличается от яйца рептилий. Зародыш развивается из небольшого зародышевого диска, находящегося на поверхности желтка. На тупом конце яйца под скорлупой и подскорлуповой оболочкой находится полость, заполненная воздухом; она помогает зародышу дышать.

Для развития яйца птиц необходима температура, сравнимая с температурой тела самой птицы. Поэтому птицы насиживают яйца, контактируя с ними специально оголяющимися участками тела — насековыми личинками. Не насиживают яйца лишь сорные куры, использующие внешние источники тепла, и гнездовые паразиты. Яйца птиц (кроме яиц сорных кур) нуждаются не только в поддержании оптимальной температуры, но и в систематическом переворачивании, что и осуществляется наседкой. Продолжительность инкубации колеблется у разных видов от 10 до 100 сут. Незадолго до вылупления птенцы переходят на легочное дыхание, т.е. делают возможным акустический контакт с насекомой.

Экология птиц. Основная форма передвижения большинства птиц — **полет**. Приспособление к полету вызвало ряд описанных выше морфологических и физиологических изменений, а также наложило отпечаток на все виды их жизнедеятельности. Благодаря способности к полету птицы обладают огромными возможностями для далеких миграций и расселения: именно полет позволил им заселить все океанические острова, недалеко лежащие в сотнях километров от материка. Полет помогает птицам избегать врагов. Многие птицы во время полета добывают пищу в воздухе или высматривают ее на земле.

Характер полета разных видов птиц далеко не одинаков, он всегда зависит в соответствии с их образом жизни. Различают два основных (ИЛ) полета птиц — парение и машущий полет. Парением называется полет птицы на более или менее неподвижных, распростертых крыльях, при этом птица постепенно снижается. Но часто парящая птица мо-

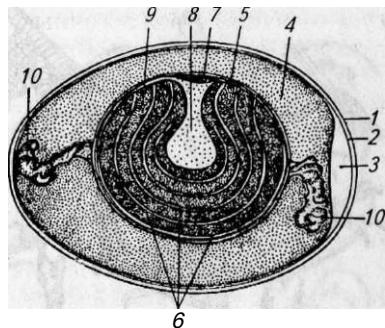


Рис. 254. Строение яйца птицы:
1 — скорлупа; 2 — подскорлуповая оболочка; 3 — воздушная камера; 4 — белок; 5 — желточная оболочка; 6 — желток; 7 — зародышевый диск; 8 — белый желток; 9 — желтый желток; 10 — халазы

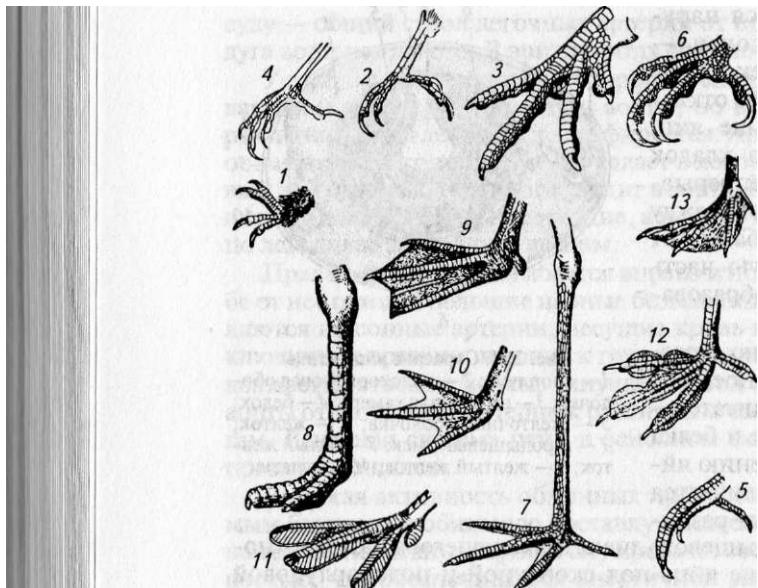


Рис. 255. Ноги птиц:

1 — стрижка; 2 — дятла; 3 — фрегата; 4 — дрозда; 5 — зимородка; 6 — сокола; 7 — аиста; 8 — страуса; 9 — крохалия; 10 — шипоклювки; 11 — чомги; 12 — лысухи; 13 — фаэтона

жет сохранять набранную над землей высоту или даже подниматься вверх, используя восходящие потоки воздуха. При машущем полете птица активно взмахивает крыльями. У многих птиц такая активная форма полета чередуется с парением. Ворона при спокойном машущем полете делает в среднем 2,9, а чайка — 2,2 взмаха в 1 с. Максимально возможная скорость полета ласточки 28 м/с, глухаря — 16 м/с, лебедя — 14 м/с. Некоторые птицы могут лететь без остановки для дыха более 3 тыс. км. Высокое развитие опорно-двигательной системы находит свое отражение в том, что птицы — самые быстрые из всех животных. Абсолютный рекорд скорости — 300 км/ч — принадлежит сапсану.

Задние конечности хорошо адаптированы к быстрой наземной локомоции и обычно имеют четыре пальца, из которых первый направлен назад (рис. 255). Он может сильно уменьшаться (гусеобразные). Лапки стрижей, адаптированные к цеплянию за неровности, а не к схватыванию веток, имеют четыре коротких пальца, направленных вперед. Ни и четыре пальца направлены вперед и соединены плавательной перепонкой у веслоногих. У попугаев и дятлов два пальца направлены вперед, а два — назад. Как приспособление к бегу число пальцев уменьшается до трех у нанду и казуаров и даже до двух — у африканских страусов.

Для птиц характерно огромное число врожденных поведенческих актов на все случаи жизни, а способность к обучению играет в жизни большинства птиц меньшую роль, чем у млекопитающих. Обитание одних и тех же условиях нередко приводит к развитию у птиц разного

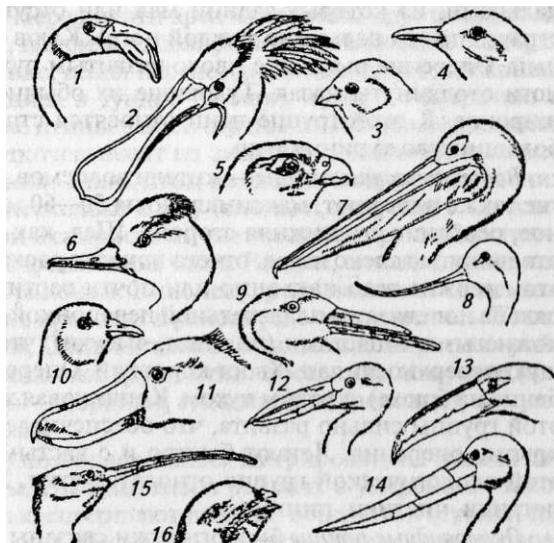


Рис. 256. Клювы птиц:
 1 — фламинго; 2 — колпачки; 3 — овсянки; 4 — дрозда;
 5 — сокола; 6 — крохалия; 7 — пеликана; 8 — шилоклювки;
 9 — водореза; 10 — голубя;
 11 — китоглава; 12 — аисти-разини; 13 — тукана;
 14 — аиста сенегальского;
 15 — ибиса; 16 — стрижка

происхождения и систематического положения сходных признаков, отражающих приспособления к данным условиям существования. Различают несколько экологических групп птиц.

Древесные птицы проводят большую часть жизни на деревьях и кустах. Кормятся преимущественно на деревьях, кустарниках и других растениях. Здесь и гнездятся. Питаются различными насекомыми, ягодами, плодами, семенами, нектаром и другими видами кормов. Клюв обычно небольшой (рис. 256). Для этих птиц, среди которых преобладают мелкие и средние по размерам, характерно стройное, вытянутое, скатое с боков тело, средней длины шея, довольно короткие тонкие ноги, пальцы которых не связаны перепонкой и несут острые когти. Из четырех пальцев обычно три направлены вперед, а один — назад; реже наружный палец может быть обращен вперед и назад или два пальца. Могут быть направлены вперед, а два — назад. Задний палец хорошо развит, так как он позволяет обхватывать ветви. Крылья не очень длинные. Оперение довольно рыхлое, у большинства видов со слаборазвитым пухом. У некоторых птиц этой группы (дятел, пищуха) хвост обращен из упругих жестких перьев, служащих опорой при передвижении по стволам деревьев. Характерные представители данной группы — Опыпинство певчих воробынных, кукушки, дятлы и др.

Наземные птицы держатся преимущественно на земле и передвигаются главным образом бегом или шагом. Крылья чаще короткие и широкие, иногда в той или иной степени редуцированные. Обычно плодоеды* летуны, полет без парения. Зато ноги мощные, с тупыми когтями,

приспособленными к хождению и копанию, с короткими толстыми пальцами, из которых задний мал или отсутствует (у африканского страуса по два пальца на каждой ноге). Клюв обычно короткий, мощный. Оперение рыхлое, с плохо развитым пухом, тело сжато с боков, ноги отодвинуты назад. Туловище их обычно массивное, плотное и широкое. К этой группе птиц относятся страусы, дрофы, стрепеты, большинство куриных и др.

Водоплавающие птицы — жители водоемов. Хорошо плавают, а многие также и ныряют, максимально на 50—60 м. Тело обычно уплощенное, особенно его нижняя сторона. Шея, как правило, длинная. Ноги отодвинуты далеко назад, отчего походка раскаивающаяся, птица при этом держит тело наклонно или почти вертикально. Три или четыре пальца ног связаны плавательной перепонкой или оторочены по бокам кожистыми лопастями (поганки, лысухи), что увеличивает загребающую поверхность лап. Хвост короткий. Оперение очень плотное (особенно на брюхе) с густым пухом. Копчиковая железа почти у всех видов этой группы сильно развита, что обеспечивает обильное жировое покрытие оперения. Летают быстро и с частыми взмахами крыльев. К этой экологической группе относятся утки, гуси, пеликаны, гагары, поганки, чистики, пингвины.

Околоводные птицы биологически связаны с водоемами. Они хорошо плавают, но обычно не ныряют. Обитатели болот, сырых лугов, побережий водоемов. Эти птицы чаще всего бродят по мелководью или по топкой сырой почве. Тело легкое, стройное. Шея обычно длинная, гибкая. Голова небольшая, у подавляющего числа видов с длинным тонким клювом, или клювом, приспособленным к выхватыванию добычи из воды; он крепкий, удлиненный, часто с крючком на конце. Хвост короткий или средней длины. Оперение рыхлое со слабо развитым пухом. У многих ноги длинные, тонкие с удлиненными пальцами, не связанными плавательной перепонкой, сочленение цевки и голени не оперено, иногда пальцы соединены плавательной перепонкой. Крылья длинные, острые, упругие или широкие. Характерные представители — цапли, аисты, журавли, кулики, чайки, крачки, буревестники.

Летающие птицы почти весь день обычно проводят в воздухе, где и добывают пищу. Тело их мускулистое, вытянутое. Шея короткая. Голова небольшая. Клюв большой или короткий, широко раскрывающийся, что облегчает хватание насекомых на лету. Крылья узкие, мощные, упругие, приспособленные к парению. Полет быстрый, в основном планирующий и пикирующий. Хвост небольшой. Ноги слабые, короткие, сцепляющимися пальцами; или сильные хватающие (соколы). Оперение плотное. Характерные представители группы — ласточки, стрижи, козодои, орлы, ястребы, альбатросы.

Птицы распространены по всему земному шару. Некоторые из них имеют огромные ареалы, другие — очень незначительные.

Среди птиц можно выделить три группы видов, имеющих различный характер пребывания в той или иной местности. Перелетные, которые весной и осенью совершают весьма далекие перелеты, и у них.

Кик правило, имеется два ареала — летний гнездовой и зимний (аисты, **листоочки**, голуби, гуси). Нередко эти ареалы удалены друг от друга на **тысячи** километров. Так, например, полярная крачка гнездится в Арктике, а зимует в Антарктике, делая перелет до 20 тыс. км в один конец. **Кулик**-щеголь, гнездящийся в тундре Сибири, летит в Австралию и **11овую** Зеландию. Многие птицы нашей страны не совершают далевых сезонных миграций, а откочевывают на зиму лишь несколько южнее области своего гнездования. Таких птиц называют кочующими. У них Дреал зимой смещается несколько к югу (снегири, чечетки, свиристели, вороны). Так, снегири и свиристели, гнездящиеся преимущественно в северных областях европейской части России, зимой в массе кочуют по ее центральным районам. Наконец, среди наших птиц немало видов, которые всю жизнь проводят в районе, где они вывелись. Таких **Птиц** называют оседлыми. Следовательно, в данном конкретном районе можно различить несколько групп птиц: постоянно живущие (**истинно** — воробьи, можно — вороны); гнездящиеся (оловьи); зимующие (чечетки) и пролетные.

Способность к полету позволяет птицам мигрировать на тысячи километров. Многие птицы, гнездящиеся в районах с умеренным и холодным климатом, осенью совершают далевые перелеты в страны, где зимы не столь суровы и они обеспечены кормом; весной они возвращаются на места своих гнездований. Среди мигрирующих птиц есть такие, что перемещаются постепенно, и такие, что совершают дальние беспосадочные перелеты. Последние перед миграцией накапливают большие запасы жира.

Пути пролета обычно повторяют исторический путь расселения видов. Так, например, зеленая пеночка, гнездящаяся в Московской области, летит через всю Сибирь и по Дальнему Востоку в Индокитай и Австралию. Этим путем она когда-то расселилась. Некоторые птицы, особенно водоплавающие, летят по определенным путям, преимущественно по долинам рек, но большинство летит широким фронтом. Одни птицы совершают перелеты большими стаями, другие — малыми группами, третьи — поодиночке. Самцы весной обычно прилетают раньше самок. Быстрота перелетов очень разная, особенно весной. Грач задень пе|>сдвигается на 55 км, аист — на 60 км. Древесная славка, не останавливаясь за 50 ч полета, покрывает расстояние около 3 тыс. км, летя от Масачусетса до Пуэрто-Рико. При весенних перелетах птицы летят к местам гнездования гораздо быстрее, чем осенью к местам зимовок. Например кулик-веретенник, зимующий в Новой Зеландии и на о. Тасмания, гуляемает путь в 12 тыс. км осенью за 2—3 мес, а весной — за 1 мес.

Сроки перелетов связаны прежде всего с состоянием кормовой базы. Весной раньше всех появляются всеядные птицы (грачи, скворцы, пн ем зерноядные и другие растительноядные и позднее всего чисто насекомоядные (стрижи, ласточки), а осенью — наоборот, первыми улетают насекомоядные.

Как птицы находят путь к местам своих зимовок и обратно на места гнездования, еще полностью не выяснено. Обычно, улетев за тысячи ки-

лометров на зимовку, птицы возвращаются на следующий год на место своего вывода, многие летят ночью, многие — через море, у многих молодые улетают раньше или позднее, чем старые, по неизвестному им пути. Здесь имеют место чувство направления во время миграции и чувство дома — хоминг. При дальних перелетах птицы используют инстинкты, связанные с явлением земного магнетизма и с ориентацией по Солнцу и звездам; на ближних — визуальные и запаховые признаки знакомой местности.

В жизни птиц большую роль играет фотопериодизм. У большинства птиц наступление фаз годового репродуктивного цикла зависит от сезонных изменений длины светового дня, но у видов, обитающих вблизи экватора, имеют значение и другие факторы, такие, например, как изменение состава корма или доступность воды. Фотопериодизм корректирует и сроки полового созревания молодняка, колеблющиеся от 2 мес у домашнего перепела до 9 лет у странствующего альбатроса, а также сезонные явления в жизни птиц — линьки и миграции.

Коммуникация птиц построена главным образом на использовании зрения и слуха. Этой цели служат специфические позы и маневры, особая окраска оперения, пение, тревожные крики и т. п. У многих птиц самцы обладают способностью к пению, их пение влияет на развитие половых процессов у самок и служит предупреждением другим особям о том, что данный гнездовой участок уже занят. Для многих птиц характерен половой диморфизм, т. е. самец резко отличается от самки своим более ярким оперением, особенно в сезон размножения, иной формой пера, размерами и другими признаками.

Спариванию у птиц предшествуют различные брачные игры — токование глухарей, тетеревов, дроф, тяга вальдшнепов, блеяние бекасом или драки самцов из-за самок (особенно у полигамных видов). Но иногда роли меняются: так, у кулика краснозобика самки крупнее самцов, и именно самки устраивают настоящие побоища за самцов, которые и насаживают яйца. Токует тот пол, который находится в избытке. Интересные игры устраивают шалашники Австралии и Новой Гвинеи. Они сооружают сложные постройки — шалаши с двориками-площадками перед ними, украшают их цветами, камешками и другими блестящими и привлекающими внимание предметами и токуют здесь.

У отдельных видов птиц по численности могут преобладать самки (дрофи, павлины) или самцы (тинаму, яканы), но чаще соотношение полов примерно равное. Взаимоотношения полов могут складываться по типу моногамии, т. е. самка спаривается с одним самцом. При этом пары могут образовываться либо только на период размножения, либо на длительный срок, возможно на всю жизнь (неразлучники, гуси, лс беди, орлы). Для птиц в большой степени характерна моногамия, и не редко многолетняя.

При полигамии самец или самка спариваются с несколькими партнерами за период размножения. Если самец имеет гарем из нескольких самок (фазан) — это полигиния; если самка спаривается с несколькими самцами (цветной бекас) — это полиандрия.

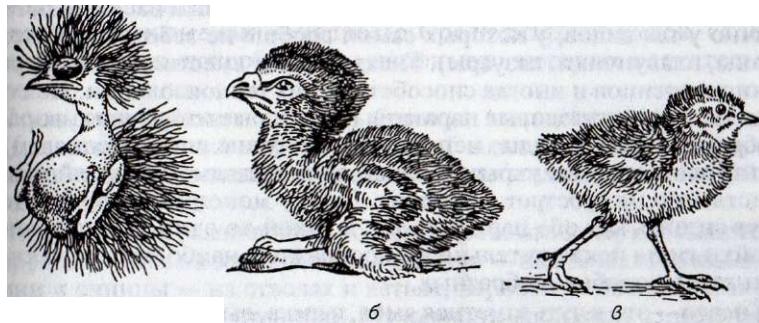


Рис. 257. Птенцы различных птиц одного возраста:
а — птенцовых (конек); б — полу выводковых (орел); в — выводковых (куропатка)

Размножаются птицы, откладывая яйца, которые обычно насиживают. Птенцы, которые появляются из яиц опушеными, зрячими и способными самостоятельно передвигаться и следовать за матерью, уводящей их от гнезда (куры, гуси), называются выводковыми (рис. 257). Птенцы, которые появляются на свет почти или совсем голыми, слепыми и остаются в отличие от выводковых длительное время в гнезде, где их выкармливают родители (воробы, канарейки), называются птенцовыми. Есть и промежуточный — полу выводковый тип (соколы, совы).

Птенцы выводкового типа часто способны с рождения питаться самостоятельно и в определенной степени способны к терморегуляции. Яйца выводковых птиц имеют гораздо более крупный желток, поскольку его остаток сохраняется в брюшной полости птенца как страховой резерв питания. Птенцов гнездового и полу выводкового типов кормят родители, засовывая корм в глотку (воробьинообразные) или предоставляя возможность самостоятельно склевывать корм из своего клюва (хищники), с земли (чайки) или с гнезда (цапли), куда отрыгивается корм; иногда птенцы достают корм из пищевода родителей (веслоногие). Голуби выкармливают птенцов особым питательным секретом, который выделяют стенки зоба (птичьим молочком). Но независимо от того, кормят птенцов родители или они питаются сами, их корм всегда более нежен и богат белком, чем пища взрослых птиц, что обеспечивает исключительно быстрый рост молодняка. Даже страус достигает окончательных размеров за год, а мелкие птицы — меньше **чем** за месяц. Сроки прекращения контактов птенцов с родителями различны. О птенцах сорных кур никто не заботится с момента вылупления; у большинства видов связь молодняка и взрослых исчезает в этот же гнездовой сезон, но у гусей и журавлей сохраняется до следующего сезона. Крупные птицы гнездятся раз в год и даже не каждый год. Многие мелкие воробьинообразные делают две и три кладки за сезон, и голуби, у которых в кладке лишь 2 яйца, — до пяти кладок.

Обычно птицы высиживают яйца в гнезде. Даже при совместном воспитании птенцов функция насиживания может быть закреплена за

каким-то одним полом, чаще за самкой. Лишь самцы насиживают яйца обычно у тех видов, у которых самки вообще не заботятся о птенцах (яканы, плавунчики, казуары). Гнезда служат вместилищем и укрытием яиц и птенцов и иногда способствуют их теплоизоляции. Не строят гнезд не только гнездовые паразиты (некоторые из кукушек, воробьинообразных), но и виды, использующие чужие гнезда (соколы), использующие готовые укрытия (совы) или откладывющие яйца непосредственно на субстрат (козодои). Даже у моногамных видов гнездо могут строить как оба партнера, так и какой-то один. Технология постройки гнезд исключительно разнообразна и наибольшей сложности достигает у воробьинообразных.

Гнездо — это и чуть заметная ямка, и ямка, выложенная травой и пухом, иногда это настил на дереве из ветвей, иногда чашеобразное оружение из веточек, травы и других материалов, иногда это дупла или норы. Некоторые птицы (ласточки) лепят гнезда из глины, смешанной со слюной, а стрижи саланганды делают съедобные гнезда из своей слюны — «ласточкины гнезда». Очень интересны гнезда ремизов, сделанные из растительных волокон и пуха тополя, или гнезда портних-славок, сшитые растительными волокнами из нескольких листьев. Иногда птицы строят огромные колониальные гнезда (ткачики).

Многим птицам свойствен колониальный образ жизни. Так, у нас на севере, на прибрежных скалах, тысячами гнездятся кайры, чистики, чайки и другие океанические птицы, образуя целые птичьи городки-базары. Колониально гнездятся и многие другие птицы, при этом колонии часто бывают смешанными, как на птичьих базарах, когда вместе гнездятся различные виды птиц.

По характеру питания птиц подразделяют на полифагов, или всеядных (майны, вороны), и стенофагов, питающихся однородными кормами (колибри, стрижи, клесты). Большинство птиц относятся к промежуточной группе. Птицы, которые питаются кормами животного происхождения, относятся к зоофагам. Среди них можно выделить энтомофагов (птиц, питающихся главным образом насекомыми), ихтиофагов (питающихся рыбой), миофагов (питающихся грызунами), орнитофагов (питающихся птицами), малакофагов (питающихся моллюсками), герпетофагов (питающихся пресмыкающимися), копрофагов (помет китов поедают некоторые чайки, качурки) и др. Птицы, которые питаются растительными кормами, относятся к фитофагам. Некоторые птицы относятся к зерноядным (в период выкармливания птенцов эти птицы часто становятся насекомоядными), плодоягодоядным (дубонос, попугай), другие питаются вегетативными частями растений (гуси, лебеди, лысухи) или низшими растениями (кряквы).

У растительноядных птиц мускульный желудок имеет мощные стенки и толстую кутикулу, в нем с помощью гастролитов происходит измельчение корма. Лишь попугаи способны измельчать корм клювом столь же полноценно, как млекопитающие с помощью зубов. Птицы, потребляющие малопитательный растительный корм (гуси, страусы, тетеревиные), обладают симбиотическими

щечными бактериями, расщепляющими клетчатку и вырабатывающими незаменимые аминокислоты. Среди птиц нет видов, специализированных к потреблению, например, мертвых растительных тканей.

Птицы, способные поглощать большое количество корма за один раз (дневные хищники и зерноядные), обладают большим зобом, служащим для депонирования корма, или имеют сильно растяжимый желудок (совы, цапли).

Птицы сильно различаются по месту добывания корма. Например, Скворец кормится на земле, дятлы и пищухи — на стволах деревьев, поползни и синицы — на стволах и ветвях, воробы — на земле и ветвях (когда выкармливают птенцов). Мухоловки, ласточки, стрижи ловят Летающих насекомых в воздухе, трясогузки — на земле, а горихвостки охотятся повсюду.

Хозяйственное значение птиц. Птицы имеют большое значение для хозяйственной деятельности человека. Большая часть их приносит несомненную пользу. Птицеводство принадлежит к числу важнейших отраслей животноводства нашей страны. Оно дает большое количество ценного мяса, яиц, пера, пуха и других видов продуктов и промышленного сырья. В наибольшем количестве разводят кур (рис. 258). Большое значение имеет разведение уток и гусей. Разводят также голубей, перепелов, индеек, цесарок, фазанов, страусов.

В вольерах, загонах, водоемах содержат крупные виды декоративных птиц: фазанов (золотой, алмазный), павлинов, уток (мандинка, каролинка), лебедей (шипун, черный) и др. В качестве декоративных Птиц содержат в клетках многие виды попугаев (волнистый, розеллы, неразлучники, какаду), ткачиков (амадины, астрильды, муний), мелкие виды голубей (смеющаяся, бриллиантовая горлицы), большинство из которых успешно размножается в неволе. Зоокультура комнатно-декоративных птиц насчитывает около 100 видов воробьинообразных из разных семейств, регулярно размножающихся в неволе. Среди них есть и полностью одомашненные виды (канарейка, японская амадина).

Технология содержания и разведения этих видов птиц имеет много общего с технологией содержания и разведения сельскохозяйственных видов гусеобразных, курообразных, голубеобразных. В настоящее время ежегодно в мире разводят на дичефермах и выпускают в охотничьи угодья более 70 млн фазанов, 4 млн серых куропаток, 2 млн кекликов и красных куропаток, 4,5 млн уток разных пород.

Многие дикие птицы служат важным объектом как спортивной, ТНК и промысловой охоты. Как объекты охоты представляют интерес не слишком мелкие многочисленные птицы, обладающие хорошими вкусовыми качествами и большим репродуктивным потенциалом. Наибольшее значение для охотничьего хозяйства имеют глухари, тетерева, рябчики, куропатки, утки, гуси, казарки, разнообразные кулики и др.

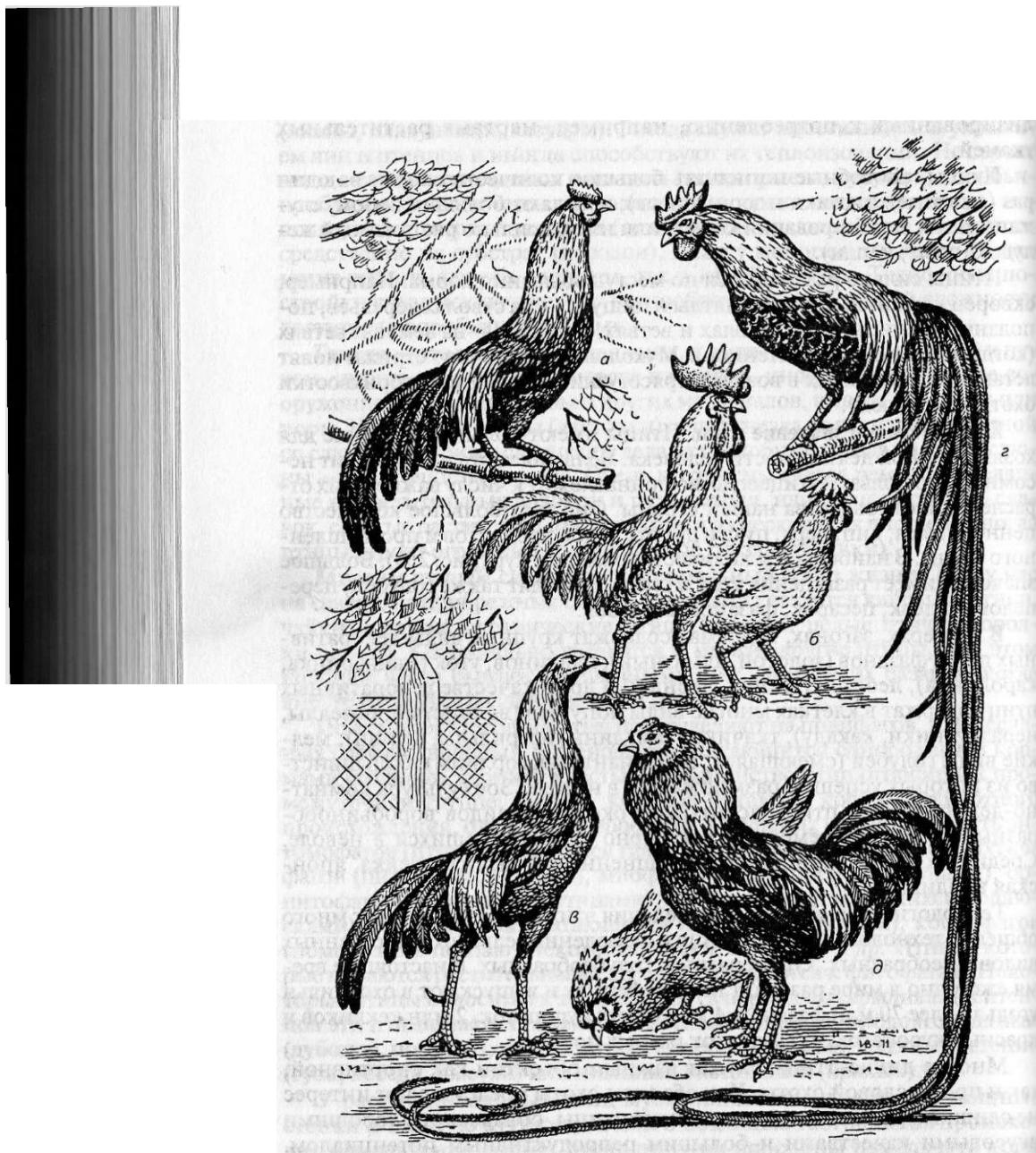


Рис. 258. Породы кур как пример образования новых форм животных путем селекции:
а — дикий банкивский петух; б — петух и курица породы леггорн; в — бойцовый петух;
г — петух длиннохвостый породы иокогама; д — петух и курица орловской породы

В прошлом дикие птицы служили также источником шкурок (гагары, поганки), перьев (райские птицы, страус), яиц (чистики, чайки), жира (птенцы трубконосых, гуахаро). Сохранился и поныне сбор пуха из гнезд обыкновенной гаги. Некоторые виды используются как ловчие животные — соколы, орлы. К птицам, используемым для связи, относятся почтовые породы домашних голубей.

В сельском хозяйстве используют и продукты жизнедеятельности птиц. Здесь важна роль колониальных рыбоядных, так называемых гуапроизводящих (помет — гуано является ценнейшим удобрением) морских птиц, осуществляющих перенос больших количеств азота из моря на сушу. Добыча гуано в России не ведется. Как производители гуано в мире имеют значение в первую очередь пеликаны, олуша и бакланы, гнездящиеся у побережья Южной Африки и Южной Америки.

Птицы приносят сельскому хозяйству, нашим садам, полям, лесам, иугам пользу, охраняя от насекомых-вредителей, моллюсков и грызунов. Известно свыше 5 тыс. видов насекомых, которые являются вредителями и с которыми необходимо бороться. Общие потери урожая от вредителей в среднем — около 20 % в год. Птицы способны заметно сокращать численность популяций вредителей, особенно в случае внезапной массовой вспышки численности какого-то вида.

Для примера, одна семья ласточки уничтожает до 1 млн насекомых, одна семья скворцов — 24 тыс. насекомых и их личинок, 1 синица лазоревка, которая питается главным образом яйцами насекомых, — 6,5 млн в год, 1 кукушка поедает в год 270 тыс. гусениц, сова, канюк — до 10—20 полевок и мышей в день. К этому следует добавить, что многие дневные птицы, совы, крупные чайки и ряд других истребляют большое количество сусликов, крыс, мышей, полевок, хомяков и т. д. Птицы питаются массу насекомых — паразитов сельскохозяйственных животных. Они уничтожают многих переносчиков заболеваний человека (мух, москитов, комаров и др.).

Даже те птицы, которые во взрослом состоянии питаются растительной пищей, выкармливают своих птенцов обычно насекомыми, червями, слизнями, чем приносят пользу растениеводству. Этому способствует и высокая численность птиц даже в нашей зоне. Так, в лиственных лесах и в парковых насаждениях с хорошо развитой кроной и богатым подлеском обитают от 1100 до 2386 семейных пар птиц на 1 км², в ельниках меньше — от 180 до 240, а в сухом сосняке — от 96 до 145. На полях с живой изгородью на 1 км обитают 112—168 пар, на таких же полях без изгороди — лишь 40—96, а на полях с перелесками и группами кустарника — до 400—442 пар.

Из сказанного очевидно, как важны охрана птиц и другие мероприятия по увеличению их численности. В нашей стране осуществляется ряд мер по привлечению птиц и увеличению их численности путем, в частности, размещения и расстановки в местах их гнездования пупянок, скворечен и других искусственных убежищ (рис. 259), их подкормки и т. п.

—259—
ХОЧУЩИЕ МОИ ПРОТОПОЧКАМЫ СВОИХ МАСТЕРСТВОВ
ПОДДЕРЖАТЬ, НАГРАДЯ ИХ СОВЕРШЕННОСТЬЮ. ВСЕДОБРЫЙ
ДЛЯ МОИХ СЫНОВЬЕВ КРУГЛЫЙ КРУГЛЫМОСТЬЮ. ВСЕДОБРЫЙ
ДЛЯ МОИХ СЫНОВЬЕВ КРУГЛЫЙ КРУГЛЫМОСТЬЮ.

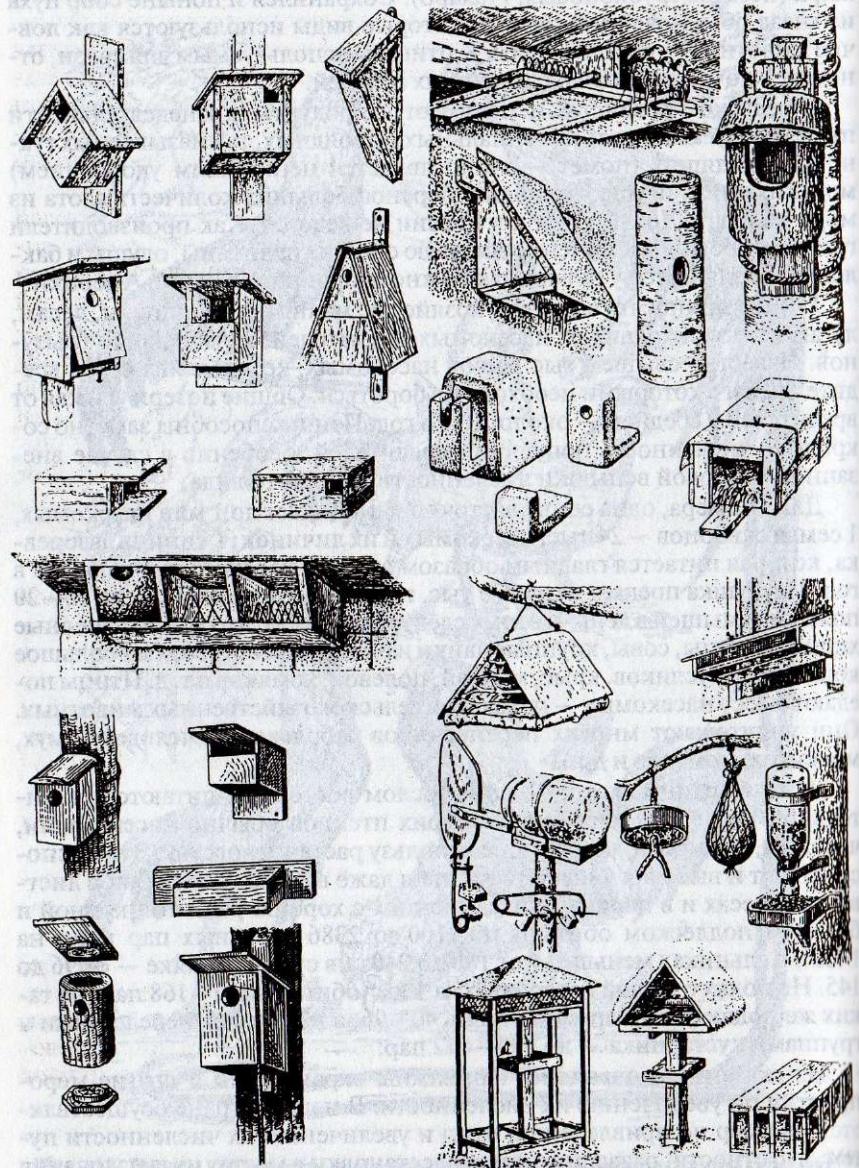


Рис. 259. Различные типы домиков для гнезд и кормушек

Но некоторые птицы могут приносить значительный ущерб хозяйственной деятельности человека. Стайные птицы (чайки, скворцы) **нередко** служат причиной аварий летательных аппаратов. Гнездование нрановых на линиях электропередачи приводит к коротким замыканиям. В рыбоводных хозяйствах наносят ущерб чайки, поедающие **мальков** и переносящие инвазионные заболевания рыб. Высокая подвижность птиц делает их эффективными переносчиками инфекций, в **том числе** таких, которыми болеют не только птицы. Это заболевания **человека** и сельскохозяйственных животных, в частности орнитоз, **чума, in гаммы** гриппозных и других вирусных инфекций.

Многие виды растений зависят от птиц, как от распространителей семян, а некоторые — как от опылителей. Но отдельные (стайные) виды птиц в силу своей мобильности могут наносить значительный ущерб полям, садам и виноградникам. Обыкновенный и розовый скворцы способны уничтожить до 1/3 урожая винограда и косточковых. В Тунисе такой ущерб оценивается ежегодно в 8—10 млн немецких марок. В Средней Азии в отдельные годы скворцы уничтожают до **50 %** черных и 25 % белых сортов винограда. В Африке трупиаловые и гкачиковые могут уничтожить до 70 % и более урожая зерновых, становясь причиной голода для местного населения. С такими птицами приходится вести борьбу

СИСТЕМАТИЧЕСКИЙ ОБЗОР ПТИЦ

Все современные птицы относятся к подклассу Веерохвостые, или Настоящие птицы. У современных птиц хвостовой отдел резко укорочен, а последние позвонки, срастаясь, образуют пигостиль, к которому веером крепятся рулевые перья. В фауне России насчитывается около 730 видов птиц, а это только 8,5 % мировой фауны. Всего же на земном шаре обитает около 9 тыс. видов. Эти тысячи видов птиц распределяются между тремя надотрядами, которые включают большое число отрядов, в современной фауне — около 30 (рис. 260 и 261).

НАДОТРЯД пингвины (**IMPENNES**). Эта немногочисленная группа своеобразных птиц (около 15 видов) утратила способность к полету (рис. 260, 6). Эти птицы давно перестали летать, но зато научились замечательно плавать и нырять. Зарегистрирован случай, когда императорский пингвин погружался на глубину 265 м.

Тело у пингвинов вальковатое, вытянутое. Оперение плотное, стержни перьев уплощенные, опахала развиты слабо. Аптерий нет. **11** и нька скатая по срокам, один раз в год. Крылья превратились в ласты, помогающие птице плавать, хорошо развит киль. Ноги сдвинуты далеко назад, поэтому на суше они держат тело вертикально. Пальцы ног соединены перепонкой.

Пингвины обитают в Южном полушарии. Живут колониями, моно-**гены**. Большую часть времени проводят в море, где питаются, но гнездится на суше, устраивая гнезда в виде ямки или норы. В кладке **пдпо-двайца**. Питаются рыбой и другими морскими животными.

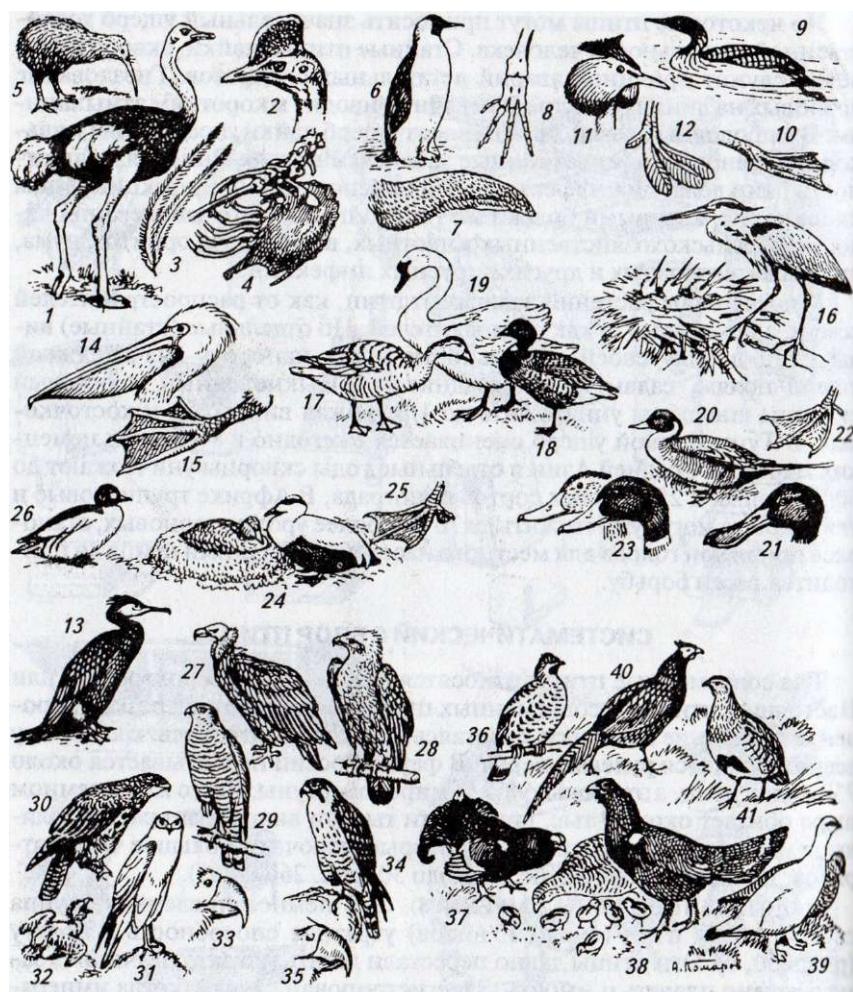


Рис. 260. Представители разных отрядов птиц:

1—африканский страус; 2—казуар; 3—двойное перо эму; 4—плечевой пояс и грудина страуса; 5—киви; 6—императорский пингвин; 7—крыло пингвина; 8—скелет ноги пингвина (короткой цевкой); 9—гагара; 10—лапа гагары; 11—большая поганка (чомга); 12—лапа поганки; 13—баклан; 14—пеликан; 15—лапа пеликана; 16—серая цапля рядом с гнездом; 17—серый гусь; 18—черная казарка; 19—лебедь-шипун; 20—шилохвость; 21—широконоска; 22—лапа настоящей утки (задний палец без лопасти); 23—кряква (впереди селезень, сзади — уйка); 24—самец и самка обыкновенной гаги (самка на гнезде); 25—лапа нырковой утки (задний палец с кожистой лопастью); 26—гоголь; 27—белоголовый сип; 28—степной орел; 29—ястреб тетеревятник; 30—канюк; 31—степной лунь; 32—птенец дневного хищника; 33—клив луня; 34—настоящий сокол, или сапсан; 35—клив сокола с предвершинным зоцом; 36—рябчик; 37—тетерев; 38—глухарь и глухарка с птенцами; 39—белая куропатка (впереди в зимнем оперении, сзади — в летнем); 40—фазан (самец); 41—серая куропатка

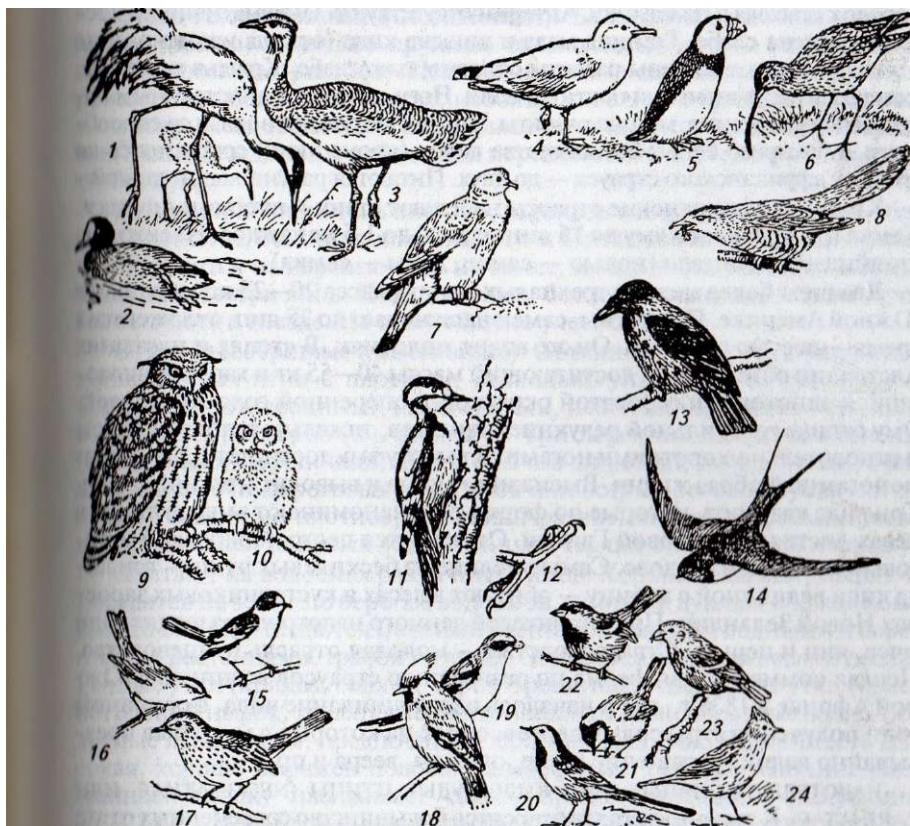


Рис. 261. Представители разных отрядов птиц:

I - журавль (взрослый и птенец); 2 — лысуха; 3 — дрофа; 4 — чайка серебристая; 5 — то-
порик; 6 — кроншнеп; 7 — голубь витютень; 8 — кукушка; 9 — филин; 10 — птенец совы;
II — большой пестрый дятел; 12 — лапа дятла; 13 — серая ворона; 14 — ласточка дер-
венская (касатка); 15 — мухоловка-пеструшка; 16 — славка; 17 — дрозд-рябинник; 18 —
Ииристель; 19 — сорокопут; 20 — белая трясогузка; 21 — большая синица; 22 — домовый
«оробей»; 23 — клест; 24 — жаворонок

Помыслового значения не имеют. Наиболее крупный императорский пингвин достигает в высоту более 1 м; близок к нему королевский пингвин; наименее многочисленный — пингвин Адели; один из самых мелких видов — галапагосский пингвин, длина тела которого около 50 см.

НАДОТРЯД БЕСКИЛЕВЫЕ, ИЛИ СТРАУСОВЫЕ ПТИЦЫ (RATITAE). Этот надотряд объединяет 10 видов птиц, утративших способность к полету, что вышло из изменения ряда органов (см. рис. 260). Представители этого надотряда — наиболее крупные из ныне живущих птиц. Самый крупный вид — пфриканский страус — высотой более 2,5 м и массой до 100 кг. Эти птицы стали наземными ходячими животными. Перья рассучены, без сцепления

бородок опахала в пластинку. Аптерии отсутствуют. Пневматичность костей выражена слабо. Грудина мала и лишена киля (отсюда наименование отряда). Грудные мышцы развиты относительно слабо. Крылья короткие, недоразвитые, у киви они почти исчезли. Ноги, ставшие единственным органом передвижения, мощно развиты. Число пальцев ног в связи со способностью быстрого бега у большинства видов (кроме киви) сократилось до трех, а у африканского страуса — до двух. Питаются растительной пищей.

Крупные африканские страусы населяют степи и пустыни Африки. Самка откладывает в яму до 15 яиц массой до 1,5 кг каждое. Насиживают яйца оба родителя (ночью — самец, днем — самка).

Два вида более мелких трехпальых нанд (масса 20—25 кг) обитают в Южной Америке. Полигамы, самец насиживает до 35 яиц, отложенных тремя—шестью самками. Он же водит молодняк. В степях и пустынях Австралии обитает эму, достигающий массы 40—55 кг и характеризующийся однотонной сероватой окраской и оперенной головой и шеей. Эму отличается сильной редукцией крыльев, тяжелым телосложением и относительно короткими ногами. В высоту эму достигает 1,8 м. Ведет моногамный образ жизни. Высиживает яйца и выводит птенцов самец. Три вида казуаров, которые по форме тела напоминают эму, обитают в лесах Австралии и Новой Гвинеи. Отличаются пестрой окраской и роговым шлемом на голове. Самые мелкие из бескилевых птиц — три вида киви величиной с курицу — обитают в лесах и кустарниковых зарослях Новой Зеландии. Представителей данного надотряда разводят ради мяса, яиц и перьев. Страусоводство — молодая отрасль птицеводства. Первая коммерческая ферма по разведению страусов возникла в Южной Африке в 1838 г., где и началось одомашнивание вида, в основном ради получения страусовых перьев, спрос на которые в мире был чрезвычайно велик (украшение шляп, одежды, веера и пр.).

НАДОТРЯД ТИПИЧНЫЕ, ИЛИ КИЛЕГРУДЫЕ ПТИЦЫ (NEOGNATHAE, ИЛИ CARINATAE). К этому надотряду относится большинство современных птиц, в том числе все птицы фауны России. Подавляющее большинство их может летать, поэтому крылья хорошо развиты. Двигающие их грудные мышцы мощно развиты и прикрепляются к килю грудины, откуда и название этих птиц. Плечевые кости полые. Контурные перья имеют хорошо выраженную пластинку опахала, бородки которого склеены друг с другом.

Килегрудые птицы распространены по всем материкам и островам земного шара. Надотряд включает около 30 отрядов. Ниже дается описание только тех из них, которые распространены в Российской Федерации и имеют большое практическое значение.

Отряд Пеликанообразные (Pelecaniformes). К этому отряду относятся крупные водоплавающие птицы (олушки, пеликаны, бакланы). Оперение густое, аптерии узкие. По земле ходят неуклюже, все четыре пальца коротких ног соединены плавательной перепонкой (см. рис. 260).

Обитают по берегам морей, на озерах и реках. Хорошо плавают, не которые прекрасно ныряют. Питаются в основном рыбой. Бакланы ловят ее, ныряя в глубь водоема, а пеликаны — выхватывая на мелководье. Моногамы, гнездятся колониями.

Отряд Листообразные (Ciconiiformes). Большинство из них (цапли, аисты, ибисы) довольно крупные околоводные птицы на длинных, тонких ногах (см. рис. 260). Цевка и нижняя часть голени лишены оперения. Шея длинная и тонкая, небольшая голова с прямым, удлиненным острым клювом. Ноги с четырьмя длинными пальцами: три передних пальца соединены небольшой перепонкой. Оперение рыхлое, почти без пуха.

Распространены очень широко, характерные места обитания — берега водоемов, болота, пойменные луга. Питаются мелкими животными. Гнездятся на деревьях или на земле, поодиночке или колониями. В кладке два-три яйца. Птенцы голые, беспомощные, длительное время остаются в гнезде. На зиму обычно улетают на юг.

Отряд Гулеобразные (Anseriformes). Объединяет различные виды водоплавающих птиц с плотным, несколько уплощенным туловищем, короткими, отнесенными назад ногами, небольшим хвостом, крыльями средних размеров (см. рис. 260). Три передних пальца соединены плавательной перепонкой. Клюв обычно широкий, уплощенный, с рядами и поперечных роговых пластинок или острых роговых зубчиков по краям. Оперение плотное, с хорошо развитым пухом. Большинство представителей во время летней линьки теряют способность к полету

Обитают на водоемах различных типов. Хорошо плавают, ныряют. Гнездятся на земле по берегам водоемов, в норах и дуплах. В кладке бывает до 20 яиц. Питаются мелкими беспозвоночными, водными и береговыми растениями, травой на лугах. Относятся к выводковым птицам. Речные утки (кряква, свиязь, чирок-трескунок), нырковые утки (красно-южный нырок, хохлатая чернеть) населяют разнообразные водно-болотные ландшафты, предпочитают пресные водоемы. Численность высокая, хорошо выражен половой диморфизм в окраске. Питание смешанное. Кладку насиживают самка, продолжительность инкубации 26—28 сут. Перелетные, но способны образовывать оседлые популяции. Гуси (пескулька, белолобый гусь, гуменник) — крупные птицы без полового диморфизма в окраске. Пары постоянны, насиживание около 30 сут. Дикие гуси (серый и сухонос) и утка-кряква считаются исходными формами, от которых были выведены разнообразные породы домашних гусей и уток. Гулеобразные — важные объекты спортивной и промысловой охоты. Мясо с прекрасными вкусовыми качествами. Для охотничьего хозяйства наибольшее значение имеет кряква. Высоко ценится нежный и теплый пух, выбираемый из гнезд морских нырковых уток — гаг. Все виды хорошо разводятся в искусственных условиях. Как объект дичеразведения имеет также ценность представитель близкого к гусям рода — канадская казарка, широко распространенная в Северной Америке. Разводят лебедей, мандаринок и др.

Отряд Соколообразные, или Дневные хищные птицы (Falconiformes). К ним относятся орлы, ястребы, луны, коршуны, соколы, орланы, грифы. Для всех них характерны острые, изогнутые когги и сильный, загнувшийся на конце клюв (см. рис. 260). Основание надклювья покрыто тонкой голой кожей — восковицей. Оперение упругое, плотное, но почти без пуха.

Места обитания хищных птиц разнообразны. Пищей им служат преимущественно различные позвоночные животные, которых ловят на земле, в воздухе и в воде, хотя мелкие соколиные (кобчики, пустельги) поедают также многих насекомых. Грифы, сипы и стервятники поедают падаль. Обычно многолетние моногамы. Гнезда устраивают на скалах, деревьях, иногда на земле, занимают чужие или искусственные гнезда. Насиживает обычно самка. Птицы полу выводкового типа. Перелетный вид — обычновенная пустельга — обитае на большей части территории России, по всей Евразии и Африке (кроме тундры, пустынь и тропических лесов).

Некоторые виды используются какловые птицы. Из представителей нашей орнитофауны это ястребы (перепелятник, тетеревятник), обитающие повсеместно. Добычу настигают быстрым коротким броском из засады или предварительно подлетают к жертве незаметно. Самки значительно крупнее самцов. Орел беркут питается наземными позвоночными, не брезгует падалью. Масса 3—6 кг. Распространен на большей части Евразии и Северной Америки, но всюду редок. Соколы (сапсан, балобан, кречет, дербник) способны к скоростному длительному преследованию жертвы на открытых пространствах. Сами гнезд не строят. Многие виды размножаются в неволе, возможно получение межвидовых гибридов.

Отряд Курообразные (Galliformes). Наземные и наземно-древесные птицы средней величины (см. рис. 260), обитают на всей территории нашей страны. Они отличаются плотным телосложением, небольшой головой с коротким клювом, довольно короткими, широкими крыльями и сильными средней длины четырехпальми ногами с большими тупыми когтями. Оперение довольно плотное, но почти без пуха. Полет этих птиц сильный, но тяжелый, с частыми взмахами крыльев, без парения. Большинство ведет оседлый образ жизни, есть кочующие (белая куропатка) и перелетные (перепел). В основном полигамы, хорошо выражен половой диморфизм. Гнезда устраивают на земле в виде неглу бокой ямки. В кладке от 6 до 22 яиц. Птицы выводкового типа. Взрослые преимущественно растительноядные (имеется объемистый зоб), но в питании птенцов преобладают насекомые.

В России обитает 20 видов куриных птиц, которые относятся к двум семействам — Фазановые и Тетеревинные.

Фазановые (Phasianidae) отличаются тем, что цевка у них не оперена и по бокам пальцев нет роговой бахромы. У самцов многих видов на ногах имеются шпоры. В России наибольшее значение для охоты имеют обыкновенный фазан, серая куропатка и обыкновенный перепел, к этому семейству относятся также каменные куропатки, а из разводимых домашних птиц — куры, перепела, цесарки. Домашние куры произошли от обитающих в Индии банкивских кур (*Gallus gallus*) (см. рис. 258). Куры являются основным объектом птицеводства нашей страны: они дают большое количество ценного мяса и яиц. Обыкновенная цесарка (*Numida meleagris*), являющаяся предком домашней, обитает на севере Африки.

Тетеревинные (Tetraonidae) характеризуются оперенной цевкой и наличием по бокам пальцев рядов роговых зубчиков. Шпор у самцов нет,

К этому семейству относятся глухари, тетерева, куропатки (белая, тундряная), рябчики. Все эти птицы являются важными объектами спортивной и промысловой охоты. Обитают в зоне лесов умеренного пояса, в лесотундре и тундре. Основной зимний корм — концевые побеги, почки, сережки и хвоя. Осеню важное место в рационе занимают ягоды.

Многие виды являются объектами зоокультуры. В больших количествах разводят различные виды фазанов (главный объект дичеразведения — обыкновенный фазан), широкомасштабное разведение куропаток сдерживает их моногамия. В Северной Америке обитает обыкновенная индейка (*Meleagris gallopava*) — предок домашней индейки, которая является объектом птицеводства.

Отряд Журавлеобразные (Gruiformes) представлен в России различными видами журавлей, пастушков и дроф (см. рис. 261). Это наземные и околоводные птицы, очень различные по величине. Питание смешанное. Гнезда устраивают на земле; птицы выводкового типа. Перелетные.

Широко распространен серый журавль (*Gruis grus*), достигающий в высоту 120 см. По берегам водоемов, заросших растительностью, обитает лысуха (*Fulica atra*), имеющая промысловое значение. Наиболее Крупный представитель — дрофа, или дудак (*Otis tarda*), массой до 16 кг. Крылья короткие, округлые. Ноги крепкие трехпалые; цевка покрыта шестигранными роговыми щитками. Это типично степные птицы. Большинство журавлеобразных — редкие виды. Предпринимаются меры по их разведению в неволе.

Отряд Ржанкообразные (Charadriiformes). По современной систематике этот отряд делится на три подотряда.

Подотряд Чайки (Lari). К этому подотряду принадлежат различные виды чаек, крачек, поморников. Птицы различной величины, с вытянутым телом, короткими ногами, длинными острыми крыльями и хвостом средней длины (см. рис. 261). Ноги четырехпалые, три передних пальца связаны плавательной перепонкой. Клюв большой, прямой, часто с крючком на конце. Оперение плотное с развитым пухом.

Живут по берегам морей, на озерах и реках. Большую часть дня лентяют над водой, выхватывая из нее корм. Крупные чайки, поморники нападают на мелких зверьков и птиц. В степных районах приносят пользу истреблением сурчиков и полевок. Хорошо плавают, но не ныряют. Некоторые оседлы, но большинство видов — кочующие или пелетные птицы. Гнездятся на земле или на скалах вблизи водоемов. В кладке обычно 3 яйца. Полувыводковые птицы.

Подотряд Чистики (Alcae). К этому подотряду относятся кайры, гагарки, чистики, туники и др.

Морские птицы средних размеров с вальковатым вытянутым телом, короткими, но острыми крыльями, небольшим хвостом (см. рис. 261). Ноги отодвинуты далеко назад, поэтому сидящая птица держится почти вертикально. На ногах только три пальца, соединенные плавательной перепонкой. Спина темная, брюшко обычно белое. Оперение очень плотное.

Населяют берега морей Северного Ледовитого и Тихого океаном. Живут большими колониями, составляя основную часть птичьих базаров. Гнездятся чаще всего на карнизах скал, откладывая 1—3 яйца прямо на камни; некоторые живут в норах. Питаются главным образом рыбой, которую ловят, ныряя в воду.

Подотряд Кулики (Charadrii) — мелкие и средней величины птицы (ржанки, шилоклювки, плавунчики, тиркушки, авдотки), большинство из которых имеет удлиненную шею, острые, но не очень длинные и крылья, короткий хвост, длинные ноги с голой цевкой, а то и голенюю (см. рис. 261). Клюв обычно длинный, тонкий. Плавательных перепонок на ногах, как правило, нет, или они слабо развиты. Оперение ровное, почти без пуха.

Обитатели болот и лугов, сырых лесов и перелесков, берегов рек, озер и морей; реже они встречаются в степи и пустыне. На зиму улетают на юг. Гнездятся обычно на земле. Самка откладывает в небольшую яму 4 яйца. Выводковые птицы. Питаются различной пищей, преимущественно мелкими беспозвоночными. Наибольшее практическое значение имеют бекасовые. Многие виды этого семейства относятся к объектам охоты, поскольку их мясо обладает прекрасным вкусом. В России наибольшее значение имеют, в частности, такие виды, как травник, филин, большой улит, обыкновенный перевозчик — кулики средних размеров (100—200 г). Моногамы, гнездятся отдельными парами. Песочник, турухтан, краснозобик, чернозобик и кулик-воробей мельче (40—100 г) и имеют более короткие ноги и клюв, чем представители улитов. Турухтан полигам, самцы образуют токовища. Бекас, дупель, гаршинеп и вальдинеп имеют массу 75—300 г (вальдинеп самый крупный), относительно короткие ноги, длинный клюв и более широкие крылья (особенно вальдинеп). Активны в сумерках и ночью. Большой и средний кроншнеп, большой и малый веретенники — длинноклювые и длинноногие птицы средних и крупных размеров (300—700 г). Кроншнепы гнездятся отдельными парами, веретенники — также и колониями. Все бекасовые перелетные.

Отряд Голубеобразные (Columbiformes). Небольшие и средней величины птицы. Голова с коротким сильным клювом; основание надклювья покрыто утолщенной кожей (восковицей), на которой открывают-ся ноздри (см. рис. 261). Крылья длинные, острые. Ноги короткие, с четырьмя пальцами, один из которых направлен назад. Хвост среднем длины. Оперение плотное, но пух развит слабо. Голуби прекрасно летают, но пищу собирают на земле. Растительноядные, в небольшом количестве поедают насекомых. Нуждаются в регулярном водопое. **Стайные**, моногамы, гнездятся парами. Клинтух (*Columba oenas*) гнездится на дуплах, вяхирь (*C. palumbus*) и горлица — на деревьях. В кладке два яйца, за год обычно делают несколько кладок, насиживание около 17 сут. Насиживают оба пола, но самка в большей степени. Тип развития птенцовий, первые дни птенцы питаются выделениями стенок зоба родителя. Перелетные, сизый голубь оседлый или кочующий. Сизый голубь (*C. livia*) — обитатель открытых пространств, гнездящийся колониями на скалах и постройках человека; остальные виды лесные; кольчатая

Ифлица (*Streptopelia decaocto*) — синантроп. Легко размножаются в немнис. Мясо высокого качества. Являются второстепенными объектами нипы из-за невысокой численности. В городах и селах широко распространены полуводомашние сизари.

Отряд Кукушкообразные (Cuculiformes). Широко распространенные и иссно-кустарниковые птицы (см. рис. 261). Обитающая у нас обыкновенная кукушка (*Cuculus canorus*) является гнездовым паразитом. И не строит; яйца откладывает в гнезда почти 125 видов птиц. Кукушки поедают большое количество мохнатых гусениц, которых избелим им **ИГ** другие птицы.

Отряд **Совообразные (Strigiformes).** Широко распространены, наследственны.... самые разнообразные ландшафты (белая сова, ушастая сова, нешикарьи и, сычи). Каки соколообразные, совы имеют острые загнутые когти и клюв. Это конвергентное сходство. Вместе с тем они значительно отличаются от дневных хищников рядом признаков, связанных с ночным образом жизни (см. рис. 261). Зоба нет. Оперение рыхлое, очень мягкое, полет бесшумный. Голова округлая, уплощенная спереди, равномерно (льное расположение перьев образует плоский лицевой диск. Огромные глаза направлены вперед, что увеличивает поле бинокулярного зрения. Хорошо развит слух. Шея столь подвижная, что позволяет птице поворачивать голову почти на 270°. Наружный палец может направляться как вперед, так и назад. Гнездятся в дуплах, на земле, в норах. Птенцы, которые виды охотно гнездятся в постройках и искусственных гнездовых ящиках. Моногамы, насиживают самку после откладки первого яйца. Птенцы нылупляются слепыми, но опущенными. Ночью, а некоторые виды даже на рассвете и закате охотятся на различных мелких позвоночных, и первую очередь на грызунов, истребляя их в большом количестве. Одна сова неядет за год уничтожает около 1 тыс. мышей и полевок. Отличительные виды сов можно легко привлечь на поля для охоты, устанавливают шесты — присады. В России такие шесты привлекают ушастую сову, в Западной Европе — сипуху. Основу питания самой крупной совы — филина (*Bubo bubo*) — могут составлять зайцы и птицы.

Отряд Дятлообразные (Piciformes). Ведущие древесный образ жизни, исчезнувшие птицы, приспособившиеся кланью по стволам деревьев (см. рис. 261). Лапы у них короткие, с острыми когтями. Обычно два пальца ног направлены вперед, а два — назад. Перья хвоста очень жесткие, заостренные на концах, что позволяет птице опираться «постом» о ствол дерева. Клюв острый, долотообразный, приспособленный к долблению древесины. Язык очень длинный, тонкий и острый, что позволяет извлекать насекомых из-под коры и ствола. Кормятся в коронах и на стволах деревьев, реже на земле. Моногамы, гнезда устраивают в дуплах, которые выдалбливают сами, или используют естественные. Птицы птенцовного типа.

В России наиболее обычен большой пестрый дятел (*Dendrocopos major*), а также средний и малый пестрье дятлы, несколько реже встречаются черный, или желтый, зеленый и трехпалый.

Отряд Воробьинообразные (Passeriformes). Наиболее многочисленный отряд, включающий более половины всех видов птиц малой и средней величины (см. рис. 261). Туловище стройное. Ноги тонкие, с четырьмя пальцами, расположенными на одном уровне, из которых три направлены вперед, а один назад. Когти острые. Клюв разной формы. Образ жизни разнообразен. Большинство — древесно-кустарниковые, дневные птицы. Есть растительноядные (почти все зерноядные виды выкармливают птенцов насекомыми), животноядные (многие питаются насекомыми) и всеядные виды. Обычно моногамы, строя гнезда. Все виды птенцовые. У многих более одного выводка в год. Наиболее распространены у нас представители следующих семейств — Ласточковые, Трясогузковые, Сорокопутовые, Свиристелевые, Завирушковые, Дроздовые, Славковые, Корольковые, Мухоловковые, Синицеевые, Поползневые, Овсянковые, Вьюрковые, Скворцовые, Врановые.

Песни самцов некоторых видов являются неотъемлемой частью звукового фона природных ландшафтов; таких птиц часто содержат в неволе ради пения. Наилучшими певцами считаются восточный соловей и славка черноголовая. Многие виды приносят пользу, уничтожая вредителей сельского и лесного хозяйства. Так, скворцы (обыкновенный, розовый), поедают те виды насекомых, которые в данный момент являются многочисленными и легкодоступными. При массовом размножении, например, саранчи они слетаются в большом количестве со значительных расстояний, эффективно подавляя вспышки численности этого вредителя. Семья розовых скворцов за лето может уничтожить свыше 100 кг саранчи. Самая маленькая птичка наших лесов королек — поедает огромное количество лесных насекомых и их яиц.

В наших парках, садах, полях, огородах обычны трясогузки, гори хвостки, синицы, жаворонки, чеканы, славки, пеночки, дрозды, мухоловки, иволги. Многие виды хорошо адаптируются к антропогенному ландшафту и охотно гнездятся в различных сооружениях и искусстве 11 ных гнездах — синицы (большая, лазоревка), белая трясогузка, муходловка-пеструшка, горихвостки (садовая, домовая) и др.

Происхождение птиц. Несомненно, что птицы произошли от одной из древних групп пресмыкающихся, а именно от ящеротазовых рептилий — архозавров, ведущих родословную от текодонтов. Об этом свидетельствует наличие у них ряда общих черт строения, в частности с динозаврами. Непосредственные предки птиц пока еще не установлены. Первые представители класса птиц появились в мезозойской эре. В юрских отложениях были обнаружены останки животного — археоптерикса (рис. 262), совмещающего признаки пресмыкающихся и птиц. Внешний вид его напоминал птицу, но имевшийся длинный хвост состоял из большого числа позвонков (около 20), по бокам попарно расположались крупные перья. Роговой клюв отсутствовал, челюсти несли зубы. На крыльях сохранились три длинных свободно двигающиеся пальца скоггями. На грудине киля не было. Вероятно, археоптериксы жили на деревьях, планируя и перепархивая с ветки на ветку, но не летая. Археоптерикс, скорее всего, представляет собой боковую ветвь в эволюции птиц.

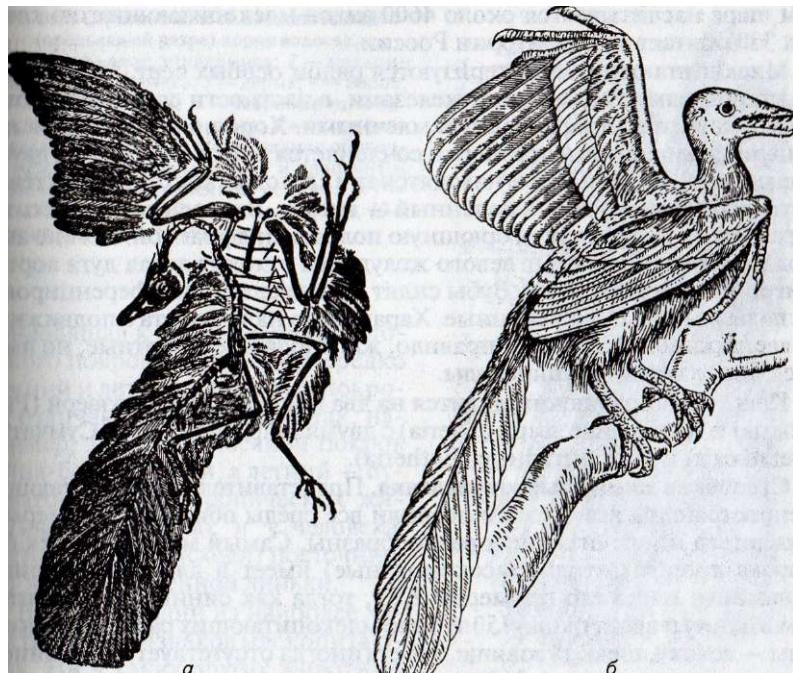


Рис. 262. Археоптерикс:
а — отпечаток на сланце; б — реконструкция

В конце мезозойской эры появились настоящие птицы. Возможно, их предком былprotoavis (*Protoavis*), найденный в триасовых отложениях. Из отложений мелового периода известны иктиорнисы (*Ichthyornis*) и гесперорнисы (*Hesperornis*). У них еще сохранялись мелкие зубы, головной мозг был очень мал. Беззубые птицы с роговым клювом возникли, по-видимому, в начале кайнозойской эры.

КЛАСС МЛЕКОПИТАЮЩИЕ (*Mammalia*)

Млекопитающие — наиболее высокоорганизованные позвоночные * и нотные из группы Amniota. Об этом свидетельствуют следующие особенности их морфологии и физиологии: 1) исключительно высокое развитие центральной нервной системы и органов чувств; 2) совершенная система терморегуляции, позволяющая поддерживать температуру организма на удивительно постоянном уровне 37—39 °C; 3) приспособления к живорождению и выкармливанию детенышей молоком, что создает благоприятные условия для выживания потомства.

Высокая организация млекопитающих и сложная высшая нервная деятельность обеспечили широкое их распространение. Ныне на земле

ном шаре насчитывается около 4600 видов млекопитающих, из которых 320 обитает на территории России.

Млекопитающие характеризуются рядом особых черт. Их тело покрыто волосами, кожа богата железами, в частности сальными, потовыми, а также специфическими млечными. Хорошо развита кора полушарий головного мозга. Череп сочленяется с позвоночником двумя мышелками. В среднем ухе находятся три слуховые косточки. Локтевой сустав направлен назад, а коленный — вперед. Внутренняя полость тела разделена на грудную и брюшную полости диафрагмой. Легкие альвеолярного строения. От левого желудочка отходит левая дуга аорты, эритроциты безъядерные. Зубы сидят в альвеолах и дифференцированы на резцы, клыки и коренные. Характерно наличие рта с подвижными мясистыми губами. Как правило, живородящие животные, но имеются также яйцекладущие виды.

Класс млекопитающих делится на два подкласса: Первозвани (Prototheria) и Настоящие звери (Theria) с двумя инфраклассами: Сумчатые (Metatheria) и Плацентарные (Eutheria).

Строение и жизненные отправления. Представители млекопитающих распространены всесветно и освоили все среды обитания. Размеры и форма тела млекопитающих разнообразны. Самый мелкий зверек белозубка-крошка (отряд Насекомоядные) имеет в длину всего лишь около 2 см, масса его примерно 2,5 г, тогда как синий кит достигает 33 м в длину и весит около 150 т. Тело млекопитающих разделено на отделы — голову, шею, туловище, хвост (иногда отсутствует), передние и задние конечности (рис. 263).

Для покровов млекопитающих особенно характерно развитие волос, образующих у подавляющего большинства видов волосяной покров. Последний предохраняет тело от потери тепла, уменьшает отдачу власти, смягчает механические воздействия, обуславливает ту или иную окраску зверя. Волосяной покров — важнейший орган терморегуляции млекопитающих, особенно сильно он развит у северных животных, у которых ежегодно сменяется. Осенью зверь меняет редкий и короткий, а потому обладающий хорошей теплопроводностью летний волосяной покров на густой и более длинный зимний, обладающий высокими теплозащитными свойствами. Весной происходит весенняя линька,

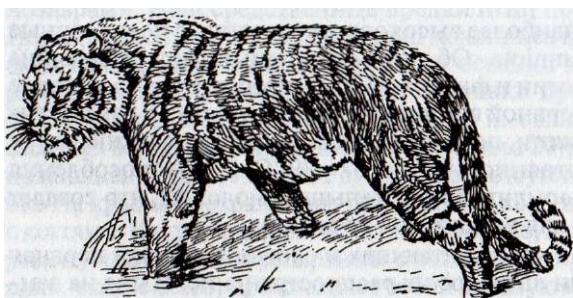


Рис. 263. Внешний вид млекопитающего (ши-ря)

Рис. 264. Строение волоса млекопитающего (продольный разрез корня волоса):

1 — роговой слой эпидермиса; 2 — мальпигиан слой эпидермиса; 3 — дерма; 4 — волос; 5 — сальная железа; 6 — жировые отложения; 7 — сердцевина волоса; 8 — корковое вещество волоса; 9 — стекловидная оболочка между влагалищем полосы и волосяным мешочком; 10 — волосяной мешочек; 11 — луковище волоса; 12 — сосочек полосы; 13 — мышца волосяного мешочка

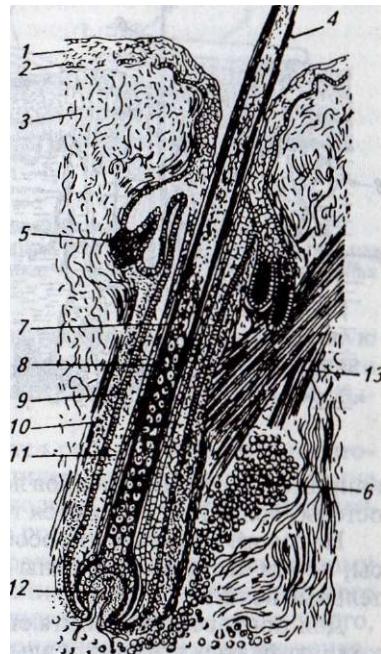
приводящая к смене зимнего волосяного покрова на летний. Нередко зимний и летний волосяные покровы отличаются и окраской: так, например, зимний волосяной покров шиза-беляка белый, а летний — буровато-серый. У тропических животных, так же как и у человека, смена волос происходит постоянно, (с) ясно выраженной линьки.

Волосы — один из наиболее характерных признаков млекопитающих. Ни у каких других животных нет таких образований; среди зверей видов, лишенных волос, очень мало. У некоторых из них, например у китов и дельфинов, волосы на теле исчезли в связи с водным образом жизни. У других — тропических животных — волосы становятся редкими в силу обитания в жарком климате (слоны, носороги). Но во всех случаях исчезновение волос явление вторичное, и у эмбрионов китов они закладываются в том или ином количестве.

Число волос на теле млекопитающих огромно, например у белки оно превышает 14 млн, у тонкорунной овцы их насчитывается до 100 млн, у песца на 1 см² кожи спины растет до 20 тыс. волос. Волосы представляют собой нитевидные роговые образования, развивающиеся из луковиц, которые образуются из клеток эпидермиса (рис. 264).

Каждый волос состоит из стержня и корня: первый вырастает над кожей, второй погружен в нее. Стержень волоса состоит из трех слоев. И стержне нет живых клеток, и потому он не может расти и изменять свою форму. Растет же волос за счет размножения клеток нижней расширенной части корня — луковицы.

Волосы, образующие волосяной покров млекопитающих, обычно неодинаковы: различают более длинные, грубые и редкие направляющие, не имеющие извитой части; более короткие, часто сидящие и имеющие извитую часть волосы называются остевыми, они придают характерную окраску животному, и наконец, дающие оттенок или, как говорят, «коду», более тонкие и нежные — пуховые. Теплозащитные свойства меха



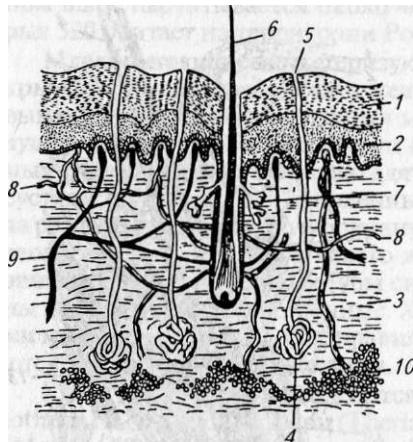


Рис. 265. Строение кожи млекопитающего:
1 — роговой слой эпидермиса; 2 — мальвийский слой эпидермиса; 3 — дерми, 4 — потовая железа; 5 — устье потовой же лезы; 6 — волос; 7 — сальная железа; 8 — кровеносный сосуд; 9 — нервы кожи, 10 — подкожные жировые отложения

например, волосяной покров летом состоит только из остьевых; у крота остьевых волос нет, а имеются только пуховые.

На голове расположены особо длинные и упругие волосы — вибриссы, корни которых окружены нервными окончаниями. Это чувствительные волосы.

Для покровов млекопитающих также очень характерно наличие и коже трубчатых потовых и альвеолярных (грозевидных) сальных желез (рис. 265). Первые выделяют пот (по составу близкий к первично!! моче) и охлаждают тело при его перегреве, вторые — сальный секрет, который смазывает поверхность кожи и волосы, предохраняя их от различных воздействий среды (например, намокания) и придавая им эластичность. У многих млекопитающих имеются еще пахучие железы видоизмененные потовые или сальные. Поскольку обоняние у зверей хорошо развито, эти железы играют большую роль в межвидовых и внутривидовых взаимоотношениях.

Но особенно характерно для млекопитающих наличие млечных (молочных) желез, выделяющих у самок высокопитательное молоко — пищу новорожденных детенышей. Наличие млечных желез и выкармливание детенышей молоком — важнейшие систематические признаки млекопитающих, от которых они и получили свое название. Млечные железы представляют собой видоизмененные потовые железы более сложного строения. Протоки млечных желез открываются на вершине или в углублении соска молочной железы. Число сосков молочных желез у млекопитающих связано с их плодовитостью и колеблется от одной до десяти и даже более пар. Обычно они расположены и два ряда на брюшной стороне тела (на груди, в паху), реже по бокам тела

Конечные фаланги пальцев почти у всех млекопитающих несут роговые образования — ногти, когти, копыта, которые являются проекциями эпидермиса (рис. 266). Ногти представляют собой плоскую и относительно тонкую роговую пластинку, расположенную на концах

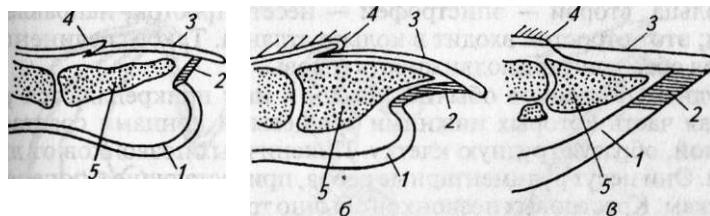


Рис. 266. Продольный разрез конечной фаланги пальцев:
 f — обезьяны (ноготь); б — собаки (коготь); в — лошади (копыто); 1 — подушечка пальца;
 1* — подошвенная пластинка; 3 — когтевая пластинка; 4 — коггевой валик; 5 — конечная
 фаланга пальца

фаланг пальцев. Когти имеют толстую выпуклую и изогнутую роговую Пластинку, которая выдается вперед острым выступом. Подушечки Пальцев большие и мягкие. У копыт толстая роговая пластинка охватывает конец фаланги пальца.

У многих млекопитающих развиваются рога, в образовании которых участвуют наружные покровы. Эпидермальное происхождение рогов у млекопитающих неоднозначно. Из других производных кожи следует отметить **ИллюиН** (на хвосте у крыс), иглы, щетину, роговой клюв и др.

Скелет взрослых млекопитающих образован костными элементами (рис. 267). Хорда развита только на ранних стадиях эмбрионального развития. Позвоночник состоит из шейного, грудного, поясничного, фасеткового и хвостового отделов. Позвоночник образован позвонками с плоскими сочленяющими поверхностями, между которыми расположены хрящевые прослойки — мениски. Шейных позвонков всегда (кроме ленивцев и дюгоней) семь. Длина шеи зависит не от числа позвонков, а от их размеров. Первый шейный позвонок — атлант — имеет

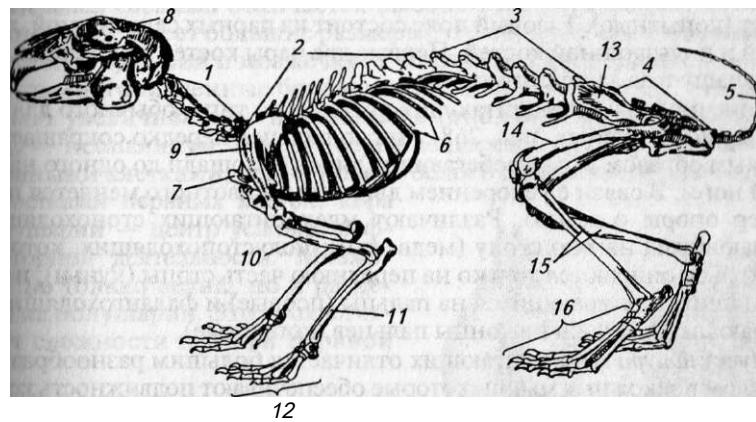


Рис. 267. Скелет млекопитающего (кролика):
 1 — череп; 2 — позвоночник; 3 — грудная кость; 4 — лопатка; 5 — предплечье; 6 — ребра;
 7 — грудная кость; 8 — голова; 9 — лопатка; 10 — плечевая кость; 11 — кости предплечья;
 12 — кости кисти; 13 — таз; 14 — бедренная кость; 15 — кости голени; 16 — кости стопы

вид кольца, второй эпистрофей — несет отросток, направленный вперед; этот отросток входит в кольцо атланта. Такое соединение позвонков обеспечивает подвижность головы.

Грудных позвонков обычно 12—15, к ним прикрепляются ребра, большая часть которых нижними хрящевыми концами срастается с грудиной, образуя грудную клетку. Поясничных позвонков от двух до девяти. Они несутrudиментарные ребра, прирастающие к поперечным отросткам. Крестцовых позвонков обычно три-четыре; они срастаются в единую крестцовую кость, давая прочную основу для прикрепления костей таза. Число хвостовых позвонков различно.

Череп млекопитающих отличается большим объемом мозговой юробки. Череп сочленяется с позвоночником двумя мышцами. Нижняя челюсть образована одной зубной костью, которая прикреплена к височной области черепа. В среднем ухе три сочлененные косточки (молоточек, наковальня, стремя), соединяющие барабанную перепонку с овальным окном. Сочлененная кость превращается в молоточек, квадратная кость — в наковальню, столбчатая — в стремя. Характерно развитие вторичного костного нёба. Зубы млекопитающих гетеродор! пные и сидят в альвеолах.

Конечности млекопитающих располагаются под туловищем, а иг выдаются в стороны, каку рептилий; этим обеспечивается значительная устойчивость тела и грузоподъемность конечностей.

Плечевой пояс млекопитающих, как правило, состоит из двух пар костей — лопаток и ключиц, а не из трех, как у птиц и рептилий. Лопатка хорошо развита. Коракоид еще в процессе эмбрионального развития прирастает к лопатке, образуя коракоидный отросток. Линия, у однопроходных коракоид сохраняет самостоятельность. Ключица развита у млекопитающих, обладающих высокой подвижностью передней конечностей (приматы, рукокрылые, кошачьи), у остальных она отсутствует (копытные). Тазовый пояс состоит из парных седалищной, лобковой и подвздошной костей. Первые две пары костей срастаются, о разуя чашу таза закрытого типа.

Конечности у млекопитающих пятитипового типа, обычного для наземных позвоночных (рис. 268). Число пальцев нередко сокращается, главным образом у быстрых бегающих видов (у лошади до одного на каждую ногу). В связи с ускорением движения животного меняется характер опоры о землю. Различают млекопитающих стопоходящих, опирающихся на всю стопу (медведьи); полустопоходящих, которых при ходьбе опираются только на переднюю часть стопы (куньи); шагающих — опирающихся на пальцы (псовидные) и фалангоходящих — опирающихся только на концы пальцев (копытные).

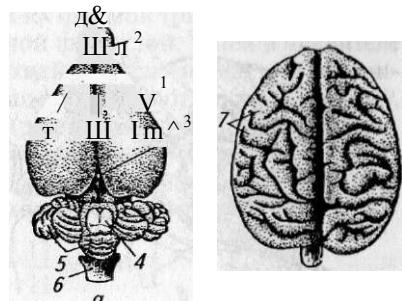
Мускулатура млекопитающих отличается большим разнообразием и числом подкожных мышц, которые обеспечивают подвижность кожи, и в частности мимику лица у приматов, изменяют положение волоссянки и покрова. Для млекопитающих характерно наличие куполообразной мышечной перегородки в полости тела — диафрагмы, которая делит полость на грудную и брюшную и, в частности, значительно повышает

Рис. 268. Разнообразие строения конечностей млекопитающих:
в — летучая мышь; **б** — крот; **в** — кенгуру; **г** — лошадь; **г**) — орангутан; **е** — дельфин

»м|м|>ективность вентиляции легких. В грудной полости находятся сердце и легкие, а в брюшной — желудок, кишечник, печень, почки и ряд других органов. Мышцы млекопитающих содержат миоглобин.

Нервная система отличается высокой степенью сложности. Голоиной мозг имеет большие размеры, что обусловлено сильным раздим гием полушарий и мозжечка. У многих млекопитающих кора образует многочисленные борозды и извилины (складки); это значительно увеличивает ее поверхность (рис. 269). Развит вторичный Вод — неопаллиум. В мозге серое вещество — кора, образованная Нервными клетками, лежит поверх белого вещества, образованного тростками нервных клеток; кора Полушарий — центр условно-рефлексной деятельности. Мозжечок не только велик, но и разделен на полушария. Это соответствует сложности высшей нервной

Рис. 269. Мозг млекопитающих:
0 — кролика; **б** — шимпанзе; **1** — полушария
Мсроднего мозга; **2** — обонятельные доли;
1 — шифиз; **4** — средний мозг; **5** — мозжечок;
6 — продолговатый мозг; **7** — извилины
Ейушарий



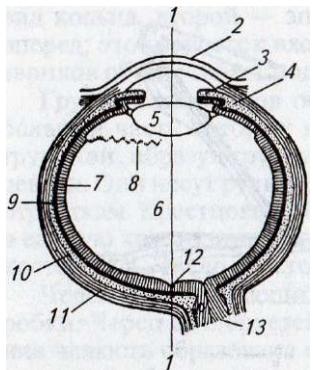


Рис. 270. Схема строения глаза млекопитающего:

1 — зрительная ось; 2 — роговица—радужка; 4 — ресничный мускул; 5 — хрусталик; 6 — стекловидное тело; 7 — сетчатка; 8 — волнистая складка; 9 — пигментная оболочка; 10 — сосудистая оболочка; 11 — склеры; 12 — желтое пятно; 13 — зрительный нерв

деятельности и разнообразию движениях млекопитающих. Черепных нервов 12 пар

Органы чувств млекопитающих развиты хорошо, в том числе зрение, обоняние и слух. Глаза менее крупные, чем у птиц, подвижные, роговица выпуклая, хрусталик линзообразный (рис. 270). У многих бинокулярное зрение. Обоняние развито очень хорошо. Оно играет в жизни млекопитающих огромную роль; у большинства хорошее чутье позволяет находить пищу, опознавать особей своего вида, избегать врагов. Поэтому у зверей, как правило, имеются большие обонятельные полости, поверхность которых значительно увеличена наличием складок — обонятельных раковин. Наружное ухо состоит из наружного слухового хода и зачастую подвижной ушной раковины. В полости среднего уха находится три слуховые косточки, соединенные между собой; с барабанной перепонкой связан молоточек, далее следуют наковальня и стремя, которое упирается в перепонку овального окна внутреннего уха (рис. 271).

Органы пищеварения. Пищеварительная система млекопитающие начинается ротовой полостью (рис. 272). Здесь на верхней и нижней челюстях расположены двумя дугами зубы. У подавляющего большинства млекопитающих, кроме самых примитивных, зубы обычно дифференцированы на резцы, имеющие долотообразную форму, острые конические клыки и коренные (предкоренные и собственно коренные). Коренные зубы у хищников обычно уплощены с боков, с острыми, режущими краями, а у растительноядных форм они имеют уплощенную верхнюю поверхность со складками эмали, что облегчает перетирание жесткой пищи. Строение зубной системы — важный систематический признак.

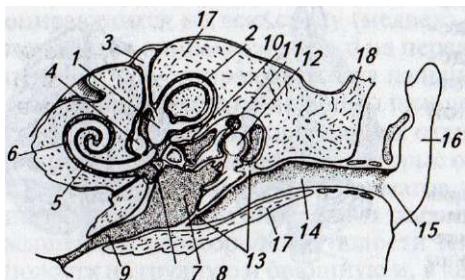


Рис. 271. Схема строения органа слуха и равновесия млекопитающего:

1 — овальный мешочек; 2 — полукруглый канал; 3 — эндодолимфатический проток; 4 — круглый мешочек; 5 — ушной канала; 6 — перилимфатическая полость; 7 — круглое окно; 8 — полость средней уши; 9 — евстахиева труба; 10 — стремя II; 11 — наковальня; 12 — молоточек; 13 — барабанная перепонка; 14 — наружный слуховой ход; 15 — слуховое отверстие; 16 — ушная раковина; 17 — каменистая кость; 18 — чешуйчатая кость

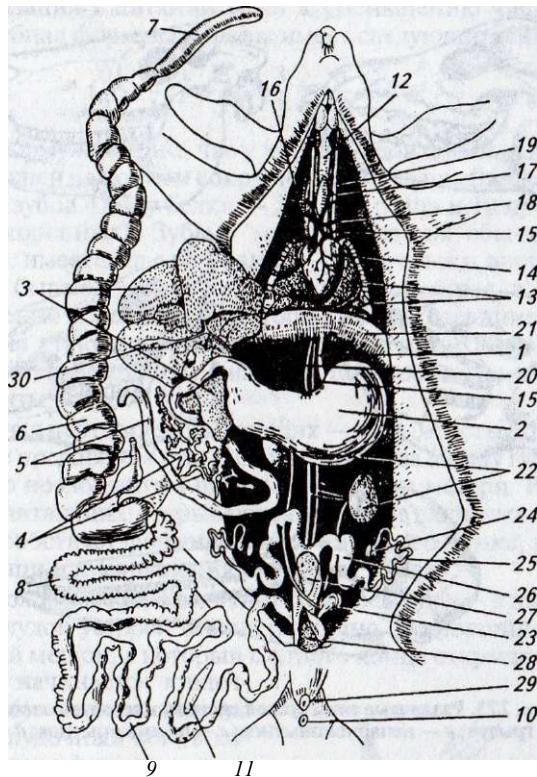


Рис. 272. Внутреннее строение млекопитающего (кролика):
I — пищевод; 2 — желудок; 3 — печень; 4 — поджелудочная железа; 5 — тонкая кишка; 6 — (Лиши) кишка; 7 — червеобразный отросток; 8 — толстая кишка; 9 — прямая кишка; 10 — «чилиное» отверстие; 11 — селезенка; 12 — трахея; 13 — легкие; 14 — сердце; 15 — аорта; /ft 16 — подключичная артерия; 17 — наружная яремная вена; 20 — задняя полая вена; 21 — «чи-»рагма; 22 — почки; 23 — мочевой пузырь; 24 — яичник; 25 — яйцеводы; 26 — матка; 27 — «ИГалище»; 28 — мочеполовой синус; 29 — мочевое отверстие; 30 — желчный пузырь

Признак (рис. 273). Разные группы зверей характеризуются определенным числом зубов разных категорий и их формой (рис. 274). Для описания зубной системы пользуются зубной формулой. В ней в числителе указывается число зубов в половине верхней, а в знаменателе — в половине нижней челюсти по категориям. Резцы обозначают буквой *i* (incisivi), **и паки** — *c* (canini), предкоренные — *pm* (praemolares) и собственно коренные — *m* (molares). Наибольшее число зубов (кроме зубатых китов) у пла-**и п**арных млекопитающих, главным образом всеядных — 44 (у свиней, **к**ротов и немногих других). Зубная формула этих животных такова:

$$\frac{i\underset{3}{l}}{3} \ c-\{ \underset{4}{pm} \underset{3}{m} = \underset{11}{x} \underset{2}{=} 44.$$

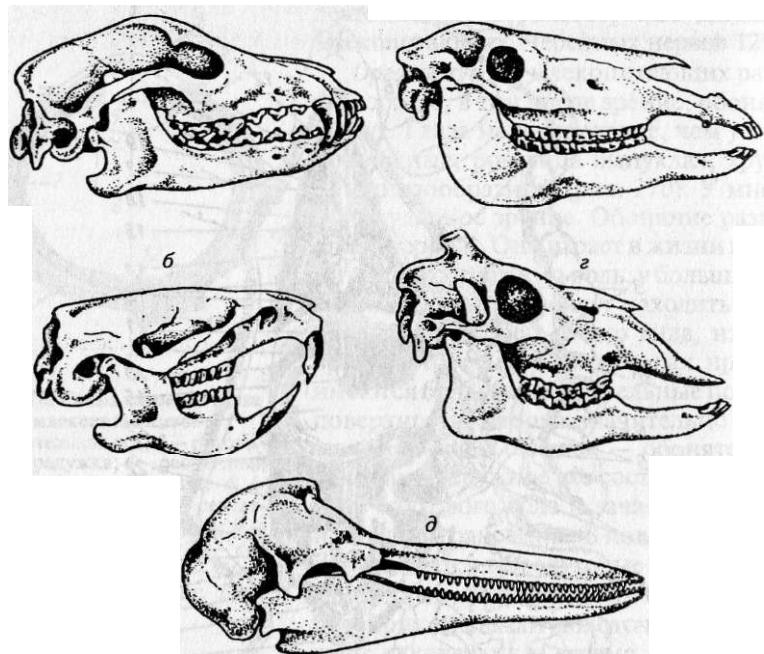


Рис. 273. Различные типы зубной системы млекопитающих:
а — хищник; б — грызун; в — непарнокопытное; г — парнокопытное; д — дельфин

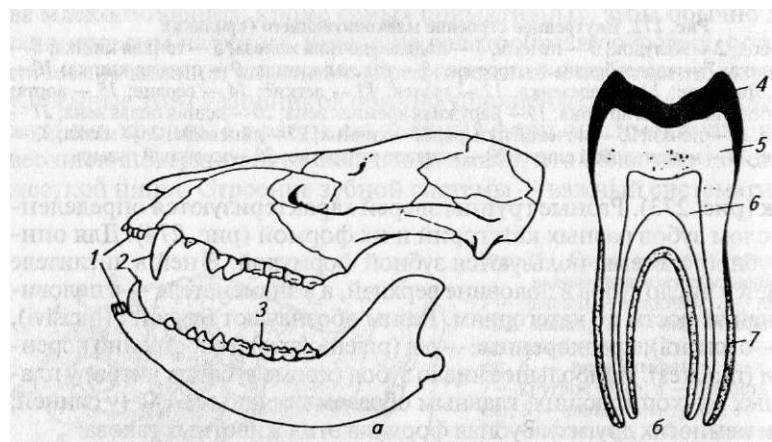


Рис. 274. Схема зубной системы млекопитающих (а) и схема строения зуба млекопитающих 1 — резцы; 2 — клыки; 3 — коренные зубы; 4 — эмаль; 5 — дентин; 6 — пульпа; 7 — цемой

Специализация в питании ведет к уменьшению числа зубов. Так, например, зубная формула коровы имеет следующий вид:

$$\frac{0}{3}, \frac{0}{3}, \frac{3}{3} = \frac{6}{10} \times 2 = 32.$$

Из этой формулы видно, что у коровы произошли редукция резцов **И** и клыков верхней челюсти и сокращение числа предкоренных. У волка **общее** число зубов 42, а у белки — 24 (по 2 резца и 10 предкоренных и Собственно коренных). Зубы у млекопитающих обычно сменяются **один** раз, т. е. имеется две генерации — молочные и постоянные зубы. Молочными бывают резцы, клыки и предкоренные, а постоянными •лце и собственно коренные. Некоторые млекопитающие в силу специфики питания утратили зубы, как, например, питающиеся мелкими насекомыми муравьеды или фильтрующие планктон из воды усатые (беззубые) киты.

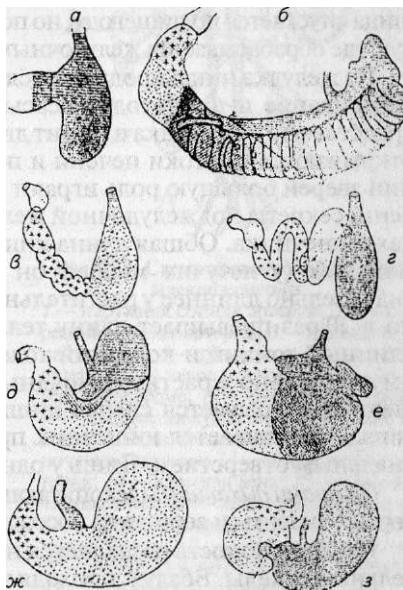
Характерная черта млекопитающих — это мясистые подвижные губы и наличие преддверия рта, т. е. пространства между губами, щеками **И** зубами, что позволяет им питаться молоком матери. В ротовую полость млекопитающих открываются протоки слюнных желез, секрет которых способствует формированию пищевого комка, расщеплению углеводов пищи и превращению крахмала в сахар.

Ротовая полость переходит в глотку, а последняя — в пищевод и далее Желудок. Желудок устроен весьма различно. В простейшем случае это юкостенный мешок, в который с одного конца открывается пищевод, К от другого начинается кишеч-

ник (рис. 275). Стенки желудка выделяют желудочный сок, в котором содержатся ферменты, расщепляющие прежде всего белки **пищи**. Наиболее сложно устроен желудок у жвачных (оленей, коров, быков, коз), состоящий из четырех отделов: первый самый большой — **пепельник** — рубец — имеет гладкие пенки, второй — сетка, небольшой со складчатыми стенками, образующими многочисленные (**шайки**). Третий — книжка — имеет

рис. 275. Форма желудка различных млекопитающих:

— угкона; б — кенгуру; в — клювокрылый дельфина; д — лошади; е — свиньи; ж — зайца; з — хомяка; части, покрытые многослойным эпителием, застрихованы; части, содержащие железы дна желудка, покрыты точками; части, содержащие иогорические железы, обозначены кривыми



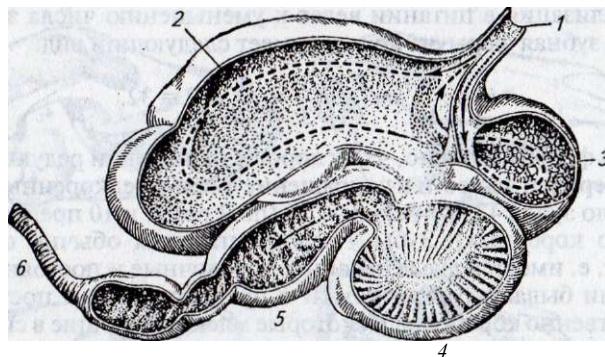


Рис. 276. Желудок коровы (штриховой линией показано движение пищи из рубца в сетку, затем обратно в рот; пунктирной линией показано движение жвачки по желобку в книжку и далее в сырчуг):
 / — пищевод; 2 — рубец; 3 — сетка; 4 — книжка; 5 — сырчуг; 6 — кишка

внутри многочисленные складки, и, наконец, последний — сырчуг — вырабатывает пищеварительный сок. Три первых отдела желудка жвачных являются видоизмененными частями пищевода, а собственно желудком следует считать только сырчуг (рис. 276). Пищевой ком, смоченный слюной, сначала попадает в рубец, где длительно задерживается. Здесь происходит его сбраживание за счет находящихся в рубце различных микроорганизмов. Затем пища поступает в сетку, а оттуда отрыгивается в рот, где вторично пережевывается и снова смачивается слюной. Получающаяся клещица спускается по пищеводу, но попадает уже не в рубец, а в книжку и сырчуг, где обрабатывается желудочным соком.

Из желудка пища продвигается в кишечник, где продолжается ее переваривание и происходит всасывание питательных веществ. Непосредственно от желудка отходит двенадцатиперстная кишка, в которую открываются протоки печени и поджелудочной железы. В пищеварении зверей большую роль играют желчь, образующаяся в большой печени, секреты поджелудочной железы и желез, расположенных в стенах кишечника. Общая длина кишечника меньше у хищных и насекомоядных (у летучих мышей он лишь в 2—3 раза длиннее тела) и значительно длиннее у растительноядных (у коровы длина его примерно в 20 раз превышает длину тела). Кишечник делится на отделы — длинный тонкий и короткий толстый. На границе этих отделов преимущественно у растительноядных и всеядных млекопитающих в кишечник открывается слепая кишка, исполняющая роль бродильного чана. Заканчивается кишечник прямой кишкой, открывающейся наружу анальным отверстием. Лишь у однопроходных имеется клоака.

Органами дыхания у млекопитающих служат легкие. Воздухоносные пути хорошо развиты и состоят из носовой полости, гортани, трахеи и бронхов.

Носовая полость разделяется на преддверие, дыхательный и обои и тельный отделы. Воздух, проходя через носовую полость, очищается от

Рис. 277. Строение легочных альвеол млекопитающего (справа — вскрытая альвеола с сетью капилляров)

МЫСЛИ, обеззараживается, согревается и увлажняется. Пройдя через хоаны в глотку, воздух попадает в гортань. Сложная система хрящей гортани препятствует попаданию пищи в **трахею** при глотании. Между хрящами расположены голосовые связки, участвующие в образовании звуков, издаваемых животными. Млекопитающие широко используют акустическую сигнализацию. Из гортани издух поступает в трахею. В стенке **Трахеи** заложены неполные хрящевые **кольца**, не позволяющие ей спадаться. В грудной полости трахея разделяется на два бронха, идущие к легким.

Легкие млекопитающих альвеолярного строения; бронхи, входя в **Тело** легких, распадаются на массу **нее** более мелких трубочек — бронхиол, которые заканчиваются полыми микроскопическими пузырьками — альвеолами (рис. 277). Число **их** достигает 300—500 млн, суммарная площадь — до 100 м². Вентиляция легких осуществляется **диоико**, как за счет поднятия и опускания ребер, так и за счет работы диафрагмы. Благодаря сложности Легких и участию диафрагмы в дыхании достигается высокая интенсивность газообмена, необходимого для поддержания постоянной температуры.

Кровеносная система млекопитающих характеризуется следующими основными признаками (рис. 278). **Имеется** большой и малый круги Кровообращения. Сердце четырехкамерное и состоит из двух предсердий и двух желудочков, артериальная и венозная кровь не смешивается. У птиц, от левого желудочка от-

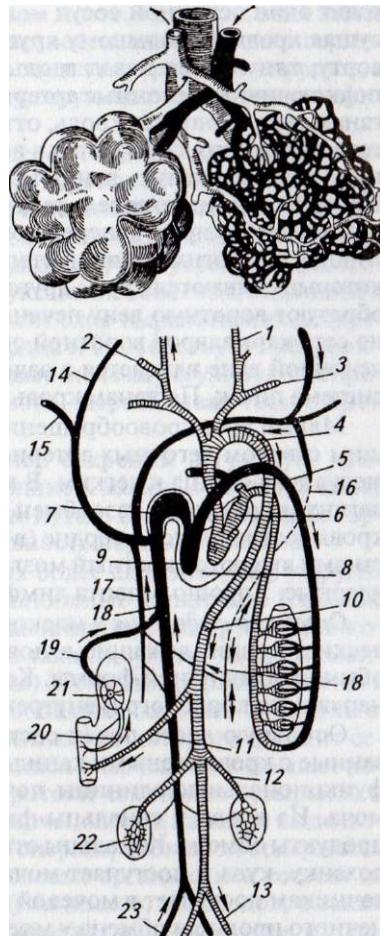


Рис. 278. Схема кровеносной системы млекопитающих:

1 — наружная сонная артерия; 2 — внутренняя сонная артерия; 3 — подключичная артерия; 4 — дуга аорты; 5 — легочная артерия; 6 — левое предсердие; 7 — правое предсердие; 8 — левый желудочек; 9 — правый желудочек; 10 — спинная аорта; 11 — внутренностная артерия; 12 — почечная артерия; 13 — подвздошная артерия; 14 — яремная вена; 15 — подключичная вена; 16 — левая непарная вена; 17 — правая непарная вена; 18 — задняя полая вена; 19 — печеночная вена; 20 — воротная вена печени; 21 — печень; 22 — почки; 23 — подвздошные вены

ходит один основной сосуд — левая дуга аорты (у птиц — правая), не сущая кровь по большому кругу. Дуга аорты продолжается в спинную аорту, тянувшуюся назад вдоль позвоночника. От дуги аорты отходя і подключичные и сонные артерии, от аорты — артерии к различным органам тела. Венозная кровь, оттекающая от заднего и среднего отделом тела животного, собирается в вены, которые, сливаясь, образуют мощную заднюю полую вену, тянувшуюся вперед вдоль позвоночника. Кровь от переднего отдела тела у большинства зверей по венам поступает в короткую правую переднюю полую вену, несущую кровь к сердцу. Лишь у немногих в равной степени развиты обе передние полые вены, которые сливаются друг с другом. Вены, несущие кровь от кишечника, образуют воротную вену печени, которая, войдя в печень, распадается на сеть капилляров воротной системы печени. От печени кровь по іє ченочной вене вливается в заднюю полую вену. Отсутствует воротная система почек. По венам кровь поступает в правое предсердие.

Малый круг кровообращения начинается от правого желудочка об щим стволом легочных артерий, несущих венозную кровь от правого желудочка сердца к легким. В капиллярах, обволакивающих альвеолы легких, происходит газообмен, и по легочным венам уже окисленная кровь возвращается в сердце (в левое предсердие). Кроветворными органами являются костный мозг и селезенка. Эритроциты мелкие, безъядерные. Хорошо развита лимфатическая система.

Органами выделения у млекопитающих служат тазовые (метанефрические) почки, лежащие в поясничной области. Это парные органы обычно бобовидной формы. Каждая из почек состоит из двух слоев — наружного коркового и внутреннего мозгового.

Основную часть почек составляют мочевые канальцы, тесно связанные с кровеносными капиллярами и формирующие структурные и функциональные единицы почек — нефроны, в которых образуется моча. Из крови в канальцы фильтруются излишки воды и конечные продукты обмена. Канальцы открываются в полость почек — почечную лоханку, куда и поступает моча. Из почечной лоханки моча по мочеточникам поступает в мочевой пузырь, а оттуда наружу. В качестве конечного продукта обмена у млекопитающих выводится мочевина. Это связано с внутриутробным развитием плода. Частично как органы выделения выступают потовые железы.

Железы внутренней секреции обеспечивают гуморальную регуляцию важнейших функций организма млекопитающих. Они вырабатываю! биологически активные вещества — гормоны, которые кровью разносятся по всему телу животного. Гормоны обладают специфичностью действия, каждый воздействует на определенный тип обменных процессов. Вместе с тем многие гормоны не обладают видовой специфичностью, оказывая воздействие и на представителей других видов. К же - лезам внутренней секреции относятся гипоталамус, гипофиз, щитовидная железа, надпочечники, поджелудочная железа, половые желе і и другие, двум последним свойственны как внутри-, так и внешнесекреторная деятельность. Особое значение имеет система гипотала

мус—гипофиз, расположенная в головном мозге. Удаление этих желез **ведет к гибели организма**. Гипофиз, в частности, вырабатывает гормоны, регулирующие деятельность других желез внутренней секреции, а гипоталамус — статины и либерины, регулирующие секрецию гипофизных гормонов.

Щитовидная железа расположена спереди трахеи, на гортани. Выделяемые ею гормоны (в первую очередь тироксин) участвуют в регулировании процессов обмена веществ, роста и развития организма. Надпочечники — парная железа, лежащая над верхним краем каждой почки. Состоят из двух слоев — наружного (коркового) и внутреннего (мозгового). Гормоны коркового слоя регулируют водно-солевой, белковый и углеводный обмен (кортизол). В мозговом слое вырабатывается адреналин, который стимулирует работу сердца и способствует превращению гликогена печени в глюкозу. Половые железы регулируют репродуктивные функции организма животного, вырабатывая тестостерон (семенники) и эстрогены (яичники).

Функционирование желез внутренней секреции регулирует центральная нервная система через подходящие к железам нервы, а также **Через гипофиз**, т. е. нервно-гуморальным путем. Вместе с тем сами гормоны оказывают воздействие на деятельность нервной системы.

Органы размножения млекопитающих более сложные, чем у других имиот. Все млекопитающие — раздельнополые животные. Половые **Органы** самцов слагаются из парных семенников, семяпроводов и копулятивного органа (полового члена), а также двух пар желез (семенных пузырьков и предстательной железы). Семенники расположены в щитней части брюшной полости или в особой складке кожи — мошонке. К семеннику прилегает придаток семенника, от которого отходит **семяпровод** (вольфов канал), открывающийся в мочеполовой канал копулятивного органа и далее наружу. Протоки семенных пузырьков и предстательной железы открываются в мочеполовой канал, и выделяемый ими секрет образует жидкую часть спермы, в которой содержатся вещества, активизирующие спермии. Половые органы самок состоят из парных (у однопроходных только левых) яичников, яйцеводов, а также **Матки** и влагалища. Яичники самок всегда лежат в полости тела, их размеры меньше, чем у других позвоночных. Яйцеводы открываются около **майчиков** воронками, выстланными мерцательным эпителием. По ним майклетка попадает в фалlopиевые трубы и далее в матку. Матка переходит в непарное влагалище, которое открывается наружу щелевидным отверстием. Строение матки у разных млекопитающих неодинаковое. **У одних** имеются две матки, открывающиеся во влагалище отдельными отверстиями — двойная матка (многие грызуны, кролики). У других тоже **имеется** две матки, но они открываются во влагалище одним общим отверстием; такая матка называется двураздельной (свиньи, некоторые грызуны, хищные). У ряда млекопитающих парные матки срастаются **примерно** на половину своей длины, а верхние части остаются свободными — двурогая матка (насекомоядные, копытные, китообразные). Наконец, у некоторых млекопитающих произошло полное слияние

Рис. 279. Схема строения яйцеводов и матки различных млекопитающих:
 а — однопроходные; б, в — сумчатые; г — плацентарные с двойной маткой; д — плацентарные с двурогой маткой; е — плацентарные с простой маткой; 7 — яйцевод; 2 — матки, 3 — влагалище; 4 — мочеполовое отверстие; 5 — мочевой пузырь; 6 — клоака; 7 — прямикна

обеих маток в простую матку, в которую открываются парные фаллопиевы трубы (некоторые рукокрылые, приматы) (рис. 279).

Зрелые яйцеклетки выпадают в полость тела, затем загоняются в яйцевод и, наконец, попадают в матку. Оплодотворение яйцеклетки происходит в верхней части яйцевода. Все млекопитающие (за исключением однопроходных) живородящие. У сумчатых плаценты не образуется, детеныши рождаются слаборазвитыми и дальнейшее развитие у большинства происходит в сумке, расположенной на животе матери. У остальных млекопитающих эмбрионы связаны со стенкой матки плацентой, через которую получают от материнского организма кислород и питательные вещества (рис. 280).

Продолжительность беременности варьирует от 2—4 нед (сумчатые, грызуны) до 2 лет (слоны). Некоторые мелкие млекопитающие могут приносить по два-три помета в год до 10 и более детенышей в каждом, крупные млекопитающие зачастую рожают один раз в 2—3 года по одному детенышу.

Экология млекопитающих. Млекопитающие населяют все материки и водоемы земного шара. Они освоили самые различ-

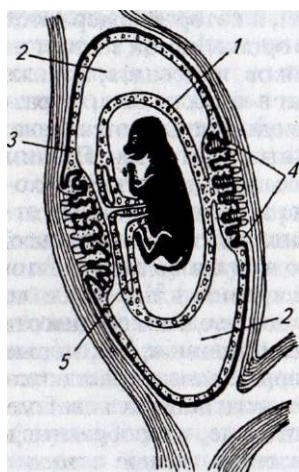


Рис. 280. Схема строения плаценты (разрез через маку):
 1 — амнион; 2 — аллантоис; 3 — хорион; 4 — кольца плаценты; 5 —rudимент желточного мешка

ные среды обитания. Поэтому среди млекопитающих можно выделить несколько жизненных форм, приспособленных к разным условиям.

У зверей, разных по своему происхождению и систематическому положению, но обитающих в одинаковых условиях среды, нередко развиваются сходные приспособительные признаки. Наличие общих признаков позволяет объединить различных по систематическому положению, но близких по образу жизни млекопитающих в несколько экологических типов (жизненных форм).

Наземные млекопитающие обитают на поверхности земли и бывают довольно крупными. Таковы различные копытные, многие хищные и др. Для этих животных особое значение имеет быстрота передвижения, что позволяет им спасаться бегством от врагов или догонять жертву. Это обычно стадные, сильные животные, пальцеходящие, нередко с уменьшенным числом пальцев, с образованием когтей, копыт. Ноги высокие, но с однообразным движением. Движение ног происходит преимущественно в плоскости, параллельной оси тела. Тело стройное, сплюснутое с боков. Нередко покровительственная окраска. Детеныши обычно рождаются хорошо развитыми.

Древесные млекопитающие приспособлены к лазанию по деревьям и кустарникам, к жизни на деревьях. Таковы большинство приматов, многие хищные (лесная куница, кошка и др.), грызуны (сони, белки). Обычно звери среднего и мелкого размера, с длинным стройным, округлым в сечении телом, недлинными, но очень подвижными в различных направлениях конечностями. Имеются специальные приспособления к лазанию — острые когти, цепкие пальцы, цепкий хвост и т. п. Детеныши рождаются обычно в дуплах и гнездах слепыми и малоподвижными.

Летающие млекопитающие приспособлены к полету (летучие мыши, крыланы). Основным органом полета у них служат крылья, образованные тонкой кожистой летательной перепонкой, натянутой между удлиненными фалангами пальцев передних конечностей, передними и задними ногами и хвостом. Хорошо развиты грудные мышцы, на грудине есть киль.

Норные млекопитающие проводят часть жизни в норах, часть вне их — на поверхности земли, где и кормятся. Таковы из грызунов сурки, суслики, хомяки, полевки; из хищных — хорьки, корсаки, песцы, барсуки и др. Многие из них зимнеспящие, так как спячка возможна в норе, где нет опасности промерзнуть. Тело обычно сильное, мешковатое. Ноги относительно короткие. Есть приспособления к рытью нор — большие тупые когти. Мех обычно короткий, грубоватый.

Подземные млекопитающие практически не выходят на поверхность, **всю** жизнь проводя под землей, находясь там и убежище, и пищу. К этой Группе принадлежат кроты, слепыши, слепушонки, цокоры и др. Очень резко обособленная группа, тело вальковатое, шея укороченная, ноги короткие, но сильные, хвост небольшой, иногда почти не выдающийся из меха. Волосяной покров короткий, бархатистый, без ясного деления на ость и пух. Глаза слабо развиты, ушных раковин нет. Имеются приспособления для рытья — когги, зубы. Некоторые роют землю передними лапами, другие рыхлят ее зубами и отбрасывают ногами.

Половодные млекопитающие часть времени проводят в воде, а часть — на суше (в норах, на поверхности почвы). Хорошо плавают и ныряют. В эту группу входят выдра, норка, бобр, калан, ондатра, водяная крыса, выхухоль и др. Для них характерно вытянутое или компактное тело, короткие ноги, пальцы которых (все или часть) связаны плавательной перепонкой. Хвост часто уплощенный с боков или даже сверху вниз покрыт чешуйками и служит рулем при плавании. Волосяной покров с грубоватой остью, хорошо смазан жиром. Ость плотно закрывает густой пух. Ушиные раковины слабо выражены.

Водные млекопитающие всю или большую часть жизни проводят в воде (ластоногие, китообразные, сирены). Тело обтекаемой формы, торпедообразное, конечности превращены в ласты, шея укорочена. Волосяной покров в той или иной мере редуцирован. Под кожей мощные отложения жира. Китообразные имеют вытянутое торпедообразное тело со слабо выраженной шеей и малоподвижной головой, заканчивающееся горизонтальным хвостовым плавником, который и служит движителем. Задние конечности подверглись редукции. Кожа голая. Они никогда не выходят на сушу, размножаются в воде. Несколько меньше связаны с водой ластоногие. Тело обтекаемой формы с хорошо выраженной шеей и подвижной головой. Волосяной покров зачастую хорошо развит, имеются задние конечности, не так сильно развит слой подкожного жира. На суше или льдинах в период размножения образуют большие скопления — лежбища, здесь же происходит рождение потомства и спаривание.

Деление млекопитающих на экологические группы достаточно условно, так как эти группы связаны промежуточными формами.

Большая часть млекопитающих принадлежат к числу оседлых животных. Но многие из них совершают периодически длительные миграции, которые часто носят сезонный характер. Так, например, летучие мыши совершают перелеты как птицы, улетая зимой далеко на юг. Каспийские тюлени зимой держатся у берегов Ирана, а весной скапливаются у о. Тюлений. Морские котики, размножающиеся на Командорских островах, на зиму откочевывают к берегам Японии. Иногда кошки млекопитающих не связаны с определенным сезоном года и вызываются недостатком пищи в данном районе. Белки нередко совершают длинные перекочевки в пределах лесной зоны в поисках кормовых мест, известны массовые миграции леммингов.

Численность многих видов зверей значительно колеблется. В годы с благоприятными условиями среды они интенсивно размножаются, в годы же, когда условия жизни ухудшаются, численность их падает. Хорошо известны так называемые «мышиные годы», когда количество мышей и нор левок резко возрастает, затем следует быстрый спад их численности. Причинами колебаний численности зверей могут быть недостаток кормов, различные заболевания, размножение врагов данного вида и другие факторы. Поэтому колебания численности различных видов зачастую взаимосвязаны, как, например, колебания численности зайца-беляка и рыси.

Большинство млекопитающих вследствие способности к терморегуляции жизнедеятельны круглый год. Но некоторые виды могут ви-

дать в спячку или длительный сон; происходит это обычно зимой, но иногда и летом. В зимний сон впадают медведи, енотовидные собаки, барсуки. В течение ряда зимних месяцев они лежат в берлоге или норе неподвижно, не питаясь и существуя за счет, накопленных запасов жира. Однако температура тела, дыхание и пульс у них во время сна остаются обычными. В любой момент зверь может проснуться, например, если его потревожить. Сурки, суслики, многие летучие мыши и ряд других зверей впадают на зиму в глубокую спячку. В этом случае состояние животного близко к анабиозу. Температура тела понижается иногда почти до 0 °С, колеблясь в зависимости от температуры окружающей среды. Дыхание и пульс становятся едва заметными. Как зимний сон, так и зимняя спячка обусловлены трудностями добывания пищи зимой. Некоторые обитатели засушливых районов впадают в летнюю спячку (суслики, сурки) на время, когда выгорает скучная растительность этих мест и питаться становится практически нечем.

Многие млекопитающие используют в питании достаточно разнообразные корма, т. е. по характеру питания они являются всеядными (кабан, барсук, бурый медведь). Жевательная поверхность коренных зубов этих животных несет много небольших бугорков, кишечник средней длины. В рационе некоторых млекопитающих преобладают корма животного происхождения, такие млекопитающие относятся к зоофагам. Желудок у них устроен довольно просто, кишечник относительно короткий, а зубы острые с коническими или остробугорчатыми коронками.

Среди них можно выделить хищных млекопитающих, которые питаются в первую очередь теплокровными позвоночными (кошачьи). У этих хищников особенно развиты клыки и коренные зубы с режущими краями. Насекомоядные (землеройки, летучие мыши) питаются преимущественно насекомыми и другими мелкими беспозвоночными. Ихтиофаги (тюлени, дельфины, выдры) охотятся за рыбой и другими водными животными. Планктоноядные (киты) — самые крупные из современных млекопитающих и позвоночных животных, питаются планктоном и мелкой рыбой. Трупоядные, или падальщики (гиены), питаются в основном трупами и отбросами.

В рационе многих млекопитающих преобладают корма растительного происхождения, такие млекопитающие относятся к растительноядным, или фитофагам. Растительноядные млекопитающие отличаются очень длинным кишечником и нередко усложненным строением желудка, что связано со сложными процессами переваривания растительной пищи. Коренные зубы имеют плоскую жевательную поверхность со складками эмали или с мелкими тупыми бугорками. •их млекопитающих в зависимости от характера поедаемой ими пищи можно подразделить на несколько групп: травоядные (быки, пищухи), в рационе которых преобладают травы; питающиеся древесной растительностью (слоны, лоси); плодоядные (сони, приматы), предпочитающие мягкие сочные плоды и ягоды; семеноядные (белки, мыши, хомя-

ки), питающиеся сухими плодами и семенами растений; корнееды — покоры, слепушонки; листоеды — ленивцы, коалы.

Некоторые звери делают на зиму значительные запасы кормов. Например, белки и бурундуки запасают орехи, желуди, шишки. Пищухи (сеноставки) осенью делают небольшие стожки сена из травы, высущенной на ветру и солнце.

ХОЗЯЙСТВЕННОЕ ЗНАЧЕНИЕ МЛЕКОПИТАЮЩИХ

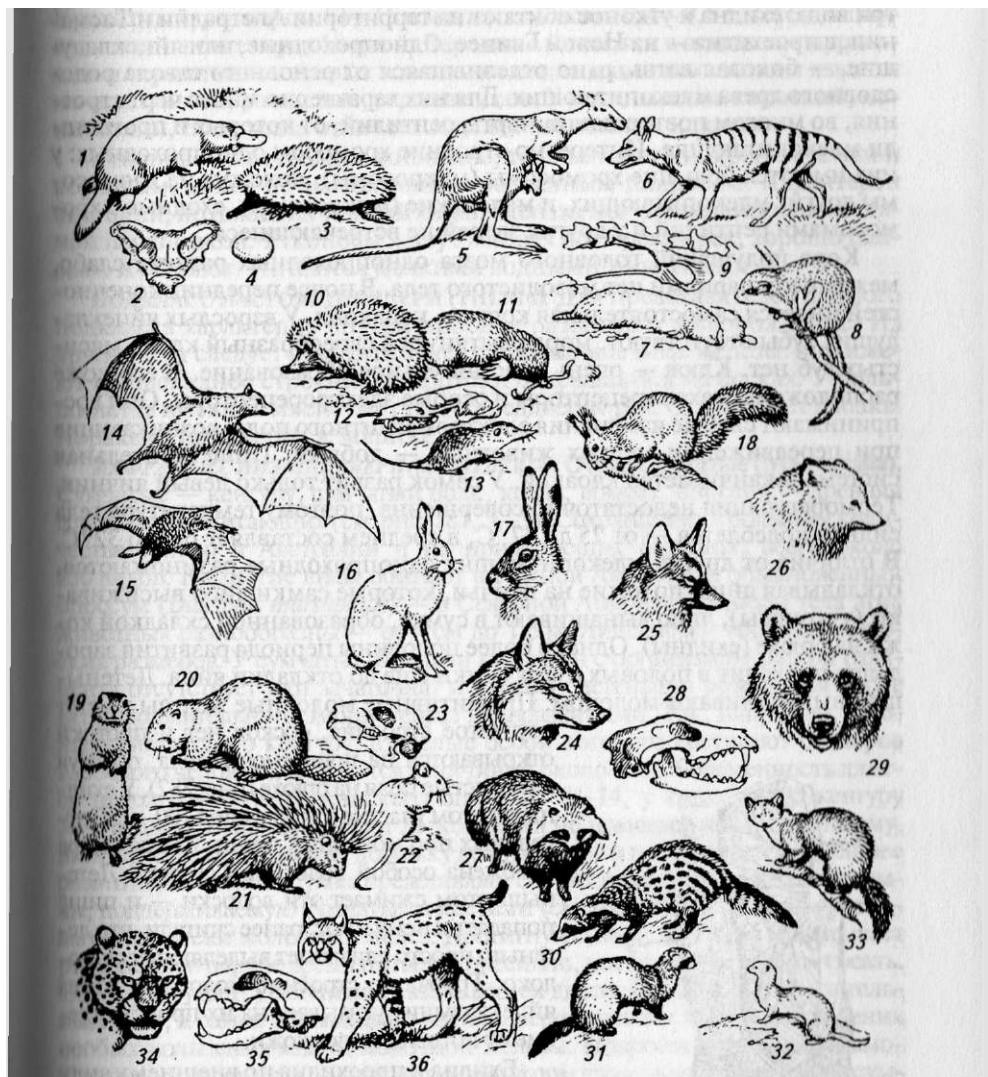
Млекопитающие имеют огромное значение для хозяйственной деятельности человека. К этому классу относится большинство сельскохозяйственных животных — крупный и мелкий рогатый скот, лошади, свиньи, верблюды, кролики, собаки и др. Многие млекопитающие являются декоративными (морские свинки, хомячки) или лабораторными животными (крысы, мыши), которых в большом количестве содержат и разводят.

К этому же классу относятся все объекты звероводства, находящиеся на разной стадии одомашнивания — серебристо-черные лисицы, голубые песцы, норки, соболи, нутрии и другие виды. Разведение в неволе сопровождается выведением новых пород животных. Только в России разводят около 500 отечественных пород, внутрипородных линий и групп, относящихся примерно к 30 видам сельскохозяйственных животных. Одних пород собак в мире официально зарегистрировано около 400. Многие млекопитающие служат объектами охоты, давая ежегодно пушнину, мясо, кожу и другие продукты. Широко был до недавнего времени развит промысел морских млекопитающих — китов, тюленей, дававший жир, мясо, а также шкуры. Многие хищные звери уничтожают различных грызунов, насекомоядные (например, ежи) и летучие мыши истребляют огромное количество насекомых — вредителей растений.

Но некоторые млекопитающие — суслики, хомяки, мыши, полемки, крысы, тушканчики и другие сами являются важнейшими вредителями сельского хозяйства. Один суслик за год уничтожает 3—4 кг зерна, а численность сусликов в южных районах огромна. Многие грызуны портят лесные и садовые насаждения, огородные растения. Крысы и мыши приносят многомиллионные убытки уничтожением и порчей продовольственных запасов. Некоторые хищники, особенно волки, наносят урон животноводству. Ряд грызунов являются опасными переносчиками и хозяевами возбудителей различных заболеваний человека. Так, распространение многих опасных заболеваний, в том числе и чумы, связано с крысами. По данным академика Е. Н. Павловского, от пандемии чумы в Европе с XI по XIX в. умерло 25 млн человек, четверть тогдашнего населения планеты.

СИСТЕМАТИЧЕСКИЙ ОБЗОР МЛЕКОПИТАЮЩИХ

ПОДКЛАСС ПЕРВОЗВЕРИ (PROTOTHERIA). Отряд Однопроходные (Monotremata). Представители этого отряда — наиболее примитивные млекопитающие. Было время, когда они были широко распространены по земному шару, но сейчас водятся только в Австралии, Тасмании и Новой Гвинее (см. рис. 281, 1 и 3). В настоящее время сохранилось только



и

Рис. 281. Представители разных отрядов млекопитающих:

1 — утконос; 2 — плечевой пояс утконоса спереди; 3 — ехидна; 4 — яйцо ехидны; 5 — рыжий кенгуру; 6 — сумчатый волк; 7 — сумчатый крот; 8 — сумчатая белка; 9 — таз сумчатого с сумчатыми костями; 10 — еж; 11 — крот; 12 — череп крота со сплошным рядом зубов; 13 — землеройка; 14 — крылан; 15 — летучая мышь; 16 — заяц белый; 17 — заяцрусак; 18 — белка; 19 — суслик; 20 — бобер; 21 — дикобраз; 22 — мышь; 23 — череп грызуна; 24 — Поль; 25 — лисица; 26 — песец; 27 — енотовидная собака; 28 — череп собаки; 29 — бурый медведь; 30 — африканская виверра; 31 — хорек; 32 — ласка в зимнем меху; 33 — соболь; 34 — леопард; 35 — череп кошки; 36 — рысь

три вида: ехидна и утконос обитают на территории Австралии и Тасмании, а проехидна — на Новой Гвинее. Однопроходные, или яйцекладущие, — боковая ветвь, рано отделившаяся от основного ствола рода словного древа млекопитающих. Для них характерно много черт строения, во многом повторяющих черты рептилий, от которых и произошли млекопитающие. Интересно строение хромосом однопроходных: у них имеются большие хромосомы (макросомы), похожие на хромосомы других млекопитающих, и маленькие (микросомы), сходные с хромосомами рептилий и у других зверей не встречающиеся.

Кора полушарий головного мозга однопроходных развита слабо, между полушариями нет мозолистого тела. В поясе передних конечностей имеется самостоятельная кость — коракоид. У взрослых яйцекладущих зубы отсутствуют, морда вытянута в своеобразный клюв, мясистых губ нет. Клюв — очень чувствительное образование. В его коже расположены mechanoreцепторы и особые электрорецепторы. Они вовремя принимают слабые изменения электромагнитного поля, возникающие при передвижении мелких животных — добычи. Пищеварительная система заканчивается клоакой. У самок развит только левый яичник. Терморегуляция недостаточно совершенна, поэтому температура тела сильно колеблется — от 25 до 37 °C, в среднем составляет около 32 °C. В отличие от других млекопитающих однопроходные размножаются, откладывая яйца, похожие на птичьи, которые самки либо высаживают (утконосы), либо вынашивают в сумке, образованной складкой кожи на брюхе (ехидны). Однако более половины периода развития зародыша проходит в половых путях самки еще до откладки яйца. Детеныши выкармливают молоком. Примитивные молочные железы имеют:

трубчатое строение, сосков нет и протоки открываются на поверхности тела, образуя железистые поля на брюхе (рис. 282). У ехидны на этом участке открывается 100—150 отдельных пор молочных желез. Каждая пора снабжена особой волосянной сумкой. Детеныш ртом сжимает эти волоски — и пищи попадает к нему в рот; ранее считали, что детеныш просто слизывает выделяющееся молоко. Трубчатое строение молочных желез яйцекладущих указывает на их происхождение от потовых желез кожи.

Ехидна и проехидна по внешнему виду напоминают ежа, тело их покрыто грубыми волосами и иглами длиной 6—8 см, морда вытянута в своеобразный клюв, лапы вооружены мощными когтями. Ехидна — зверь с массой тела от 2 до 7 кг и длиной около 50 см. Самка откладывает лишь одно яйцо массой всего около 1,5 г, которое вынашивает в складке кожи на брюхе,

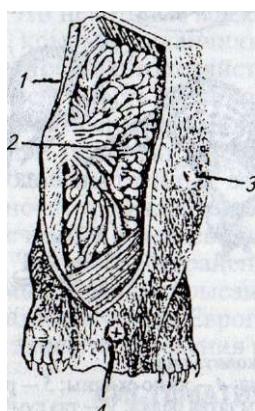


Рис. 282. Млечные железы утконоса:

1 — железы; 2 — выводные протоки; 3 — железистое поле; 4 — клоака

Крохотный, размером 13—15 мм и массой всего 0,4—0,5 г, детеныш появляется на свет через 10 дней. Взрослые питаются в основном муравьями и термитами, которых извлекают очень длинным липким языком **из**-под камней, расщелин, термитников. У ехидна клюв короткий, комический, проехидна отличается длинным тонким клювом.

Утконос — полуводное животное, тело которого покрыто густым и коротким, долго не намокающим волосяным покровом. Характерен широкий, похожий на утиный клюв. Пальцы ног соединены плавательной перепонкой. Утконосы живут в норах по берегам рек, хорошо плавают и ныряют. Питаются мелкими водными животными.

ПОДКЛАСС НАСТОЯЩИЕ ЗВЕРИ (THERIA). Для представителей данного подкласса характерно внутриутробное развитие и живорождение. Из других черт следует отметить более сложные млечные железы: они имеют грудевидное строение, и протоки открываются на сосках. У большинства имеются мясистые губы и преддверие рта. Отсутствует клоака. Коракоид прирастает к лопатке.

ИНФРАКЛАСС НИЗШИЕ ЗВЕРИ (METATHERIA). Отряд Сумчатые (Marsupialia). Сумчатые — кенгуру, сумчатый волк, коала, wombат, опоссум — древняя группа млекопитающих (см. рис. 281, 5—9). Сохранились в большом количестве только в Австралии и на прилегающих островах, всего около **250 видов**. Немногие виды обитают в Южной Америке, а обыкновенный **опоссум** (*Didelphis marsupialis*) — в Северной Америке. Размеры тела этих животных разнообразны — от 4 см до 1,6 м. Яйцеклетки микроскопических размеров. У сумчатых зародыш не связан с организмом матери (плацента отсутствует или зачаточна) и поэтому детеныши рождаются после непродолжительного эмбриогенеза недоразвитыми. Так, например, детеныш гигантского кенгуру, взрослые особи которого достигают размеров взрослого человека, рождается величиной около 3 см. Беременность длится недолго — у американского опоссума 8—14, у гигантского кенгуру **30—40** дней. Роды протекают легко, самка не помогает новорожденному, **лишь** иногда пролизывает дорожку до сумки, где и происходит дальнейшее развитие детеныша. Сумка представляет собой складку кожи на брюхе самки, поддерживаемую сумчатыми костями (см. рис. 281, 9), в сумке располагаются соски молочных желез. Достигнув сумки, детеныш захватывает гром сосок и первое время питается пассивно, поскольку не в силах сосать. **Сосок** разбухает, у детеныша разобщаются дыхательный и пищеварительный пути, и самка впрыскивает молоко периодически за счет сокращения **особых** кольцевых мышц молочной железы. Подросший детеныш **самостоятельно** сосет мать, начинает поедать корм взрослых и выходит из сумки, **но** при опасности прячется туда. Продолжительность лактации от **60 дней** у мелких до 250 дней у крупных видов. У сумчатых сохраняются примитивные черты организации. Мозг у них мал, борозды отсутствуют, **или их** мало. В скелете имеются сумчатые кости. Половые пути самки раздвоены, у некоторых видов непарным является только влагалище. Пенис самцов часто раздвоенный. Смене подвержены только передние зубы. Температура тела в среднем выше (около 36 °C), чем у однопроходных, и **колеблется** в пределах 3—5 °C. Зубы обычно слабо дифференцированы.

В Австралии, где не было плацентарных млекопитающих, сумчатые образовали ряд экологических типов — наземных, роющих, древесных, хищных, растительноядных и всеядных, — имеющих конвергентное сходство с подобными группами плацентарных.

Из сумчатых широко известны различные виды кенгуру, которые передвигаются прыжками на сильно развитых задних ногах; укороченные передние конечности служат для захвата пищи. На эвкалиптовых деревьях живет медлительный сумчатый медведь коала, питающийся листьями. Сумчатые куницы, сумчатые белки и сумчатые летяги ведут древесный образ жизни. В почве роют сложные и глубокие норы слепые сумчатые кроты. Сумчатый дьявол — хищник. В Америке встречаются своеобразные сумчатые — опоссумы.

Многие сумчатые дают ценную пушину. Мясо кенгуру хорошего качества. Некоторые виды являются объектами зоокультуры.

В палеогене сумчатые были широко распространены по материкам земного шара, но позднее почти везде (кроме Австралии и Америки) были вытеснены более совершенными плацентарными млекопитающими.

ИНФРАКЛАСС ВЫСШИЕ ЗВЕРИ, ИЛИ ПЛАЦЕНТАРНЫЕ (EVTHERIA). К данному инфраклассу относится большинство современных млекопитающих. Как и у сумчатых, яйцеклетки плацентарных также имеют микроскопические размеры. Зародыши развиваются в матке, при этом наличие связи с организмом матери через временный орган — плаценту — позволяет им значительно дольше находиться в ее теле и появляться на свет более зрелыми, способными самостоятельно сосать.

Плацента образуется путем прирастания наружной оболочки зародыша — хориона — к стенке матки, в результате чего возникает плоское губчатое тело (см. рис. 280). Сам хорион образуется путем срастания аллантоиса и серозы. От хориона в глубь утолщенной стенки матки врастаят многочисленные тонкие выросты — ворсинки, снабженные густой сетью капилляров. Капилляры ворсинок и капилляры утолщенной стенки матки самки тесно переплетаются друг с другом, что позволяет кислороду, питательным веществам и другим компонентам диффузно поступать из крови матери в кровь эмбриона; из крови эмбриона в кровь матери поступают диоксид углерода и другие конечные продукты метаболизма. Из плаценты кровь по сосудам пуповины течет в тело зародыша и обратно.

Различают несколько типов плаценты. *Диффузная*, когда ворсинки распределяются по всей поверхности хориона равномерно (китообразные, многие копытные). *Дольчатая*, когда ворсинки собраны на отдельных участках хориона в виде пятен (большинство жвачных). *Зональная*, когда ворсинки располагаются широким поясом (некоторые хищные, хоботные). *Дискоидальная*, когда ворсинки собраны на одном четко ограниченном участке хориона, имеющем форму диска (грызуны, приматы).

Головной мозг плацентарных млекопитающих имеет более сложное строение, чем мозг сумчатых. Сумки нет, нет сумчатых костей, влага

лице непарное. Зубы, как правило, хорошо дифференцированы на резцы, клыки и коренные и подвержены смене (кроме собственно коренных). Подкласс плацентарных включает 17 отрядов.

Отряд Насекомоядные (*Insectivora*) — небольшие зверьки, отличающиеся примитивностью строения. Внешний вид насекомоядных разнообразен, но для всех характерен вытянутый подвижный хоботок на конце морды (см. рис. 280). Волосяной покров короткий, мягкий. Челюстей удлинен. Остробугорчатые зубы слабо дифференцированы. Головной мозг мал, полушария имеют гладкую поверхность. Питаются различными мелкими животными, хорошо развито обоняние.

В нашей стране насекомоядные представлены различными видами ежей, землероек, кротов, а также выхухолью. Обыкновенный еж (*Talpa europea*) ведет наземный образ жизни, активен ночью, зимой впадает в спячку. Землеройки обитают в подстилке, верхнем слое почвы, активны круглосуточно и круглогодично. Кроты ведут подземный, роющий образ жизни. Передние лапы служат для рытья, грунт выбрасывается наверх, где образуются характерные кучки земли — кротовины. Выхухоль — редкий зверек, ведущий полуводный образ жизни.

Отряд Рукокрылые (*Chiroptera*). Представители данного отряда приобрели способность активного полета. Передние конечности у них видоизменились в крылья. Несущей поверхностью крыльев служит тонкая кожистая летательная перепонка, натянутая между очень длинными вторым—пятым пальцами передних конечностей, предплечьем, плечом и задними конечностями; у многих эта перепонка захватывает и хвост. На грудине имеется небольшой киль, сама грудина хорошо развита, на ней располагаются грудные мышцы, обеспечивающие работу крыльев. Первый палец передней конечности и кисть задней остаются свободными, что помогает зверьку при лазанье и цеплянии (см. рис. 280). Отряд включает два подотряда — крыланы и летучие мыши. В нашей фауне встречаются летучие мыши, принадлежащие к трем семействам: 1) одковоносые, Бульдоговые и Гладконосые.

Рукокрылые активны в сумерках и ночью. Питаются разнообразной пищей. Представители нашей фауны — главным образом ночных насекомыми, которых ловят на лету. Хорошо развиты слух и осязание. Характерна способность к эхолокации. В полете они постоянно издают прерывистые звуки высокой частоты, которые отражаются от находящихся на пути зверька объектов, в том числе и от насекомых. Отраженный сигнал улавливается зверьком и анализируется, позволяя ему не только избежать столкновения, но и охотиться даже в полной темноте. Рукокрылые ведут одиночный и стайный образ жизни. В умеренных широтах активны в теплое время года, поэтому совершают сезонные миграции или впадают в спячку.

Привлечению летучих мышей способствует развешивание дуплянок с нижним летком, где они проводят светлое время суток.

Отряд Приматы (*Primates*). Отряд включает два подотряда — полуобезьяны и обезьяны (рис. 283). Для обезьян характерны крупные полушария головного мозга, кора которых образует сложную систему из-



Рис. 283. Представители разных отрядов млекопитающих:
1 — гренландский тюлень; 2 — дельфин; 3 — африканский слон; 4 — кабан; 5 — джейран; 6 — дхархар; 7 — индийский буйвол; 8 — як; 9 — антилопа канна; 10 — марал; 11 — северный олст.; 12 — нога парнокопытного; 13 — чепрачный тапир; 14 — нога тапира; 15 — индийский и < корог; 16 — лошадь Пржевальского; 17 — кошачий лемур; 18 — руконожка; 19 — долгоняйт; 20 — лапа лемура; 21 — череп лемура; 22 — игрунка; 23 — ревун; 24 — мандрил; 25 — мир тышка; 26 — череп гориллы; 27 — гибон; 28 — орангутан; 29 — шимпанзе; 30 — горилла

нилин и борозд, что определяет высокий уровень их высшей нервной деятельности. Глазницы направлены вперед. Большинство обезьян стопоходящие. Конечности хватательного типа, первый палец противопоставлен остальным. На пальцах имеются ногти. Одна пара молочных желез расположена на груди. Желудок простой.

Обезьяны обитают в тропиках и субтропиках. Представители наиболее многочисленного семейства мартышек обитают в Старом свете. Представители семейства Человекообразные обезьяны обитают в Юго-Восточной Азии (орангутан) и в Африке (горилла и шимпанзе — *Pan troglodytes*). Семейство Люди представлено одним видом — человек разумный (*Homo sapiens*).

Зубная формула

$$\frac{2}{2}, \frac{1}{1} \text{ e} - \frac{2}{2}, \text{ p t} \frac{2}{2}, \text{ t} \frac{3}{3} = \frac{8}{8} \times 2 = 32.$$

Отряд Зайцеобразные (Lagomorpha). К этому отряду относятся зайцы, кролики и небольшие зверьки — пищухи (см. рис. 281). От грызунов зайцеобразные отличаются строением костного нёба, имеющего вид узкой перемычки между зубными рядами. Зубная система напоминает зубную систему грызунов, но в верхней челюсти находятся сдвоенные резцы — спереди большие долотообразные, сзади них маленькие столбиковидные. Клыков нет, между резцами и коренными зубами пустое пространство — диастема. Питаются растительной пищей. У нас обычны заяц-беляк (*Lepus timidus*) и заяц русак (*L. europaeus*). Зайцы служат предметом охоты, а также объектом зоокультуры. Местами они могут наносить ущерб плодовым деревьям, объедая зимой кору. Домашних кроликов разводят ради мяса, шкурки и пуха.

Зубная формула кролика

$$\frac{i}{1}, \frac{c}{0}; \text{ pm} + \text{ t} \frac{2}{5} = \frac{8}{6} \times 2 = 28.$$

Отряд Грызуны (Rodentia) — самый многочисленный и широко распространенный отряд млекопитающих. Это мелкие или средней величины растительноядные млекопитающие (см. рис. 281). Общим признаком всех грызунов является своеобразное строение зубной системы, приспособленной к разгрызанию и разжевыванию твердой растительной пищи. Клыков нет, имеется диастема. В каждой челюсти по одной паре больших, долотообразных, лишенных корней и постоянно растущих резцов. Твердая эмаль покрывает переднюю поверхность резцов, поэтому стачиваются они неравномерно и всегда остаются острыми. Коренные зубы приспособлены к перетиранию жесткой растительной пищи. Они имеют широкую жевательную поверхность, обычно плоскую или несущую тупые бугорки или гребни. У многих видов коренные зубы не имеют корней и тоже растут в течение всей жизни зверька. Нижняя челюсть соединяется так, что может несколько сдвигаться

вперед и назад. Когда зверек грызет пищу, он выдвигает нижнюю челюсть немного вперед, так что концы резцов верхней и нижней челюстей сходятся, но коренные зубы при этом не соприкасаются. Когда же грызун приступает к измельчению пищи, нижняя челюсть сдвигается несколько назад, что приводит к соприкосновению поверхностей коренных зубов.

Зубная формула

$$i\frac{1}{1}; c\frac{0}{0}; p\frac{t}{t} + t\frac{4}{4} = \frac{-}{5} x 2 = 20.$$

У большинства грызунов сильно развита слепая кишка, в которой проходят процессы сбраживания пищи. Полушария головного мозга обычно гладкие, без извилин. Пальцы несут когти. Грызунам свойственна высокая плодовитость.

Грызуны имеют большое хозяйственное значение. Многие из них являются серьезными вредителями сельского и лесного хозяйства. Ряд грызунов служат переносчиками и вирусоносителями опасных заболеваний человека (чумы, туляремии). Но многие дают ценное мясо, пушину и служат важными объектами охотничьего промысла и зоокультуры (бобры, ондатры, нутрии, шиншиллы и др.). Некоторые виды являются декоративными и лабораторными животными (крысы, песчанки).

Практическое значение, в частности как вредители сельского и лесного хозяйства, имеют представители различных семейств.

Семейство Беличьи (Sciuridae) включает белок, бурундуков, сусликов и сурков. Суслики — зверьки величиной с белку, с вальковатым телом, короткими ногами и обычно небольшим хвостом. Населяют преимущественно лесостепные, степные и пустынные районы. Только своеобразный длиннохвостый суслик обитает в Восточной Сибири вплоть до Арктики, селясь на открытых местах среди тайги. Суслики живут в норах, часто колониями. На зиму впадают в спячку. Начинают размножаться в годовалом возрасте весной. Самки приносят 4—10 детенышней. Питаются различными растениями. Являются носителями ряда заболеваний, опасных для человека (в первую очередь чумы).

Семейство Соневые (Myoictidae) включает различные виды соней (полчок, орешниковая, садовая, лесная), внешне напоминающих белок, но значительно меньших размеров. Живут в дуплах деревьев и гнездах, которые сооружают на деревьях и кустах. Преимущественно ночные животные. На зиму залегают в длительную спячку (откуда и на звание этих животных). Крупные виды сонь — соня-полчок и садовая соня могут причинять местами вред садоводству, уничтожая плоды и ягоды. В России наибольший убыток от этих грызунов терпят сады Кавказа.

Семейство Тушканчиковые (Dipodidae). Своебразные грызуны, передвигающиеся прыжками на длинных задних ногах; передние конечности у них сильно укорочены и служат преимущественно для удара

I живания пищи. Ряд видов (большой, малый, мохноногий) населяет главным образом степную зону России.

Семейство Хомяковые (Cricetidae) включает хомяков, цокоров, полевок, хомячков. Хомяки (*Cricetus*) — зверьки средних размеров (величиной с большую крысу) с мешковатым телом, короткими лапами и небольшим хвостом. Окраска верха ржаво-серая, низа — черноватая, по бокам тела расположены большие белые пятна. Хомяки распространены на юге нашей страны, населяя поля, огороды, заросли бурьяна, овраги, перелески, поймы рек. Живут в норах, обычно поодиночке. Самки приносят за год 1—2 помета из 3—18 детенышей. Местами вредят посевам зерновых и огородаам. На зиму делают большие запасы зерна и корнеплодов.

Мохноногие хомячки (*Phodopus*) по форме тела похожи на хомяков, но значительно мельче их (обычно с крупной мышью). Окраска различная — серая, бурая, рыжеватая, песочная, часто с черной полоской на спине. Встречаются в степных, лесостепных и пустынных зонах России, где часто селятся на полях, огородах, бахчах и плантациях, нанося пред посевам. По образу жизни похожи на хомяков. Как лабораторных и декоративных животных разводят джунгарского хомячка и хомячка Роборовского.

Полевки (подсемейство *Microtinae*) похожи на мышей, но отличаются более коротким хвостом, длина которого, как правило, короче 1/2 длины тела, и широкой, тупой мордочкой. Зубы с плоской жевательной поверхностью и глубокими боковыми складками эмали. Живут в неглубоких норках. За год самки приносят несколько пометов из 3—10 детенышей. На зиму в спячку не впадают. Наиболее распространена обыкновенная полевка (*Microtus arvalis*), которая обитает преимущественно на полях, лугах и огородах. Она питается зелеными частями растений, зерном и корнеплодами, местами принося вред растениеводству. Рыжая полевка (*Clethrionomys glareolus*) похожа на обыкновенную полевку, но отличается рыжеватой окраской спины. Обитает в лесу. Вредит, уничтожая семена и повреждая кору древесных пород в лесных посадках. Ондатра (*Ondatra zibethicus*) ведет полуводный образ жизни, икклиматизирована в начале прошлого века на всей территории России.

Семейство Песчанковые (Cerbillidae). Большая группа грызунов, внешне похожих на крыс или крупных мышей, но с хвостом, густо покрытым волосами, образующими на конце кисточку (у крыс хвост одет кольцами роговых чешуек и редкими отдельными волосками). Обитали пустыни. Живут как среди песков, так и на участках с твердым грунтом, охотно селятся на культурных землях, где могут наносить вред посевам. Переносчики чумы. Для изучения природно-очаговых инфекций КИ паразитарных заболеваний некоторое виды разводят в лабораторных условиях (монгольская песчанка, большая песчанка и др.).

Семейство Мышиные (Muridae). Самое обширное семейство современных грызунов, насчитывающее около 400 видов. Наиболее обычны домовая мышь (*Mus musculus*) грязно-серого цвета с относительно коротким хвостом, лесная мышь коричневатая и чисто белая снизу,

с длинным тонким хвостом, полевая мышь с коричневато-рыжей спинкой, вдоль которой тянется черная полоска, и мышь-малютка (*Mus minutus*), резко отличающаяся от других представителей своего семейства размерами. Домовая мышь в северных районах живет преимущественно в домах, питаясь продуктами людей. Но на юге этот вид частично живет и на полях, принося большой вред посевам. Полевая и лесная мыши держатся в природных угодьях и местами также вредят сельскохозяйственным культурам.

Крысы (*Rattus*) отличаются от мышей более крупными размерами. В России обитает несколько видов, из которых наиболее распространена серая крыса-пасюк (*R. norvegicus*). Крысы наносят большой вред, уничтожая и повреждая продовольственные продукты. Все они являются носителями и распространителями возбудителей некоторых опасных для людей инфекций.

В нашей стране проводятся широкие работы по истреблению вредных грызунов с использованием сложных химикатов и аппаратуры. Наибольшее распространение имеют химические способы истребления этих вредителей. Но нужно помнить, что химические меры борьбы с грызунами требуют значительных затрат и часто, особенно при неправильном применении, приводят к гибели многих других животных. Поэтому химические методы по возможности должны заменяться механическими, биологическим и бактериальным методами истребления. В борьбе с мышами и крысами, а также некоторыми другими грызунами иногда используют приманки, зараженные бактериями мышиного 11-го крысиного тифа. Этот бактериальный метод особенно rationalен в условиях различных населенных пунктов, где применение многих химических и химических препаратов недопустимо.

В лабораторной практике традиционно используют альбиносные формы домовой мыши и серой крысы.

Отряд Хищные (Carnivora) объединяет наземных и полуводных млекопитающих, питающихся в основном пищей животного происхождения (см. рис. 281). Зубы четко дифференцированы на резцы, клыки и коренные. Резцы довольно мелкие. Клыки, напротив, большие, копьевидные, острые. Коренные зубы обычно остробугорчатые. Почти у всех хищных последний предкоренной зуб верхней челюсти и первый истинно коренной нижней челюсти выделяются большими размерами и эти зубы называются хищными. Число коренных зубов обычно сокращено. К данному отряду относятся семейства Псовые, Медвежьи, Куши, Кошачьи, Гиеновые, Енотовые и Виверровые. В нашей стране обитают представители первых четырех семейств. Некоторые хищники являются носителями вируса бешенства, опасного для человека и домашних животных.

Семейство Псовые (Canidae) объединяет пальцеходящих зверей средних размеров, приспособленных к бегу. Тело уплощено с боков, морда удлинена, хвост хорошо опущен, конечности высокие. На передних ногах по пять, а на задних по четыре пальца. Когти не втяжные, туго сжимающиеся. Хищники, преследующие свою добычу. Часто образуют семьи и

стай. Ведут (кроме времени размножения) бродячий образ жизни. Потомство приносят в норах, логовах. Размножаются один раз в году весной. Представители нашей фауны — волк, шакал, обыкновенная лисица, корсак, песец, енотовидная собака, красный волк. К этому семейству относится и домашняя собака (*Canis familiaris*). Лисица и песец — традиционные объекты охоты и звероводства. Волки там, где их численность еще высока, режут домашний скот и диких копытных.

Зубная формула волка

$$1\frac{3}{3}; c\frac{1}{1} p\frac{4}{4} t\frac{4}{3} \equiv -\frac{1}{1} 0 \times 2^n = 42.$$

Семейство Медвежьи (Ursidae) объединяет крупных стопоходящих зверей с массивным туловищем на мощных, но относительно коротких лапах, с очень коротким, спрятанным в волосяном покрове хвостом. Когти не втяжные. Медведи всеядные животные (белый медведь — хищник). На зиму (кроме самцов белых медведей) впадают в спячку. В России, в Арктике, обитает белый медведь, в лесной зоне — бурый медведь и на Дальнем Востоке — белогрудый медведь.

Семейство Куньи (Mustelidae) включает зверьков мелких и средних размеров, различного телосложения, массой от 100 г до 40 кг. Стопоходящие или полустопоходящие. Когти не втяжные, но острые. Большинство — настоящие хищники, но есть и всеядные виды (барсук). Размножаются весной. В спячку впадает только барсук. Многие виды дают ценную пушину и являются важными объектами охоты и звероводства (соболь, американская норка). Ряд видов (степной хорь, ласка, горностай) истребляют грызунов-вредителей. Семья степных хорьков, например, за лето уничтожает до 300 сусликов. В фауне России встречаются 18 видов, в том числе и акклиматизированная американская норка, а также соболь (*Martes zibellina*), европейская норка, ласка, горностай, колонок, лесная куница, черный хорь, речная выдра, калан и др.

Семейство Кошачьи (Felidae) — наиболее специализированные хищники с вытянутым, гибким телом и округлой головой, добывающие своих жертв путем подкатаивания. Только гепард (у которого не втяжные когти) и в какой-то мере рысь преследуют добычу. Это пальцоходящие звери, лапы с очень острыми втяжными когтями. Хищные **зубы** развиты очень хорошо. Ведут наземный образ жизни, многие хорошо лазают по деревьям. Из мелких кошек в России обитают европейская лесная кошка (*Felis silvestris*), степная кошка (*F. libyca*), хаус, из средних — рысь, из крупных — тигр, леопард, ирбис.

Отряд Ластоногие (Pinnipedia). Систематически очень близок к хищным. Отряд включает семейства моржей, ушастых тюленей и настоящих тюленей (рис. 283). Это крупные животные массой от 40 кг до 4 т. Ластоногие большую часть времени проводят в воде, выходя на сушу или лед для отдыха, спаривания, рождения молодняка и во время линьки. Это наложило отпечаток на их организацию. Тело ластоногих **вытянутое**, торпедообразное. Конечности имеют вид ластов. В воде ос-

новным движителем являются задние конечности. При движении по суше или льду настоящие тюлени (гренландский тюлень, каспийский тюлень, кольчатая нерпа) используют только передние конечности, задние ласты направлены всегда назад и не участвуют в передвижении. У моржей, сивучей и морских котиков задние ласты могут подгибаться вперед и помогают животным двигаться по твердому субстрату. У моржей на коже имеются редко расположенные одиночные грубые волосы. У взрослых тюленей волосяной покров редкий и жесткий, состоящий только из грубых остьевых волос (ценятся шкурки молодняка некоторых видов — бельков). Волосяной покров морских котиков густой, с плотным пухом. Ластоногие имеют толстый слой подкожного жира, который сохраняет внутреннее тепло тела.

Моржей добывают местные народности ради мяса, жира и огромных клыков (моржовая кость), идущих на косторезные изделия. При промысле тюленей получают технический жир, кожевенное и меховое сырье. Шкуры молодых морских котиков ценятся как высококачественное меховое сырье. Северные морские котики образуют лежбища на островах тихоокеанского побережья, где их охраняют и проводят выборочный забой молодняка.

Отряд Китообразные (Cetacea). Крайне специализированная группа млекопитающих, приспособившаяся к обитанию только в водной среде (см. рис. 283). На сушу не выходят. Тело торпедообразной формы, шея не выражена, передние конечности превратились в листы, задние редуцированы, на конце туловища имеется горизонтальный плавник, служащий основным органом движения животного. У некоторых видов имеется и спинной плавник. Кожа голая, лишенная волос, очеш, эластичная. Потовые и сальные железы отсутствуют. Под кожей лежит толстый жировой слой, играющий роль теплоизолирующей прослойки (у крупных видов его толщина достигает 50 см). Позвоночник слабо дифференцирован, скелет задних конечностей отсутствует. Ноздри смещены на верх головы. Легкие отличаются большой емкостью. **Синий** кит, например, ныряя, может вдохнуть в легкие до 14 тыс. л воздуха. Большой объем легких позволяет китообразным долго находиться под водой (15—45 мин) и нырять на большую глубину. Рекордсменом в этом отношении является кашалот (*Physeter macrocephalus*): зарегистрировано его пребывание под водой в течение 1 ч 52 мин и погружение на глубину до 3 тыс. м. В мышцах много миоглобина. Зрение и слух развиты хорошо, зубатые киты способны к эхолокации. Млечных желез одна пара, молоко очень жирное. Китообразные рожают в воде обычно одного очень крупного детеныша — до половины длины матери.

Китообразные делятся на два подотряда — Зубатые киты и Усатые киты. У зубатых китов (*Odontoceti*) имеются однородные конические зубы, помогающие добывать рыбу и головоногих моллюсков, которыми они питаются. Одно дыхательное отверстие — дыхало. К зубатым китам относятся семейства Речные дельфины, Кашалоты (кашалот — самый крупный представитель, длина до 20 м), Клюворыльные и Дельфины (белуха, касатка, афалина, белобочка). У усатых китов (*Mystacoceti*)

чубов нет, но во рту имеется своеобразный цедильный аппарат из двух рядов роговых пластин (китовый ус), несущих по нижнему краю бахрому из волокон. Забрав в пасть воду, киты выталкивают ее обратно через ряды пластин китового уса, и находящиеся в воде мелкие животные задерживаются бахромой пластин и заглатываются. Дыхательные отверстия парные. К усатым китам относятся семейства Гладкие киты (гренландский, южный), Серые киты и Полосатики (синий, горбач). Синий кит — самое крупное животное, рекордный экземпляр достигал в длину 33,6 м и весил 160 т.

Китобойный промысел, который велся преимущественно в морях Арктики и Антарктики, давал очень ценный технический и медицинский жир, мясо, амбрю, другие виды сырья и продуктов. В настоящее время численность, в первую очередь усатых китов, подорвана настолько, что некоторые виды находятся на грани исчезновения. Промысел китов не ведется с 70-х гг. прошлого века.

Отряд Хоботные (Proboscidea). К данному отряду принадлежат слоны — наиболее крупные наземные животные (высота в плечах 3—4 м, масса 4—5 т). Для них характерны длинный подвижный хобот — удлиненный нос, сросшийся с верхней губой, бивни — парные резцы передней челюсти, выступающие изо рта и растущие в течение жизни (см. рис. 283). Ноги пятитипные, каждый палец одет копытцем. Кожа почти голая. Клыков нет, в каждой половине челюсти функционирует по одному коренному зубу, которые при снашивании заменяются. Слоны живут до 50—80 лет, половой зрелости достигают в 10—16 лет. Рождают по одному детенышу, продолжительность беременности 18—22 мес. Индийский слон хорошо приручается и используется на сельскохозяйственных и лесозаготовительных работах. Бивни имеют только самцы. У более крупного африканского слона бивни имеются как у самцов, так и у самок. На севере Сибири в вечной мерзлоте иногда находят туши вымерших мамонтов, живших в Евразии в ледниковый период.

Отряд Непарнокопытные (Perissodactyla). Крупные растительноядные животные с наиболее развитым третьим пальцем, по которому проходит ось конечности. Конечные фаланги пальцев защищены роговыми копытами. Желудок однокамерный. Слепая кишечника большая. Ключицы отсутствуют. Длина кишечника в 12 раз больше длины тела. В паузе расположены два соска.

Отряд делится на три семейства — Тапировые, Носороговые и Лошадиные (см. рис. 283).

Семейство Тапировые (Tapiridae) включает четыре вида, обитающие в Южной Америке, и один вид — чепрачный тапир — в Юго-Восточной Азии. Это наиболее примитивные представители отряда, характеризующиеся тем, что у них на передних ногах по четыре, а на задних по три пальца с небольшими копытцами. Морда с коротким подвижным хоботком.

Семейство Носороговые (Rhinoceratidae). К этому семейству относятся массивные звери (до 3600 кг) с толстой почти голой кожей, с довольно короткими трехпалыми конечностями. Голова большая, на

морде расположен рог (до 158 см длиной), иногда позади — второй; рога являются производными эпидермиса. Обитают в Африке (самый крупный белый носорог и черный носорог) и в Южной Азии (три вида).

Семейство Лошадиные (Equidae). Телосложение стройное, конечности высокие, на каждой ноге хорошо развит и одет копытом только средний (III) палец, от II и IV пальцев сохранились лишьrudименты — «грифельные косточки», а I и V — редуцированы. Волосяной покров плотный, низкий и густой, на шее короткая грива, хвост с длинными волосами. Лицевой отдел черепа сильно удлинен. Зубная формула

$$\begin{matrix} i & 3 & . & e & 0 & z & d \\ 3' & 0 & - & \Gamma & p & 3 & 3 \end{matrix} \quad \begin{matrix} 3 \\ 3 \end{matrix} \quad \begin{matrix} 3 = 9 - 1 & 0 \\ 9 - 10 \end{matrix} \quad \begin{matrix} x & 2 = 36 - 40 \end{matrix}$$

Зубы постоянно растущие с плоскими жующими поверхностями, имеют сложноскладчатую коронку.

Это обитатели открытых пространств, приспособленные к быстрому и продолжительному бегу со скоростью 50—60 км/ч. Держатся небольшими косяками по 3—10 голов, иногда образуют большие стада. Беременность около года, рождают одного, реже двух жеребят, которые вскоре способны следовать за матерью.

Семейство объединяет лошадь Пржевальского (сохранившуюся и некоторых зоопарках мира), зебру, дикого осла, кулана.

Ослы отличаются от лошадей узкими копытами, длинными ушами и короткой шерстью на хвосте, кроме концевой кисти удлиненных полос. В Азии сохранились куланы, совмещающие признаки лошадей и ослов. Родоначальниками важнейших домашних животных — лошади и осла являются тарпан (*Equus caballus* рис. 284) и африканский осел (*E. asinus*). В сельском хозяйстве ряда стран в качестве упряженных и верховых животных используют также гибриды осла и кобылы — мулом. Численность большинства видов низкая, а лошадь Пржевальского находится на грани полного исчезновения.

Отряд Парнокопытные (Artiodactyla). Животные размерами от мелких до крупных (рис. 283), растительноядные или всеядные, ведущие наземный образ жизни (лишь бегемоты тесно связаны с водоемами). Отличительная особенность — на передних и задних конечностях по четыре пальца (один редуцирован). Конечные фаланги пальцев одеты роговыми копытами. Наибольшее развитие получили III и IV пальцы, между которыми проходит ось тела, II и V пальцы значительно короче или совсем редуцированы. Ключицы отсутствуют, так как конечное i и двигаются в основном в плоскости, параллельной оси тела. Кости конечностей удлинены, что является приспособлением к быстрому бегу. Многие представители отряда имеют важное хозяйственное значение. Почти везде они служат объектами промысла, дающими мясо, кожу, сырье для фармацевтической промышленности, трофеи.

Отряд включает два подотряда: Нежвачные и Жвачные.

Подотряд Нежвачные (Nonruminantia) включает свиней, пекарей и бегемотов. Это животные с массивным телом, короткой толстой шеей.

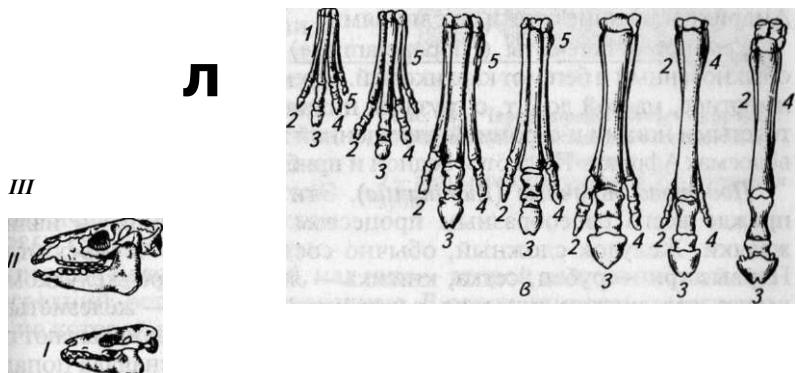
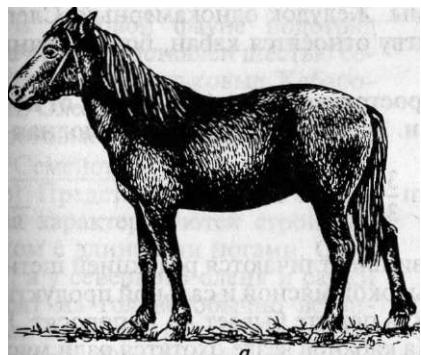


Рис. 284. Происхождение лошади:
I - тарпан — предок современных лошадей; б — изменение размеров и строения черепа
I ряду предков лошади (I — эогиппус; II — протогиппус; III — мегариппус); в — измене-
ние передних конечностей в эволюционном ряду предков лошади (1—5 — нумерация
Мальцев)

Невысокими ногами и коротким хвостом. Голова вытянута, часто клиновидная, рогов не бывает. Кожа толстая, с хорошо развитым слоем подкожного жира. Волосяной покров редкий, иногда почти отсутствует. На ногах по четыре пальца, из которых два средних развиты сильнее боковых, лишь у пекари задние конечности трехпалые. Клыки сильно развиты, коренные зубы многобугорчатые. Пища пережевывается во рту и по пищеводу поступает непосредственно в желудок. У самок на Грохе два ряда сосков.

К семейству Свиньи (Suidae) относятся животные средних размеров, отличающиеся массивным, сжатым с боков телом, короткой шеей, удлиненной конической головой с хрящеватым «пятачком» на конце, ороткими сильными ногами. Это всеядные животные. Держатся группами.

пами, реже поодиночке. Полигамы. Желудок однокамерный. Слепая кишка короткая. К этому семейству относятся кабан, бородавочник, кистеухая и лесная свиньи.

В Европе и Азии широко распространен кабан (*Sits scrofa*), от которого произошли домашние свиньи. Зубная формула кабана полная

$$\Phi \ 4 ; \ R \ 4 \ P \ # \ 1 \ { * \ 2 = 44 . }$$

От дикого предка домашние свиньи отличаются редукцией щетинного покрова, большой массой, высокой мясной и сальной продуктаминостью. Свиньи очень плодовиты, в помете бывает до 14 поросят. Бременность длится около 4 мес. На кабанов везде охотятся ради мяса.

Семейство Пекаревые (Tayassuidae). Два вида пекари обитают в Америке и внешне схожи со свиньями.

Семейство Бегемотов (Hippopotamidae) включает два вида: бегемота: обыкновенный и бегемот карликовый. Бегемот обыкновенный — круглое животное, массой до 3 т, с грузным цилиндрическим телом, короткими толстыми ногами и огромной уплощенной головой. Кожа голая. Живут в водоемах Африки. Питаются водной и прибрежной растительностью.

Подотряд Жвачные (Ruminantia). Эти животные характеризуются прежде всего своеобразным процессом пищеварения — наличием жвачки. Желудок сложный, обычно состоящий из четырех отделов. Первые три — рубец, сетка, книжка — являются преджелудком и не имеют железистого эпителия, а четвертый — сычуг — железистый желудок (см. рис. 276). Напервом этапе лишь слегка пережевывают пищу, быстро ее проглатывая. Грубо пережеванная пища сначала попадает в первый отдел — рубец, где под влиянием слюны и деятельности микробов организмов подвергается брожению. Из рубца пища перемещается в сетку с ячеистым строением стенок. Отсюда она отрыгивается обратно в ротовую полость, где повторно подвергается размельчению зубами и обильно смачивается слюной. Образовавшаяся полужидкая масса вновь заглатывается и попадает в третий отдел желудка — книжку, стенки которой образуют параллельные складки наподобие страниц книги. Здесь пища несколько обезвоживается и переходит в последние II отдел желудка — сычуг, где подвергается воздействию желудочного сока

Для жвачных характерно отсутствие резцов в верхней челюсти; и функционально заменяет твердый поперечный валик (рис. 285). Верхних клыков обычно нет, но у безрогих видов, особенно у самцов, они хорошо развиты. Между клыками и коренными зубами имеется диастема. На коренных зубах находятся складки эмали лунчатой формы. Кишечник жвачных очень длинный. Слепая кишка хорошо развита. Молочные железы образуют вымя, расположенное в паутинах самки о двумя—четырьмя сосками. У большинства видов на лобных костях чешуйчатые рога различной формы и строения. Обычно это стройные животные, способные к быстрому бегу

В мировой фауне подотряд жвачных представлен шестью семействами: Оленевые, Кабаро-жьи, Оленевые, Жирафовые, Вилороговые, Полорогие (Бычье).

Семейство Оленевые (Cervidae). Представители этого семейства характеризуются стройным телом с длинными ногами. Самцы, а у северного оленя и самки носят на голове обычно ветвистые костные рога, сменяемые ежегодно. Окраска взрослых, как правило, однотонная, у новорожденных — пятнистая. В России из семейства оленевых обитают лось (самый крупный представитель), северный, благородный, пятнистый олени и косуля. Рога у этих животных ветвятся и состоят из ствола и отростков, число которых увеличивается с возрастом (рис. 286). Самый нижний отросток — первый надглазничный, над ним — второй надглазничный, выше — средний, остальные — венечные. В основании рога имеется роговица, по которой рог отпадает. Зубная формула

$$\frac{-}{1} \frac{1}{*} \frac{3}{P_3} \frac{T}{T_3} - , \frac{3}{T_3} \quad \frac{6-7}{10} \times 2 = 32-34.$$

В Европе основными объектами охоты являются косуля и благородный олень, в России — лось. В северных районах страны и в Сибири разводят одомашненных северных оленей, которых используют как **транспортных** животных; кроме того, от них получают мясо, молоко, меховые и кожевенные шкуры. Разводят пятнистого оленя, родина ко-

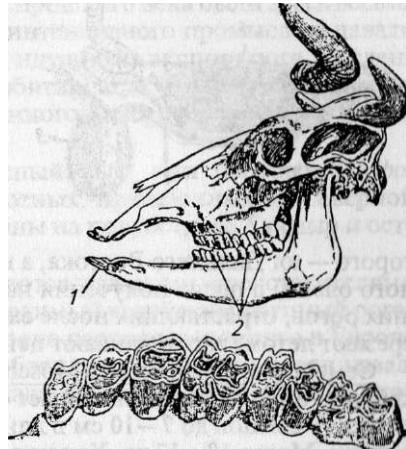


Рис. 285. Череп коровы (внизу — коронки коренных зубов со складками эмали на поверхности):
1 — резцы нижней челюсти; 2 — коренные зубы

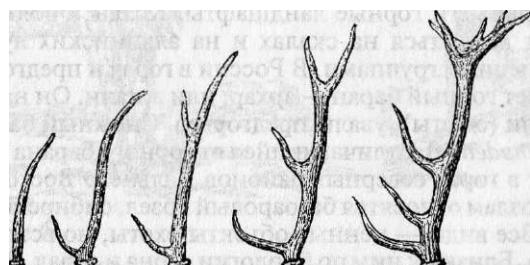


Рис. 286. Разные стадии развития рогов у благородного оленя

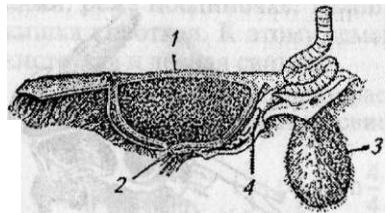


Рис. 287. Мускусный мешок кабарги:
1 — мешок; 2 — его отверстие; 3 — мошонка;
4 — пенис

торого — юг Дальнего Востока, а на Алтае — марала (подвид благородного оленя) в целях получения пантов — молодых, еще не окостеневших рогов, отрастающих после ежегодной смены. Из пантов, которые срезают летом, изготавливают ценное лекарство — пантоクリн.

Семейство Кабарожки (Moschidae) представлено одним видом — кабарга. Внешне она напоминает оленя, но рогов нет, развиты верхние клыки (у самцов до 7—10 см в длину), задние конечности длиннее передних. Масса 10—17 кг. Хорошо развита мускусная железа («кабарожья струя»), секрет которой используют в парфюмерии. Кабаргуразноят, в Китае используют метод «доения» для многократного получения мускуса (рис. 287).

Семейство Половорогие (Bovidae). Самцы, а у многих видов и самки имеют пару неветвящихся прямых или изогнутых рогов. Они образованы костными выростами лобных костей черепа, одетыми роговыми чехлами эпидермального происхождения. Рост рога в отличие от оленя идет от основания. У всех видов они не подвержены смене. Клыков и верхней челюсти нет. Зубная формула у большинства видов

$$1\frac{0}{3}, c\frac{0}{1}, p\frac{3}{3}; t\frac{3}{3} = \frac{6}{10} \times 2 = 32.$$

Наиболее крупным представителем этого семейства является зур (Bison bonasus), почти исчезнувший в начале XX в. В настоящее время его численность заметно выросла. Стоит вопрос о его реакклиматизации в некоторых заповедниках. Несколько видов горных козлов (на нижней стороне морды имеется пучок удлиненных волос — борода) и баранов обитают в пределах России на Кавказе, Алтае и в Сибири. Они населяют открытые и горные ландшафты. Козлы в большей степени предпочитают держаться на скалах и на альпийских лугах. Пасутся и обычно небольшими группами. В России в горах и предгорьях Южной Сибири обитает горный баран — архар, или аргали. Он населяет выемы горные степи (сырты), увалы предгорьев. Снежный баран, или гонсторог (Ovis canadensis), отличающийся от горного барана толстыми мирами, обитает в горах северных районов Дальнего Востока, Якутии и Таймыра. К козлам относятся безоаровый козел, сибирский козел, кашмирский тур. Все виды — ценные объекты охоты, но встречаются достаточно редко. Близки к ним по биологии серна и горал. В степях Нижнего Поволжья и Калмыкии ныне бродят небольшие стада сайгаков!

(*Saiga tatarica*), которые еще в 70-х гг. прошлого века были здесь весьма многочисленны, служили объектом интенсивного промысла и давали превосходное мясо, шкуру и ценные, идущие на экспорт рога. Численность дзерена (*Gazella gutturosa*) — обитателя приграничных районов Алтая, Тувы и Забайкалья и единственного у нас представителя газелей незде крайне низкая.

Вновь появился в Азии мускусный бык, или овцебык (*Ovis moschatus*). Несколько десятков животных, привезенных из Северной Америки, успешно акклиматизированы на полуострове Таймыр и острове Врангеля.

Отдельные виды семейства являются важными представителями домашних животных (рис. 288). Разводимый человеком крупный рогатый скот произошел от когда-то широко распространенного в Европе и Азии быка тура (*Bos taurus*), истребленного несколько веков назад. В горах Алтая можно встретить стада одомашненных быков яков

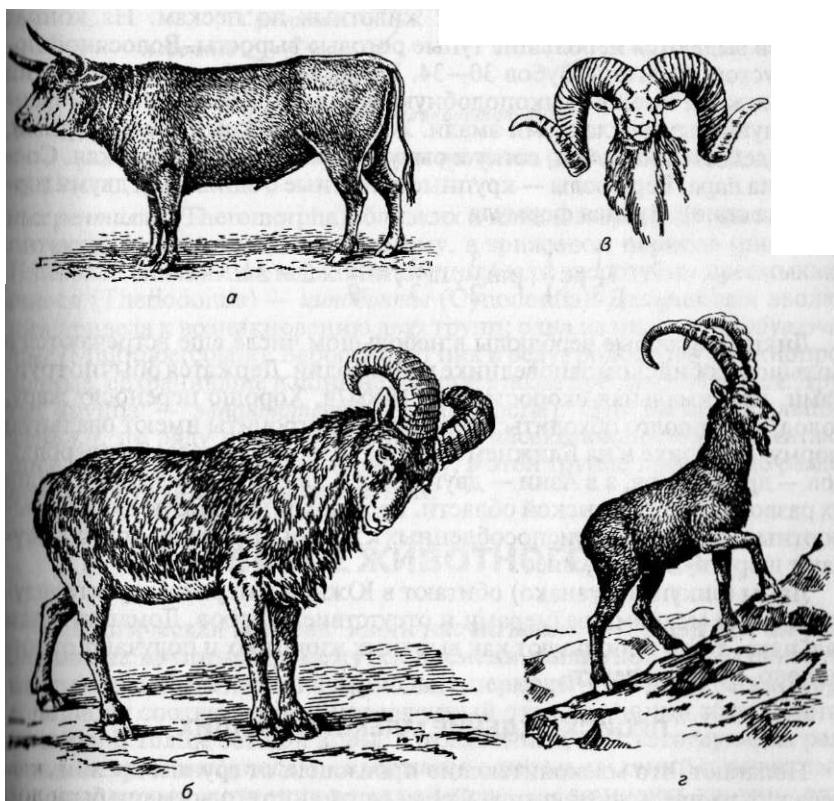


Рис. 288. Предки сельскохозяйственных млекопитающих:
• дикий бык тур (реконструкция); *б* — муфлон; *в* — архар; *г* — безоаровый козел

(*B. mutus*), в диком состоянии здесь уже исчезнувших. Наши домашние овцы ведут свою родословную от горных баранов муфлона (*Ovis musimon*) и архара (*O. ammon*), а козы (*Capra hircus*) — в частности от своеобразного бэзоарового козла (*C. aegagrus*), и сейчас еще встречающегося в горах Кавказа.

В Закавказских республиках разводят также буйволов, которые отличаются от крупного рогатого скота редким волосяным покровом и огромными рогами. Эти животные — одомашненная форма азиатского буйвола (*Bubalus arnee*).

Отряд Мозоленогие (Туиорода). Этот отряд объединяет верблюдом (*•Camelus*) и лам (*Lama*). Размеры средние и крупные, шея длинная, верхняя губа раздвоена, пальцеходящие. На длинных стройных конечностях сохраняются лишь по два пальца — III и IV, а I, II и V редуцированы. Животное опирается не на концы фаланг пальцев, как парнокопытные, а на все суставы пальцев. На нижней поверхности ступни у верблюдов имеются большие эластичные мозолистые подушки, что обеспечивает передвижение этих животных по пескам. На концах пальцев выдаются небольшие тупые роговые выросты. Волосяной покров густой и мягкий. Зубов 30—34. Резцы верхней челюсти и третий резец нижней имеют клыкоподобную форму. Предкоренные и коренные с лунчатыми складками эмали. Желудок сложный трехкамерный, подразделяется на рубец, сетку и съчуг. Слепая кишка короткая. Соеков одна пара. Верблюды — крупные животные с одним или двумя горбами на спине. Зубная формула

$$\frac{i}{3} \frac{i}{1}; c \frac{i}{1} p m^{\wedge} \frac{m}{2} \frac{m}{3} = \frac{5}{9} \times 2 = 34.$$

Дикие двугорбые верблюды в небольшом числе еще встречаются в Большом Гобийском заповеднике в Монголии. Держатся обычно группами, максимальная скорость бега 16 км/ч. Хорошо переносят жару, холод, могут долго обходиться без воды. Эритроциты имеют овальную форму. В Африке и на Ближнем Востоке разводят одногорбых верблюдов — дромадеров, а в Азии — двугорбых (*Camelus bactrianus*); в России их разводят в Астраханской области. Верблюдов используют как трат портных животных, приспособленных к жизни в пустыне, от них получают шерсть, молоко, мясо.

Ламы (викунья, гуанако) обитают в Южной Америке, характеризуются более мелкими размерами и отсутствием горбов. Домашних лам (альпака, лама) используют как выночных животных и получают от них молоко, шкуры, шерсть.

ПРОИСХОЖДЕНИЕ МЛЕКОПИТАЮЩИХ

Полагают, что млекопитающие произошли от группы древних гемлеозийских пресмыкающихся. Среди рептилий того времени было и несколько групп, носивших отдельные признаки, характерные для млекопитающих. В пермский период сформировалась группа зверопос



Рис. 289. Родословное дерево млекопитающих (сокращенная схема)

ныхрептилий (*Theromorpha*), близких к котилозаврам. Первые млекопитающие появились, по-видимому, в триасовом периоде (рис. 289). Наиболее близкими к млекопитающим были зверозубые пресмыкающиеся (*Theriodontia*) — цинодонты (*Cynodontia*). Дальнейшая эволюция привела к возникновению двух групп; одна из них — многобугорчатые (*Multituberculata*). Вероятно, от них и ведут родословную однопроходные, сохранившие и поньне ряд признаков пресмыкающихся. Другая группа — эвантофтерии (*Eupantotheria*). Это были небольшие зверьки, по ряду признаков близкие к насекомоядным млекопитающим. В конце мезозоя, по-видимому, в этой группе произошло разделение на сумчатых и плацентарных.

РАЗВИТИЕ ЖИВОТНОГО МИРА

Геологическая история Земли насчитывает 4—7 млрд лет. Ее разделяют на крупные промежутки времени различной продолжительности — эры; эры делят на периоды, а периоды — на века. Каждой эре и периоду соответствует определенный этап эволюции живых организмов. В толще земной коры, в отложениях, соответствующих различным эрам и периодам, содержатся остатки растений и животных. Их можно, хотя и не в полной мере, восстановить картину развития растительного и животного мира отдельных эр и периодов в истории нашей планеты.

Архейская эра (от греч. archaios — древнейший) — древнейшая эра, охватывающая наиболее ранние этапы развития Земли. Вероятно, в начале этой эры произошло зарождение жизни. Возраст древнейших обнаруженных остатков бактериоподобных организмов оценивается примерно в 3,5 млрд лет. Многие исследователи полагают, что жизнь зародилась на морском мелководье. Первые организмы были гетеротрофами. К сожалению, изучение первых этапов развития жизни затруднено из-за давности того периода. Первые живые организмы были одноклеточными, практически не оставившими ископаемых остатком.

Протерозойская эра (от греч. proteros — более ранний) — эра возникновения примитивной жизни. В отложениях обнаружены остатки бактерий, простейших, водорослей. Появились первые многоклеточные организмы (губки).

Палеозойская эра (от греч. palaios — древний) — эра древней жизни, характеризующаяся формированием всех типов растений и животных. В эту эру как растительный, так и животный мир достигают большого разнообразия. В начале эры жизнь сосредоточивалась только в море. Животные были представлены практически всеми известными типами беспозвоночных, а растения — преимущественно разными водорослями. Суша представляла собой единый материк — Пангею, который со временем разделился на части. Заселение суши растениями и животными началось в силуре—девоне. Первыми наземными растениями были псилофиты, которые постепенно вытеснялись папоротникообразными (хвоши, плауны, папоротники), в том числе и древовидными формами. В конце палеозоя появляются голосеменные растения. Первыми наземными животными были представители паукообразных, многоножек и древних насекомых, а из позвоночных — земноводные 11 позже — примитивные рептилии. В карбоне появились первые летающие насекомые — стрекозы. В море господствовали своеобразные членистоногие — трилобиты, процветали различные представители кишечнополостных (кораллы), иглокожие, моллюски. Начинается расцвет и позвоночных животных. В ордовике появляются первые бесчленистоногие (предки нынешних миног и миксин), от которых отделились челюстноротые — первые рыбы. Древнейшие рыбы — плакодермы (Placodermi), предки хрящевых рыб (акул, скатов и химер) известны с силура, позже появляются двоякодышащие и кистеперые, а затем и костистые рыбы.

Мезозойская эра (от греч. mēsos — средний) — эра средней жизни, характеризующаяся развитием пресмыкающихся, первых млекопитающих и птиц. Эту эру нередко называют веком пресмыкающихся, которые господствовали не только на суше, но и в воздухе, и в море. Поражает разнообразие форм мезозойских пресмыкающихся, их нередко гигантские размеры. Наиболее замечательной группой были динозавры. Мезозойская фауна и флора были значительно разнообразнее и вышеступали в значительно обновленном составе. Появляются бесхвостые и хвостатые земноводные, а также первые млекопитающие и птицы, произошедшие от пресмыкающихся. В морях доминировали головоногие

(аммониты, белемниты) и рыбы. Широко были представлены шестилучевые кораллы, моллюски, иглокожие, из одноклеточных — фораминиферы. На протяжении мезозоя постепенно изменялся и мир растений. Папоротникообразных сменили голосеменные, занявшие господствующее положение. В конце эры появились первые цветковые, тогда как голосеменные постепенно утратили свое положение.

Кайнозойская эра (от греч. *kainos* — новый) — эра новой жизни. Она продолжается и в настоящее время. В эту эру флора и фауна Земли постепенно приобретают современный вид. В растительном мире начинается расцвет покрытосеменных (цветковых). На границе эр вымирают характерные для мезозоя головоногие моллюски — аммониты, белемниты и многие другие группы беспозвоночных. Резко сократилось число групп древних костных рыб, стремительно развиваются костистые рыбы. Исчезли динозавры и большинство групп мезозойских рептилий, утративших господствующее положение среди позвоночных. Начинается расцвет млекопитающих и птиц, а среди беспозвоночных — насекомых, которые, эволюционируя, дают большое разнообразие форм. Два-три миллиона лет назад развитие животного мира ознаменовалось появлением разумного существа — человека.

Происхождение и эволюция животных. Современные животные — это результат эволюционной истории. Живая природа находится в непрерывном движении, она постоянно меняется и развивается. Многочисленные данные, полученные в результате сравнительного изучения различных живых организмов и ископаемых находок, позволяют проследить филогенез той или иной группы животных. Вместе с тем следует помнить, что возникновение всех групп организмов отделено от современности сроком в десятки и сотни миллионов лет (для типов беспозвоночных этот срок составляет минимум 500—600 млн лет) и за этот период могли произойти различные изменения. Животные, испытывая постоянное давление отбора, находили сходные пути решения возникающих проблем. Поэтому возможно, что различные уровни структурной организации достигались и в результате конвергенции.

Всесторонние исследования убеждают в последовательном усложнении организации животных в процессе эволюции. Поэтому филогenetические отношения животных разных систематических групп могут быть изображены в виде разветвленного древа, в основании которого находятся примитивные одноклеточные животные.

Сравнительно-морфологический метод изучения позволяет обнаружить те или иные признаки сходства и различия в строении разных групп животных, что помогает составить представление о возможных переходных формах и их гипотетических предках.

Использование данных сравнительной эмбриологии основывается па биогенетическом законе (правиле), согласно которому онтогенез является кратким повторением филогенеза, т. е. прошлое не проходит в эволюции животных бесследно и животные в своем индивидуальном развитии повторяют в той или иной степени этапы исторического развития вида. Это подтверждают многочисленные данные, полученные

корифеями нашей отечественной науки А. О. Ковалевским и И. И. Мечниковым в области эмбриологии беспозвоночных и низших хордовых и К. М. Бэрром — в области эмбрионального развития позвоночных.

Палеонтологические данные также дают возможность устанавливать родственные связи животных на основе изучения ископаемых остатков вымерших животных. В последние годы к этому добавились генетические данные и данные молекулярной биологии.

Первые организмы обладали неклеточным строением; на следующем этапе возникли клеточные формы жизни — прокариоты. От них произошли одноклеточные ядерные организмы — эукариоты, давшие начало саркодовым и жгутиковым.

Первые живые организмы были гетеротрофными, позже появляются автотрофные, фотосинтезирующие организмы. Зеленые жгутиконосцы, соединяющие в строении и биологии черты животных и растений, свидетельствуют о происхождении двух царств от общих предков.

У основания родословного древа животных находятся одноклеточные организмы. Эволюция простейших (Protozoa) протекала в разных направлениях, но одноклеточная организация тела у большинства сохранилась. Древние саркомастигофоры дали начало саркодовым и паразитическим микроспоридиям, а также миксоспоридиям. Другая ветвь родословного древа ведет к жгутиконосцам и далее к инфузориям (простейшим наиболее сложного строения) и к паразитическим споровикам.

Существует несколько гипотез о происхождении многоклеточных. Большинство специалистов считают, что многоклеточные животные произошли от незеленых колониальных жгутиконосцев. Гипотетическим предком многоклеточных (по Мечникову) была фагоцителла. По-видимому, возникновение многоклеточных организмов произошло не однажды. Разные колониальные виды одноклеточных через фагоцителлоподобных потомков дали начало трем ветвям современных многоклеточных животных — губкам, кишечнополостным, гребневикам и остальным многоклеточным.

Вероятно, от воротничковых жгутиковых произошли губки, неподвижный образ жизни которых не способствовал их прогрессу, поэтому они сохранили много примитивных черт строения. Позднее от других колониальных жгутиковых, возможно, произошли двухслойные предки кишечнополостных и гребневиков. Большинство их перешло к неподвижному или малоподвижному существованию, как это наблюдается у современных кишечнополостных. Кишечнополостные имеют клеточные структуры различной сложности и назначения, которые помогают представить этапы развития третьего зародышевого листка мезодермы. Зачатки мезодермы, имеющиеся у кишечнополостных еще более отчетливо выражены у гребневиков, животных по происхождению и организации близких к кишечнополостным.

Следующий этап в развитии жизни — появление билатерально симметричных трехслойных животных. Большое прогрессивное значение имело возникновение третьего зародышевого листка — мезодермы — к формированию его производных, впервые появляется выделительмин

система. Представители этой группы ведут активный, подвижный образ жизни; в связи с поступательным движением происходит обособление головного отдела и концентрация органов чувств на нем. К современным представителям этой группы относятся плоские черви, среди которых наиболее примитивными принято считать бескишечных турбеллярий, возможно, аналогичных предковым формам.

В родстве с плоскими червями состоят круглые черви и немертины. Тип Круглые черви включает ряд классов, представители которых имеют как во взрослом состоянии, так и в период эмбрионального развития ряд сходных черт строения с ресничными плоскими червями (наличие паренхимы, покровы с ресничным эпителием, протонефридии и др.).

От турбелляриеподобных предков, возможно, отделились вторичнополостные животные, которых с полным основанием называют высшими многоклеточными животными. Основанием к этому служат такие черты их организации, как концентрация нервных клеток в ганглиях, образование кровеносной системы, выделительные органы метанефридиального типа, сегментированность тела. Образование целома обеспечило гомеостаз внутренней среды организма и автономизацию жизненных процессов.

Одну из ветвей вторичнополостных животных составляют трохофорные организмы, которые отличаются некоторыми особенностями эмбриогенеза и формированием личинки — трохофоры или близкой к ней. Раньше других от трохофорных отделилась ветвь, приведшая к возникновению моллюсков. Несмотря на то что моллюски имеют на всех этапах онтогенеза несегментированное тело, у них имеются нервные ганглии, кровеносная система, метанефридиальные выделительные органы. Кроме того, типичная для морских моллюсков личинка — парусник — сходна с личинкой многощетинковых кольчатых червей — трохофорой.

Сравнение кольчатых червей с членистоногими убеждает в большом сходстве их строения. И для тех, и для других характерны вторичная полость тела, метамерность, нервная система в виде надглоточных ганглиев, окологлоточного кольца и брюшной нервной цепочки, посегментное расположение конечностей, которые у многощетинковых кольчатых червей имеют вид параподий, состоящих из двух лопастей, а у первичноводных членистоногих можно проследить наличие двусторонности их строения. Общие черты строения кольчатых червей и членистоногих указывают на близкое родство этих двух групп. Членистоногие — прогрессивная ветвь, берущая начало от общего предка с современными многощетинковыми кольчатыми червями.

Другая ветвь родословного древа животного мира ведет от древних вторичнополостных червеобразных животных к вторичноротым животным — иглокожим, полуходовым и хордовым.

Хордовые животные — одна из наиболее высокоразвитых групп животного мира. Происхождение хордовых в деталях остается еще неясным. Первичные хордовые, как показали исследования А. Н. Северцова, представляли собой существа, довольно близкие к современному ланцетнику.

Это были вторичнополостные, вторичноротые животные с признаками внутренней сегментации, имевшие хорду, нервную трубку вдоль спины и жаберные щели в стенках глотки. От этих примитивных первичных хордовых произошли, с одной стороны, современные бесчелюстные (ланцетники и близкие к ним формы), а с другой — первичные позвоночные. Боковую ветвь низших хордовых образуют личноногие хордовые, в организации взрослых особей которых произошли регressive изменения в связи с переходом к сидячему образу жизни при пассивном питании.

В эволюции позвоночных одна из ветвей привела к появлению примитивных бесчелюстных (предков миног и миксин), которые дали начало челюстноротым и, соответственно, всем остальным классам *Vertebrata*. У представителей данной эволюционной ветви в связи с активным питанием и общим повышением поведенческой активности развились подвижные челюсти, парные конечности, сложный скелет и ряд других усложнений в строении. Все это обеспечивало им развитие и широкое распространение. Первыми представителями челюстноротых позвоночных были древние рыбы. От древних палеозойских рыб ведут родословную кистеперые рыбы. Их и принято считать родоначальниками наземных позвоночных, первыми из которых были примитивные амфибии. От древних примитивных амфибий произошли как земноводные, так и первые палеозойские пресмыкающиеся, достигшие в мезозойскую эру большого разнообразия форм.

В мезозое от пресмыкающихся отделились два высших класса позвоночных — млекопитающие и птицы. Млекопитающие произошли от древних рептилий в самом начале мезозойской эры, раньше птиц. Но их бурное развитие, приведшее к современному богатству форм, отмечается, как и у птиц, в кайнозойской эре. В юрских отложениях найдены остатки своеобразных существ — археоптериксов, которые со вмешанием признаками рептилий и птиц. Настоящие птицы появились позже и достигли большого разнообразия в кайнозойскую эру.

ОСНОВЫ ЭКОЛОГИИ ЖИВОТНЫХ

Экология — наука, изучающая взаимоотношения живых организмов между собой и с окружающей средой; цель экологии — выявить законы, управляющие этими отношениями. Экология животных — быстро развивающаяся отрасль зоологии и экологии. Этот раздел экологии изучает образ жизни животных и влияние различных факторов среды на основные аспекты жизнедеятельности животных: питание, размножение, выживание, изменение численности и т. д.

Экологические исследования ведутся в нескольких направлениях:

- 1) изучение взаимоотношений и взаимосвязей животных с окружающей средой;
- 2) изучение формирования, структуры и динамики популяций различных видов животных;

3) изучение роли животных в природных сообществах — биогеоценозах (экосистемах).

Разработка этих проблем представляет большой теоретический интерес и имеет существенное практическое значение. Данные экологии животных широко используются другими отраслями зоологической науки. Тесная связь существует между экологией и физиологией животных, поскольку на многие физиологические процессы оказывают огромное влияние различные экологические факторы. Морфологические особенности животных — следствие адаптации животных к различным факторам внешней среды. Экология служит основой зоогеографии, поскольку условия среды наравне с историческими причинами определяют распространение животных и пути формирования фауны. Наконец, современная систематика, выделяя отдельные виды животных, учитывает и экологические различия.

Велико также и прикладное значение экологии животных. Это подтверждается хотя бы тем, что в последнее время получила развитие особых отрасль — экология сельскохозяйственных животных. Данные этой отрасли экологии необходимо учитывать, в частности и при районировании пород сельскохозяйственных животных, поскольку каждая порода проявляет продуктивные качества только в определенных условиях среды; при осуществлении мероприятий по борьбе с паразитами и вредителями, а также мероприятий по увеличению численности различных видов животных. Ведь одним из основных методов борьбы с паразитами является создание условий, неблагоприятных для их жизни и размножения; напротив, при осуществлении мероприятий по увеличению численности тех или иных животных необходимо создавать условия, позволяющие им выживать и успешно размножаться.

Регулирование и управление численностью различных видов паразитов и вредителей сельскохозяйственного производства, переносчиков заболеваний, промысловых синантропных и других животных возможно только в том случае, если опираться на знание их образа жизни, характера питания, особенностей размножения, т. е. на знание зависимости их биологии от внешних факторов. Важность экологических исследований возрастает в связи с глобальными изменениями биосферы Земли в результате хозяйственной деятельности человека. Стоит вопрос не просто о сохранении биоразнообразия, а о перспективе сохранения жизни и человека как вида на нашей планете.

Взаимоотношения животных с окружающей средой. Средой именуется все то, что окружает животное в течение жизни и прямо или косвенно влияет на его жизнедеятельность. Условия среды, на которые животное реагирует приспособительными реакциями и которые определяют условия существования организма, называются *экологическими факторами*. В совокупности экологические факторы образуют среду обитания, а самые важные и незаменимые — условия существования. Для рыб — это вода. Для дрозофилы, в частности, температура: при 19 °C она живет в среднем 12 дней, а при 34 °C — только 6. Некоторые факторы оказывают прямое воздействие на животное, влияя на его по-

ведение, интенсивность протекающих в организме жизненных процессов, а иногда вызывая его гибель. При температуре 10 °С куколка бабочки-капустницы потребляет около 100 мм³ кислорода на 1 г массы, а при 20 °С уже примерно 250 мм³. Другие факторы влияют на животных косвенно. Так, например, количество гумуса в почве на наземных и древесных животных прямого воздействия практически не оказывает. Но этот фактор влияет на растительность, а через нее и на обитающих в данном месте животных.

Факторы среды могут быть подразделены на стабильные и изменяющиеся во времени. Первые неизменны в течение длительного времени, и поэтому их действие на животных постоянно и стабильно. Таковы, например, сила тяжести, солнечная радиация, химический состав и свойства атмосферы и др. Вторые факторы изменяются либо закономерно, либо случайно. К закономерно меняющимся факторам относятся смена дня и ночи, смена сезонов на протяжении года, приливы и отливы, изменения длины светового дня. К факторам, не подчиняющимся закономерности, следует относить силу и направление ветра, осадки, антропогенные воздействия. Периодически изменяющиеся факторы среды нередко обуславливают появление у животных биологических ритмов — периодически повторяющихся изменений интенсивности и характера биологических процессов и явлений. В той или иной форме биологические ритмы присущи всем живым организмам. Существуют, например, околосуточные, или циркадианые, биологические ритмы с периодом от 20 до 28 ч. Околосуточный биологический цикл выражается в смене в течение суток периода бодрствования периодом отдыха и сна. Околосуточные ритмы характерны для большинства биохимических, цитологических, физиологических и психологических процессов. У человека примерно 100 физиологических функций подчинены околосуточным ритмам. Существуют также лунные ритмы, соответствующие по циклу fazам Луны (29,53 сут) или лунным суткам (24,8 ч). С вращением Луны связаны приливные ритмы, равные циклам морских приливов (24,8 или 12,4 ч). Лунные и приливные ритмы характерны для многих морских животных. Приспособливаясь к сезонным изменениям природы, животные выработали ясно выраженные сезонные биологические ритмы. Например, весной у многих животных наступает сезон размножения. В это же время происходит весенняя линька. Особенно хорошо сезонность жизненного цикла выражена у перелетных птиц. Ориентироваться во времени животным помогает особый механизм — биологические часы, в основе которых лежит строгая периодичность протекающих в клетках физико-химических процессов. В сезонных перестройках жизнедеятельности у большинства видов ой ромную роль играет *фотопериодизм*. Так называют реакцию организмом на соотношение светлого (день) и темного (ночь) периодов суток.

И в медицине, и в сельском хозяйстве, и в биотехнологиях необходимо учитывать внутреннюю ритмику организмов.

Несмотря на то что средовых факторов существует великое множество, а представители разных видов реагируют на каждый из них

по-разному, можно выявить ряд общих законов их действия на организмы. Главный из них — *закон оптимума*. Экологические факторы бывают разной интенсивности. Температура среды, например, может быть высокой, средней или низкой. Диапазон значений того или иного фактора, в пределах которого жизнеспособность организмов возрастает, называется зоной оптимума. Оптимум не одинаков для разных видов животных (холодостойких и теплолюбивых, влагостойких и сухолюбивых, теневыносливых и светолюбивых и пр.). Отклонения от оптимума как в сторону понижения, так и в сторону повышения интенсивности фактора вызывают угнетенное состояние организма — зона *пессимума* — (бывают верхние и нижние границы выносливости по каждому фактору). Если действие фактора выходит за определенные возможные для вида пределы, то организм погибает. Губительные значения фактора называют критическими точками.

Нужно помнить, что факторы среды действуют на животное не каждый по отдельности, а в совокупности. На организм одновременно действуют не один, а несколько факторов (их комплекс). Например, влияние температурного фактора на жизнедеятельность насекомых оказывается различным в зависимости от влажности окружающего воздуха. Значение снежного покрова в жизни мелких грызунов меняется в районах, где он сильно уплотнен действием постоянных ветров и где он остается рыхлым, без наста. Во многих случаях воздействие того или иного фактора среды ограничивает возможность существования данного животного. Наличие или отсутствие молочая определяет распространение молочайного бражника (питающегося им на стадии гусеницы).

Если какой-либо фактор выходит за пределы выносливости организма, то существование этого организма становится невозможным даже при других благоприятных условиях. Им оказывается тот фактор среды, который сильнее всего отклоняется от оптимума. Факторы, выходящие за пределы максимума или минимума выносливости, называются *лимитирующими факторами*. Недостаточно теплый климат России ограничивает количество видов и численность, например, амфибий и рептилий.

Некоторые животные могут жить и размножаться в условиях весьма изменчивой среды, другие приспособлены к строго определенным, узким колебаниям значений ее факторов. Степень приспособленности вида к условиям среды называется *экологической валентностью*.

Потребность животного в определенных условиях существования часто резко меняется с его возрастом. Взрослый карп может жить в водоемах, замерзающих на зиму, но для развития его икры благоприятна температура воды выше 18 °С. Важно подчеркнуть, что факторами среды в значительной степени определяется численность животных данного вида, поскольку они обуславливают степень плодовитости и смертности.

Другая возможность влиять на жизнедеятельность интересующих нас видов вытекает из экологического *принципа совместного действия факторов*. Он заключается в том, что результат влияния любого эколо-

гического фактора на организм во многом зависит от того, в какой комбинации и с какой силой действуют в данный момент и другие факторы.

Факторы среды и их влияние на животных. Факторы среды крайне разнообразны. Различают *абиотические*, или факторы неживой природы; *биотические*, или взаимные влияния организмов друг на друга и *антропогенные*, или все виды воздействия человека на природу.

Абиотические факторы среды — компоненты и явления неживой, неорганической природы, прямо или косвенно воздействующие на живые организмы. Они весьма различны для животных, обитающих в воздушной и водной средах. К этой группе факторов относятся климатические или атмосферные (свет, температура, влажность, осадки, ветер), почвенно-грунтовые, рельеф местности.

Климатические факторы играют важную роль в жизни наземных животных. Особенno значительное воздействие на различных животных оказывают свет, температура воздуха, влажность, осадки, ветер и др.

Свет играет разную роль в жизни животных. Для животных он важен не как источник энергии, а как условие для ориентации в пространстве.

Температура оказывает различное влияние на животных, и в первую очередь на животных с непостоянной температурой тела, или *пойкилотермных*; она определяет возможность их жизни. Так, например, при температуре выше 45—50 °C большинство насекомых погибает от перегрева. У животных с непостоянной температурой тела физиологический оптимум лежит в пределах 25—40 °C. При снижении температуры все физиологические процессы замедляются. Лягушка при температуре воздуха 16 °C еще активно прыгает, охотится, реагирует на приближение опасности, при 12 °C она может только медленно ползать, при 4 °C ее движения почти прекращаются и становятся некоординированными. Сезонные изменения температуры среды вызывают у животных адаптивные изменения механизмов терморегуляции.

Терморегуляция — совокупность физиологических процессов в организме животных, направленных на поддержание температуры тела. Только две группы живых существ — птицы и млекопитающие (*гомоотермные*) способны поддерживать постоянную температуру тела, независимо от температуры окружающей среды. Обмен веществ у этих животных идет с постоянно высокой скоростью. Разница температуры их тела и температуры окружающей среды может составлять почти 90 °C. Например, песцы при температуре тела 38,3 °C могут выдерживать пятидесятиградусные морозы.

Влажность воздуха влияет на степень испарения воды животным и выведения ее органами выделения. У многих животных отмечено прямое воздействие влажности среды на разные проявления их жизнедеятельности. У бобовой зерновки развитие эмбриона в яйце при относительной влажности воздуха 24 % длится 4 дня, а при 100 % — уже 6 дней. Мучной клещ хорошо развивается при влажности зерна выше 13 %, плохо — при содержании влаги в нем около 12,5 % и гибнет, когда влажность пищи менее 12 %. Дожди влияют на животных как непо-

средственно, так и косвенно — через урожай растительных кормов. В жизни животных зон умеренного и холодного климата большое значение имеет снежный покров — его продолжительность, глубина, плотность и другие свойства.

Недостаток осадков часто приводит к пересыханию водоемов, что резко ухудшает условия жизни водных и полуводных животных. Из физико-химических свойств воды особо важны для существования различных животных содержание в воде газов и солей, а также ее температура и прозрачность (светопроницаемость). Например, форель лучше всего чувствует себя в реках и озерах, вода которых содержит около 7 мл кислорода в 1 л, а карп благополучно живет и размножается в воде, содержащей 3—4 мл кислорода в 1 л.

Прозрачность воды оказывает на водных животных как косвенное, так и прямое воздействие. Верхняя зона морей особенно богата разнообразными организмами. В глубинной зоне условия жизни настолько своеобразны, что в ней сложилась совершенно особая фауна. Здесь царит вечная темнота, температура воды низкая, держится на уровне 1 - 3 °C.

Почвенно-грунтовые факторы — это различные физические и химические свойства почв. В строении почвенных животных обычно обнаруживается много адаптаций к подземному образу жизни. Например, у крота тело имеет цилиндрическую форму, удобную для его продвижения по округлым в сечении норкам. Его передние конечности превратились в органы рытья. У пустынных тушканчиков на подошвах лап имеется густая щетка упругих волос, которая более чем вдвое увеличивает площадь опоры зверька, что очень важно при передвижении прыжками по сыпучим пескам.

Не только характер почвы влияет на жизнь животных, но и животные оказывают влияние на физическую структуру и химический состав почвы. Примером могут служить дождевые черви.

Рельеф земной поверхности оказывает влияние на строение, распространение и жизнь животных. У горных животных обычно имеются приспособления к жизни среди скал и камней. У горных козлов, например, копыта малы и заострены.

Радиоактивное излучение — один из наиболее опасных экологических факторов, так как вызывает нарушения наследственного аппарата клеток. Среди одноклеточных ряд инфузорий способны выдержать без последствий такие дозы излучения, которые в тысячи раз превышают смертельные дозы для человека.

К числу абиотических факторов относятся также содержание кислорода, солевой состав воды и почвы, атмосферное электричество, ветры, течения и т. п. Каждый фактор так или иначе влияет на жизнь животных. Однако в тех пределах, в которых они действуют, эти факторы не ограничивают жизнь на Земле.

Биотическими факторами называются прямые и косвенные воздействия на организмы других живых организмов. Значение этих факторов в

жизни животных велико, поскольку они постоянно находятся в разнообразных взаимоотношениях с различными элементами живой природы.

Ни один вид, ни один организм не могут существовать без других организмов. Вся живая природа переплетена сложной системой связей, от которых зависят возможности питания, размножения, распространения видов и многие свойства их местообитаний. Зависимость организмов друг от друга чрезвычайно разнообразна. Некоторые формы этой зависимости играют в природе особенно важную роль.

Прежде всего, практически любой организм или его останки служат пищей другим животным. Растительноядные животные — фитофаги — питаются растениями, но в то же время сами служат пищей плотоядным зоофагам и всеядным полифагам.

Способы питания одних организмов другими разделяют обычно на хищничество, паразитизм, собирательство и пастьбу. Эти способы различаются по затратам времени и энергии на добывание пищи. Типичные хищники тратят много сил на поиск и овладение живой добычей. Собиратели тратят энергию в основном на поиск и сбор пищи. Паразиты живут в условиях избытка пищевых ресурсов, используя хозяина и как место обитания. Пасущиеся животные пытаются подножным кормом, который обычно бывает в изобилии, его не приходится долго искать, и он легкодоступен. Своеобразными собирателями являются фильтраторы и грунтоеды в водоемах и почвах.

Различные животные вступают с растениями и другими животными в сложные взаимоотношения также при поисках убежищ и укрытий. Дятел долбит дупло в дереве для устройства гнезда. Шимпанзе перед сном ломает ветви, чтобы сделать себе ночное убежище. Лебеди вступают в битву друг с другом из-за открытых мест для гнезда.

Все это говорит о большой сложности взаимоотношений животного с окружающими его растениями и другими животными.

Взаимоотношения животных подразделяют на межвидовые и внутривидовые. Межвидовыми называются взаимоотношения особей, принадлежащих к различным видам. Формы таких взаимоотношений крайне разнообразны. Обычно различают симбиотические, асимибийтические и нейтральные взаимоотношения. Эти связи пронизывают всю природу. Без них, как и без трофических связей, невозможно формирование устойчивых сообществ.

Симбиотические отношения — это совместное существование животных разных видов; по характеру отношений между партнерами выделяют несколько типов симбиоза: мутуализм, комменсализм, квартирантство.

Мутуализмом называют совместное обитание двух или более видов животных, при котором все они приносят друг другу какую-либо пользу. Таковы, например, взаимоотношения некоторых муравьев с тлями или актиний и раков-отшельников. Это взаимовыгодные отношения, при которых совместное существование видов повышает выживаемость каждого из них. Взаимоотношения жвачных копытных и их же

лудочной микрофлоры — широко известный пример таких взаимовыгодных связей.

Комменсализм — форма отношений, выгодная для одного из партнеров и безразличная для другого; в этом случае один или несколько животных пытаются за счет другого, не причиняя ему, однако, вреда. Это может быть так называемое нахлебничество — питание остатками пищи другого вида, использование его выделений. Так, стервятники и гиены поедают остатки пищи львов и других крупных хищников. Одностороннюю выгоду получают некоторые виды, используя других для расселения. Например, мелкие клещи, которые пытаются в разлагающихся материалах, расселяются жуками или мухами, служащими клещам в качестве живого транспорта.

Квартирантство — совместное обитание в норах, гнездах или обитание одного животного на теле или внутри тела другого без вреда для хозяина. Рыба-прилипало обладает мощной присоской, с помощью которой она присасывается к крупным рыбам и перемещается на большие расстояния.

А симбиотические отношения (хищничество, паразитизм и конкуренция) — взаимоотношения двух или более видов животных, зачастую приводящие к гибели одного из них, или к различным отрицательным для него последствиям.

Хищничество. Этот тип отношений предполагает наличие хищника и жертвы. Например, во многих случаях численность оленей в каком-либо районе определяется численностью волков. Но нельзя забывать, что от хищников в первую очередь гибнут слабые, больные животные. Отношения хищник—жертва являются важным фактором естественного отбора для обоих видов. Выживают те жертвы, которые быстро реагируют на опасность, лучше бегают или летают, имеют зоркие глаза, чуткие уши, более развитую нервную систему. Хищники, таким образом, ведут отбор на прогрессивную эволюцию жертв и сами эволюционируют в этом направлении. Оставляют потомство те, кто более успешен в ловле добычи, более силен и вынослив, более сообразителен и т. п.

Паразитизм — взаимосвязь животных двух видов, когда один организм — паразит — живет в теле или на поверхности тела другого — хозяина, питаясь его тканями и соками или отнимая его пищу (кишечные паразиты) и этим нанося ему вред. Например, у людей обнаружено около 65 видов плоских и до 70 видов круглых паразитических червей. У крупного рогатого скота их около 60 видов, у овец примерно 100 видов. Уже не одно тысячелетие рядом с человеком соседствует прирученная им собака; это привело к обмену их паразитами: из 125 видов червей, паразитирующих в собаках, 36 паразитируют также и у человека.

Длительное существование с паразитом приводит к выработке организмом хозяина внутренних защитных средств, снижающих восприимчивость его к паразитарному заболеванию (неполный иммунитет). Молодые организмы обычно менее устойчивы и тяжелее переносят заболевание.

Конкуренция — этот тип отношений возникает в том случае, когда особи одного или разных видов существуют за счет общего ресурса, которого всем не хватает. В качестве ресурсов выступают факторы среды, используемые для жизнедеятельности, и их запасы в результате этого уменьшаются (пища, вода, убежища, удобные места для размножения, запасы минеральных солей и т. п.).

Ресурсы, необходимые для жизни, в природе почти всегда ограничены. Если вид встречает в своем местообитании конкурента, ему достается меньше ресурсов, и это отражается на возможностях размножения и на численности популяций. Поэтому конкуренция неблагоприятна для обоих взаимодействующих видов. Каждый из них выиграл бы в отсутствии другого. Конкуренция в мире животных особенно остро проявляется в борьбе различных видов за пищу, места для гнездования, убежища, за пространство для индивидуального участка.

Нейтраллизм. В этом случае представители разных видов, обитающие вместе, непосредственно не влияют друг на друга. Они связаны косвенными отношениями, через цепочку других видов. Например, дятлы и лесные мыши не контактируют между собой, но зависят от состояния древостоя и всех тех условий, которые создает им лесная растительность.

Существуют и другие формы зависимости организмов друг от друга.

Антропогенные факторы — это различные воздействия на животных хозяйственной деятельности человека (вырубка лесов, осушение болот, возведение плотин, выбросы в атмосферу различных химических веществ и пр.).

Четкого определения этих факторов нет, это все формы деятельности человека, которые каким-либо образом влияют на животных. Диапазон воздействия человека на животных очень велик; эти воздействия могут проявляться не сразу, а через большой промежуток времени, даже через десятки лет. При этом влияние человека на природу и на животных не однозначно. Для некоторых видов изменения ландшафтом под влиянием деятельности человека весьма благоприятны, некоторые виды животных становятся синантропными, других человек сознательно расселяет и способствует увеличению их численности. Но порой деятельность человека ухудшает условия жизни животных, а иногда приводит даже к вымиранию. Распашка целинных степей повлекла в ряде районов исчезновение таких типично степных животных, как байбак, дрофа, стрепет, но способствовала расселению и размножению многих вредителей полевых культур.

В настоящее время облик нашей планеты меняется коренным образом, и ключевыми в таком изменении оказываются антропогенные факторы. Различают два типа антропогенных факторов. Первые оказывают прямое воздействие на животных, когда деятельность человека приводит к гибели или изъятию животного (добыча, целенаправленное уничтожение вредителей сельского, лесного хозяйства и попутное — под колесами автомобилей). Вторые оказывают косвенное влияние (вытеснение животных, изменение и уничтожение их среды обитания)

Губительно действуют на животных лесные пожары, загрязнения водоемов отходами промышленного производства и сточными водами, химическая обработка сельскохозяйственных угодий и лесов, кислотные дожди и другие изменения среды.

Популяции животных и их структура. Реально существующая систематическая единица — вид. Все виды живых организмов в природе представлены популяциями. Разные популяции одного вида связаны между собой либо постоянно, либо периодически в результате перемещения отдельных особей.

Популяция — исторически сложившаяся естественная группа особей одного вида, обладающая общим генофондом, занимающая определенную территорию и приспособленная к условиям этой территории (биотопа). Любой вид состоит из популяций, потому что территория, на которой он обитает, неоднородна по условиям, и вид распределен на ней неравномерно. Осваивая подходящую территорию и размножаясь на ней, члены популяции вступают друг с другом в разнообразные отношения. В популяциях разных видов проявляются все формы биотических связей, известные для межвидовых отношений.

Все основные характеристики популяций — количественные. Главная из них — *численность*. Она сразу показывает, благоприятны или нет условия существования вида на данной территории. Абсолютное число особей в каждой популяции сосчитать чаще всего бывает нелегко (например, число всех мышей на большом поле или окуней в озере), поэтому обычно используют другой показатель — *плотность популяции*. Он отражает среднее число особей, приходящихся на условно выбранную единицу пространства, где их легко учесть (на 1 м², 1 км², 1 л или 1 м³ и т. п.). Численность отдельных популяций каждого вида животных постоянно меняется. Сезонные изменения зависят от того, что большинство животных нашей фауны размножается только летом, поэтому их популяции осенью, как правило, значительно многочисленнее, чем весной.

Максимальное число потомков, которое за жизнь могла бы произвести одна особь, называется *биотическим потенциалом вида*. Он у всех видов разный. Баланс рождаемости и смертности, а также вселения или выселения особей увеличивает или уменьшает численность и плотность популяции на занимаемой ею территории. В благоприятных условиях происходит *рост популяции*. Теоретически популяция может расти неограниченно, увеличиваясь в геометрической прогрессии. На деле этого никогда не происходит, потому что каждое местообитание имеет ограниченные ресурсы для жизни вида. Сумма этих ресурсов оценивается как *емкость среды*. Если численность популяции превысит емкость среды, начнется массовая гибель особей.

На примере популяции можно наблюдать одну из важнейших и удивительных особенностей живой природы — саморегуляцию популяций. Методы саморегуляции отработаны естественным отбором у каждого вида по-разному.

У подвижных животных прямого подавления соседей не происходит, при повышении плотности популяций усиливаются миграционные процессы, т. е. выселение части особей на другие территории. Это особенно наглядно происходит, например, у белок или тундровых грызунов леммингов. Перелеты стадной саранчи огромными тучами тоже выселение за пределы мест размножения. Некоторые виды способны тормозить воспроизведение популяций, вплоть до полного его прекращения. Таким образом, популяция представляет собой не просто сумму особей, а сложную систему, которая обладает способностью к регуляции своей численности и рациональному использованию ресурсов среды.

Различают популяции географические и экологические.

Географическими популяциями называют те естественные группы особей вида, которые населяют какой-либо определенный участок земного шара. Так, например, можно говорить о популяции песцов Командорских островов или популяции щук Сенежского озера. Нередко такая популяция вида территориально совпадает с тем или иным его подвидом.

Географические популяции обычно подразделяются на более мелкие группы особей данного вида, получившие название экологических популяций. Под экологической популяцией понимают совокупность особей данного вида, населяющую какое-либо определенное местообитание со специфическими экологическими условиями. Приспособливаясь к этим условиям, особи отдельных экологических популяций приобретают некоторые особенности поведения, образа жизни, а иногда даже морфологические отличия. Так, например, в лесостепной полосе имеется две экологические популяции волков: одна из них живет в лесных массивах, а другая — в степи, укрываясь на день в оврагах и в лесных посадках. Они четко различаются как по образу жизни, так и по поведению. В странах Европы существует две экологические популяции косуль — лесные и полевые.

Популяции животных различаются и по характеру взаимоотношений между отдельными особями. У некоторых видов все особи ведут одиничный образ жизни, встречаясь лишь на период размножения. У других популяции включают такие объединения, как *семьи*, *группы*, *стаи*, *стада*, *колонии* со своими сложными связями внутри них. В популяциях постоянно происходят изменения. Каждая популяция характеризуется определенным составом и структурой. Структура популяции определяется соотношением особей популяции по различным показателям — возрасту, полу, размерам и т. п. Для понимания динамики популяции и оценки ее состояния важно знать возрастную и половую структуру, т. е. количественное соотношение разных возрастных и половых групп животных в популяции. Так, если в популяции какого-либо промыслового животного больше, чем обычно, молодых особей, то это значит, что популяция данного животного эксплуатируется излишне интенсивно; напротив, если в популяции слишком много старых животных, то можно думать, что этот вид в данной местности недоиспользуется.

Соотношение особей по полу или возрасту отражает демографическую структуру популяции. Демографические описания важны для

предсказания судьбы конкретных популяций. Изменения численности популяции сильно зависят от того, какова в ней доля особей, приступивших к размножению, много или мало молодого пополнения, каков процент особей, закончивших размножение и т. д. Популяции характеризуются такими показателями, как *рождаемость*, *смертность*, *вселение* и *выселение особей*, а также *скорость роста*. Например, рождаемость — это число молодых особей, появившихся на свет за день, месяц или год, а смертность — число умерших за этот же период.

Численность популяции меняется по годам в зависимости от интенсивности размножения и смертности данного животного, что, в свою очередь, зависит от условий среды. Изучение условий среды позволяет прогнозировать численность различных животных, а это очень важно для регулирования этой численности, организации охраны и рационального использования.

Экосистемы и роль в них животных. *Биоценоз* (от греч. *bios* — жизнь, *koinos* — сообщество) — взаимосвязанная совокупность микроорганизмов, растений, грибов и животных, населяющих определенный участок суши или водоема. Человек может создавать искусственные биоценозы, в частности агроценозы, но они бывают устойчивыми только в том случае, если строятся по природным законам.

По систематическим признакам биоценоз делится на фитоценоз, зооценоз и микробиоценоз. Совокупность животных — членов данного биоценоза — носит название *зооценоза*. Место, занимаемое природным биоценозом (участок земли или водоема с характерными для него факторами среды), носит название *биотопа*. Условия биотопа во многом определяют видовой состав биоценоза. В структурном отношении биоценозы делятся на горизонты, слои, ярусы и т. д., т. е. они имеют пространственную структуру.

Роль животных в динамике биоценозов велика, каждый биоценоз имеет *видовую структуру*, т. е. численное соотношение отдельных видов. В каждой группе организмов в составе биоценоза (растений, грибов, бактерий, животных) имеются как многочисленные, так и малочисленные виды. Массовые виды составляют основу, как бы костяк любого биоценоза. Они определяют его облик, поддерживают главные связи. Такие виды называют *доминантными*. Биологи обычно и называют типичные природные биоценозы по доминирующими видам растений — ельник, степ ковыльная и т. п. В каждом таком биоценозе доминируют и определенные виды животных. Среди птиц в ельнике преобладают пеночки, синицы, а среди мелких грызунов — рыжая полевка. Большую роль играет изменение животными среды обитания в результате их жизнедеятельности. Например, своей роющей деятельностью животные вносят значительные изменения в почву данного сообщества. Выделения и трупы животных изменяют ее химический состав. Дятлы поселяются только в таких древостоях, где есть подходящие старые деревья для устройства дупел, а личинки комаров развиваются только в тех участках, где течение сильно замедлено густой прибрежной растительностью.

Уменьшение видового разнообразия грозит резкой вспышкой численности отдельных оставшихся видов. Это очень важное экологиче-

ское правило имеет непосредственное отношение к деятельности человека. Большинство видов, которые считаются вредителями сельского или лесного хозяйства, становятся массовыми именно из-за уменьшения численности их врагов и конкурентов. Таким образом, к появлению вредителей приводит и деятельность самого человека. Устойчивость биоценоза ослабевает постепенно, по мере снижения видового разнообразия. Для членов биоценоза не менее важна и *биотическая среда*, т. е. условия, которые создаются в результате присутствия других видов. Прежде всего, это возможность обеспечения себя пищей через прямые или косвенные связи. Питаюсь растениями, животные сильно влияют на растительность биоценозов. Их взаимоотношения определяют роль того или иного вида в динамике сообщества.

Положение вида в составе биоценоза называют его *экологической нишей*. Это, по образному выражению американского эколога Ю. Одума, профессия вида. Экологическая ниша — место в биогеоценозе, которое занимает вид, не конкурируя с другими видами за источник энергии, или, иными словами, экологическая ниша есть совокупность всех факторов среды, в пределах которых возможно существование вида. Для каждого вида характерна своя экологическая ниша. Два вида в одной экологической нише существовать не могут. Возможно лишь частичное перекрывание экологических ниш. Знание законов организации биоценозов дает возможность поддерживать природные сообщества и грамотно создавать искусственные агроценозы.

Каждый живущий организм связан с окружающей средой потоками вещества и энергии. В биоценозах же, кроме того, все виды связаны друг с другом сложными пищевыми связями. В итоге любой биоценоз представляет собой некое единство со своим биотопом. Система, состоящая из биоценоза и биотопа, называется *биогеоценозом* или *экосистемой*. В первом названии подчеркивается тесная взаимосвязь живых (био-) и неживых (гео-) компонентов на определенном участке земной поверхности. Экосистема характеризуется относительно самостоятельным обменом веществами и особым типом использования солнечной энергии. Примерами экосистем (биогеоценозов) являются луга, леса, поля, водоемы, для каждой из которых характерны определенные биомасса и экологическая продуктивность. Экосистема находится в непрерывном изменении и развитии вследствие постоянно меняющихся взаимоотношений ее компонентов и общего метаморфоза природы данного района. Состав и взаимодействие компонентов зависят от времени суток, времени года, а также исторически. Из потоков веществ, поддерживающих жизнь организмов разных видов, в экосистемах складывается *биологический круговорот веществ*. Функционально биоценоз делится по ступеням экологической пирамиды на группы организмов — продуцентов, консументов и редуцентов, объединенных пищевыми (трофическими) связями. Эти связи носят всеобщий характер, так как нет ни одного вида на Земле, который не служил бы пищей другим или сам не использовал бы для этих целей другие виды.

Совместная деятельность разных по экологическим функциям групп организмов и является двигателем биологического круговорота веществ. Экосистемы устойчивы лишь в том случае, если входящие в их состав биоценозы поддерживают круговорот веществ достаточно полно. При этом в экосистемы должна постоянно поступать энергия извне, так как, расходуясь на жизнедеятельность организмов, энергия постепенно рассеивается в виде тепла в окружающем пространстве. Одна и та же порция вещества и заключенной в нем энергии не может бесконечно передаваться по сложной пищевой сети, связывающей организмы в биоценозе.

Приведем в качестве примера короткую пищевую цепь: трава (первый трофический уровень) — саранча (второй трофический уровень) — скворец (третий трофический уровень). Трава с экологической точки зрения — продуцент (в результате фотосинтеза накапливает органическое вещество), саранча — консумент первого порядка (питается продуцентом), а скворец — консумент второго порядка (питается травоядным животным).

На самом деле трофические связи значительно сложнее; они образуют пищевые сети, состоящие из переплетения более коротких пищевых цепей. Пищевую сеть можно схематически изобразить в виде густой паутины, охватывающей весь органический мир, начиная от любого вида. Например, домашние овцы поедают десятки видов растений, сами идут в пищу человеку, крупным хищникам и множеству паразитов. Каждый из этих видов связан пищевыми отношениями со своим кругом жертв и потребителей, мертвые остатки растений и животных также служат источником пищи для множества других видов. Пищевые сети не имеют ни начала, ни конца, так как каждый вид прямо или косвенно связан со многими другими.

Лишь небольшая доля усвоенной животным пищи идет на рост, т. е. на увеличение его биомассы. Большая часть тратится на поддержание обмена веществ. Подсчитано, что в среднем на рост идет всего лишь около 10 % усвоенной энергии. Таким образом, передача количества, массы и энергии по пищевым цепям подчиняется так называемому «правилу десяти процентов». В каждом последующем звене пищевой цепи количество задерживаемой энергии уменьшается примерно в 10 раз, и уже через 4—5 звеньев она практически полностью иссякает.

Органическое вещество, создаваемое растениями в процессе фотосинтеза за единицу времени, называется первичной продукцией. Часть этой продукции идет на прирост биомассы растений и может быть использована далее в пищевых цепях. За ее счет происходит прирост биомассы животных, которая называется вторичной продукцией; вторичной продукции образуется, по крайней мере, в 10 раз меньше, чем первичной. Следовательно, на создание 1 кг массы растительноядных животных в природе затрачивается в 10 раз больше солнечной энергии, чем на 1 кг массы растений. Соответственно, продукция животноядных обходится в 100 раз дороже.

«Правило десяти процентов» можно выразить в виде пирамиды. Нижняя, широкая ступень пирамиды отражает количество энергии,

запасенной продуцентами, а каждая последующая ступень оказывается в 10 раз меньше предыдущей. Если же мы будем сравнивать не энергию, а массу организмов каждого трофического уровня (биомассу), то правильность пирамиды может быть нарушена. Например, в океане первичная продукция, создаваемая одноклеточными водорослями, чрезвычайно высока, поскольку они размножаются с большой скоростью. Но не менее высока и скорость поедания этой продукции консументами, поэтому общая масса водорослей все время остается невысокой. Она оказывается меньше, чем общая масса животных, выросших за ее счет.

В 60—70-х гг. прошлого века учеными разных стран была составлена карта продуктивности суши и океана. Из наземных экосистем наиболее высокая первичная продукция зарегистрирована в экосистеме влажного тропического леса, а в океанах — в экосистеме коралловых рифов. Она составляет до 25 г сухого органического вещества на 1 м³/сут. Это предельные возможности природных экосистем в современных условиях. Средняя продуктивность степей, лесов, лугов, озер составляет от 5 до 15 г на 1 м²/сут. В этих же пределах варьирует и продукция сельскохозяйственных угодий, и лишь очень интенсивное земледелие с большими дополнительными вложениями средств и энергии может увеличить продуктивность до уровня, сопоставимого с максимальным.

Обширные пространства суши и океана относятся к районам с низкой продуктивностью; к их числу относятся жаркие или холодные пустыни, скалистые ландшафты, тундры, а также центральные океанические акватории. Здесь создается от 0,5 до 3—4 г органического вещества на 1 м²/сут. Основными факторами, ограничивающими фотосинтез растений, выступают недостаток тепла, недостаток влаги и необходимых биогенных элементов, например вдали от берегов в океане.

Разные экосистемы Земли сильно различаются и по интенсивности круговорота биогенных элементов. Она определяется во многом скоростью минерализации мертвых органических остатков, которая зависит от деятельности редуцентов. Там, где мало тепла и влаги, круговороты замедлены. Увеличение их интенсивности — один из путей повышения биологической продукции.

Даже в самых устойчивых экосистемах Земли круговорот биогенных элементов не может быть полностью замкнут. Часть элементов переносится ветрами и течениями, сносится в понижения рельефа, миграирует вместе с поверхностным стоком и подземными водами. В результате все экосистемы суши и океана оказываются связанными в глобальную экосистему — биосферу.

Экосистемы со сбалансированным круговоротом биогенных элементов могут существовать бесконечно долго, пока внешние силы не выведут их из равновесия. И действительно, темнохвойная тайга, ковыльные степи, широколистственные дубравы занимали свои места тысячелетиями после последнего оледенения, и лишь деятельность человека за последние столетия сильно изменила эти ландшафты. Однако экосистема — это не застывшая, а открытая, постоянно меняющаяся

система. Естественное развитие экосистемы ведет к ее смене, при которой одни биоценозы сменяют другие под влиянием природных факторов среды. Для экосистем характерна саморегуляция — способность к восстановлению равновесия после природного или антропогенного воздействия. Вместе с тем в природе существует множество нестабильных экосистем, направленно изменяющихся даже без какого-либо вмешательства извне. Мелеют и застаивают неглубокие озера, превращаясь в болота, на месте которых со временем появляются заросли кустарников и т. д.

Исторические смены экосистем носят название *сукцессий*. Они происходят либо в силу естественных изменений природы данного района, либо под воздействием хозяйственной деятельности человека, либо вызываются стихийными бедствиями.

Сукцессии могут быть первичными и вторичными (восстановительными). *Первичные сукцессии* начинаются как бы с нуля на субстрате, не измененном деятельностью живых организмов. На также безжизненные участки заносятся ветром и водой семена, споры, летят насекомые, забегают мелкие грызуны и т. п. Некоторые из них остаются на данном участке. Они, как правило, вытесняются новыми вселенцами, успев частично изменить среду. В таких сообществах еще не сформировались сложные пищевые цепи, не заняты все экологические ниши, растительная продукция не полностью используется консументами первого порядка и поэтому накапливается в экосистеме. Постепенно сообщество становится все более устойчивым. В нем нарастает видовое разнообразие, происходит все большее расхождение видов по экологическим нишам, ослабляется конкуренция. Такие экосистемы называются зрелыми. Их сообщества устойчивы, так как изменения среды, вызываемые одними видами, компенсируются деятельностью других. Круговорот веществ в зрелых экосистемах сбалансирован.

Сукцессии, начинаяющиеся после частичного нарушения экосистем, называются *вторичными* или *восстановительными*. Они происходят, например, после рубки леса, вспашки луга, добычи полезных ископаемых открытым способом. В этих случаях уничтожаются не все элементы экосистемы, остаются сформированная живыми организмами почва, семена, споры, выживают некоторые виды животных. Восстановительные сукцессии протекают несколько иначе, чем первичные, но тоже приводят к формированию стабильных, зрелых экосистем. Время первичных сукцессий исчисляется в природе сотнями лет, восстановительные происходят несколько быстрее. Например, ельники в европейской части России после рубок восстанавливаются за 60—80 лет, проходя стадии кустарниковых зарослей и мелколиственных лесов.

Наряду с крупномасштабными и долгосрочными сукцессиями в природе протекает множество мелкомасштабных и краткосрочных. Заставают, тоже проходя ряд этапов, выбросы кротов, сусликовины в степях и т. п.

Особая категория сукцессий характерна для таких местообитаний, где скапливаются запасы мертвого органического вещества — в лесном опаде, навозе, компостах. На эти богатые энергией запасы набрасывается целая армия мелких и микроскопических потребителей — консументов и редуцентов. Своей деятельностью они быстро меняют субстрат, происходит смена сообществ, и процесс идет по всем законам сукцессии, постепенно замедляясь, пока органическое вещество не будет минерализовано и частично превращено в перегной, включающийся в состав почвы.

Процессы минерализации органических веществ в биологическом круговороте биогенных элементов столь же важны, как и создание продукции. Везде: в лесах, степях, на лугах и в болотах — с разной скоростью идет эта гигантская неслышная и невидимая работа, поддерживающая круговорот биогенных элементов. Замедление таких процессов, сбои в них угрожают устойчивости экосистем. Поэтому здоровая, богатая жизнью почва — это не только источник питательных веществ для растений, но и условие стабильности наземных сообществ.

Понимание законов протекания экологических сукцессий важно для многих сторон деятельности человека. Из этих законов следует, что экосистема не может одновременно быть устойчивой и накапливать избыток первичной продукции. Этот избыток мы собираем в виде урожая на фермах, полях и огородах, создавая при этом крайне неустойчивые экосистемы, которые требуют постоянной поддержки человека — дополнительного кормления, поения, удобрений, посевов и т. п. Эта неустойчивость проявляется и во вспышках численности вредителей, паразитов, и в эрозии почв, и в исчерпании запасов минеральных соединений. Если на следующий год не засеять поле вновь, то оно стремительно преобразуется в пустошь, а затем в луг или кустарниковые заросли. За первичную сельскохозяйственную продукцию человек платит нестабильностью среды и необходимостью вложения большого труда и дополнительной энергии. Здесь идет постоянная борьба против природных сукцессий.

Управление сукцессиями — один из основных путей экологически грамотного сотрудничества с природой. Чтобы не подрывать ее стабильности и иметь возможность получать первичную продукцию, люди должны так формировать ландшафты, чтобы они включали и зрелые, и незрелые экосистемы.

ОСНОВЫ ЗООГЕОГРАФИИ

Зоогеография — это наука, изучающая географическое распределение животных и устанавливающая общие закономерности их распределения по земному шару. Она изучает области распространения отдельных видов животных и их систематических групп, выясняет исторические причины и экологические факторы, определяющие границы такого распространения, описывает животный мир отдельных регио-

нов Земли и прослеживает пути формирования фауны этих регионов. В разных областях суши, пресных вод и морей обитают различные группы животных. Немногие животные, подобно человеку, способны жить во всех широтах. Зоогеография имеет большое прикладное значение, поскольку ее данные широко используются в сельском и промысловом хозяйствах. Очевидно, чтобы организовать широкие работы по борьбе с вредителями растений и паразитами сельскохозяйственных животных, необходимо знать, где они встречаются. Планировать добычу рыбы, дичи, пушных зверей и других промысловых животных невозможно без глубоких знаний их распространения и тенденций изменения численности. Изучение распространения паразитов человека и переносчиков возбудителей заболеваний имеет большое значение для организации здравоохранения людей.

Важным понятием зоогеографии является понятие *фауна* — исторически сложившаяся совокупность животных, населяющих определенную территорию (акваторию). Так, например, все животные, населяющие озеро Байкал, образуют его фауну. Слагающие ее виды животных хорошо приспособлены к условиям обитания в данном районе. На различных континентах обитают свои представители животного мира, при этом концентрация таксонов обычно наблюдается на самых благоприятных и больших для обитания территориях каждого континента, в менее благоприятных местах число таксонов снижается. Иными словами, животные наиболее разнообразны и многочисленны на самых больших и благоприятных для их обитания территориях земного шара. Среди видов животных, входящих в состав фауны большей части регионов, обычно можно выделить виды, которые образовались и развились в этом районе, и виды-иммигранты, проникшие сюда из других областей. Как правило, для крупных таксонов животных характерны центры происхождения, откуда они потом и распространялись, — явление, называемое географической радиацией.

На протяжении истории нашей планеты ее фауна в разных регионах изменялась. Так, в палеогене на юге современной России обитали древние слоны, жирафы, лошади и другие субтропические и тропические животные (позже вымершие). Ледниковый период изменил состав фауны. Появились мамонты, шерстистые носороги, мускусные быки, северные олени, пещерные львы и медведи, песцы, лемминги и другие животные. По окончании ледникового периода многие из этих видов вымерли или постепенно мигрировали на север, а местная фауна приобрела современный вид, формируясь как из живших здесь видов, сумевших приспособиться к новым условиям жизни, так и из иммигрантов, проникших из других регионов.

Несмотря на такие исторические изменения, отдельные виды животных, характерные для существовавшего здесь прежде животного мира, сохраняются. Эти виды животных называются реликтовыми или реликтами. Например, в тундре Северной Америки и Гренландии кое-где сохранились стада своеобразных мускусных быков, обитавших

в ледниковый период в большом количестве по всей Арктике. Их можно рассматривать как реликтов фауны ледникового периода.

Чтобы охарактеризовать распространение современных представителей животного царства, выделяют «фаунистические области» со свойственным им видовым составом. Представление об этих областях впервые было сформулировано еще в XIX в., важную роль в этом сыграли работы А. Уоллеса. Главные зоогеографические области суши — голарктическая (палеарктическая, неарктическая), эфиопская, индомалайская, неотропическая, австралийская и антарктическая. Все зоогеографические области делятся на подобласти, например палеарктическая область делится на европейскую, средиземноморскую, сибирскую, среднеазиатскую и маньчжурсскую. Каждая из подобластей имеет своих характерных представителей фауны тундры, тайги, леса, степи, пустыни, побережья, островов.

1. **Голарктическая область** занимает большую часть Северного полушария. Этую территорию относят также к двум отдельным областям — **Неарктической** (Северная Америка) и **Палеарктической** (Евразия, к северу от тропиков, и Северная Африка). Общность неарктической и палеарктической фаун объясняется тем, что в плиоцене и четвертичном периоде Чукотка и Аляска соединялись сухопутным «мостом», по которому в обоих направлениях шла миграция животных. Из Евразии в Америку мигрировали горные бараны, лоси, олени, бурый медведь, а из Северной Америки в Евразию — северный олень, овцебык и другие. Фауна Голарктики, несмотря на большую территорию, относительно бедна. Эндемичны кроты, бобры и тушканчики; из птиц — тетеревинные и гагары; из рыб — осетровые, лососевые, щуковые и др.

2. **Эфиопская область** занимает Африку, кроме северной. Фауна типично материковая, очень богатая. Эндемичны гориллы, шимпанзе, трубковусы, жирафы, бегемоты, страусы, цесарки и др. Обитают лемуры, носороги, слоны. Особенны характерны антилопы, типичен бородавочник. Отсутствуют кроты, олени, медведи. Из воробышкообразных характерны ткачики и нектарницы. Попугаев мало, отсутствуют тетеревинные. Из пресмыкающихся многочисленны хамелеоны, агамы, гекконы, вараны, гадюки, питоны, бокошайные черепахи, крокодилы. Из земноводных много жаб, лягушек (шпорцевые), червяг. Из рыб в пресных водоемах преобладают карловые, характерны протоптеры, мнохоптеры. Из насекомых обильны термиты, саранчовые.

3. **Индо-Малайская область** занимает южную Азию. Главный тип фауны — лесной, в котором большую роль играют древесные животные. Из млекопитающих эндемичны долгопяты, шерстокрылы, туапайи, орангутаны, гиббоны. Многочисленны летучие мыши, в том числе растительноядные (летучие собаки), грызуны, кошачьи; характерны тапиры, носороги, олени, быки, антилопы, слоны, речные дельфины. Число видов птиц очень велико, в том числе фазановых. Из пресмыкающихся распространены гекконы, вараны, агамы, сцинки, змеи, черепахи и крокодилы. Земноводные менее разнообразны, хвостатые от

существуют, квакши малочисленны. Фауна рыб весьма разнообразна — особенно карповых.

4. *Неотропическая область* занимает Центральную и Южную Америку. Фауна очень богата. Среди млекопитающих эндемичны броненосцы, ленивцы, муравьеды, морские свинки, шиншиллы, ламы и др. Обитают сумчатые (опоссумы), тапиры, капибары, ягуары, оцелоты, ламантины, речные дельфины и др. Характерны кровососущие летучие мыши. Все обезьяны принадлежат к широконосым (игрунки, ревуны). Фауна птиц исключительно разнообразна — нанду, тинаму, колибри, попугаи (в том числе ара), гоацины, кондоры, гарпии, туканы. Пресмыкающиеся представлены кайманами, бокошнейными черепахами, змеями (удавы), ящерицами (особенно игуаны). Среди земноводных типичны бесногие, большое число бесхвостых (древесницы, пипы). Из рыб распространены лепидосирен, арапайма, пиранья и др. Из пауков выделяются гигантские птицееды. Среди насекомых многочисленны муравьи-листорезы, характерны крупные дневные бабочки.

5. *Австралийская область* занимает Австралию и Новую Гвинею и Тасманию. Фауна характеризуется обилием эндемиков (сумчатые и однопроходные) и малым количеством плацентарных (кроме завезенных человеком, встречаются летучие мыши и настоящие мыши). Из птиц эндемичны беседковые, райские, казуары, эму, какаду, волнистые попугайчики, сорные куры и др. Из пресмыкающихся характерны плащеносная ящерица, молох, из рыб — неоцератод.

6. *Антарктическая область* занимает Антарктиду, Новую Зеландию. Фауна характеризуется крайней бедностью. Здесь очень мало наземных животных, практически нет летающих насекомых, сухопутных млекопитающих и пресноводных рыб. Млекопитающие представлены ластоногими (тюлень Уэдделла, Росса, тюлень-крабоед, морской леопард) и китами (из усатых — синий кит, финвал, сейвал, горбач; из зубатых — бутылконос, косатка). Из птиц летом многочисленны буревестники, поморники, альбатросы, крачки и др. Самые типичные птицы — пингвины, среди которых обычны крупный императорский пингвин, пингвин Адели.

Между зоогеографическими областями лежат различные по ширине переходные полосы, где происходит смешение и взаимопроникновение фаун. Северо-западная граница Австралийской области — это «линия Уоллеса». А. Уоллес, один из основателей зоогеографии, определил эту зону как границу между областями.

Каждый вид занимает определенную область распространения, или ареал. Ареал (от лат. *area* — площадь) — часть земной поверхности (или акватории), в пределах которой встречается тот или иной вид животных.

Характерной особенностью распространения животных является ограниченность. Амфибии, например, в своем распространении приурочены в первую очередь к тропикам. Другая особенность распространения — зональность. Большинство амфибий приурочены к тропикам, но саламандры, как правило, являются северно-умеренными видами. Довольно обычное явление — прерывистое распространение, т. е. на-

личие одних и тех же или родственных животных на более или менее разобщенных территориях.

Границы ареалов животных, как правило, обусловливаются, с одной стороны, их естественным стремлением к расселению на возможно большей территории, а с другой — действием различных факторов, ограничивающих это расселение. Естественное расселение животных нередко можно наблюдать в тех случаях, когда они еще не заселили все места, подходящие для их жизни. Чаще всего расширение ареала идет путем постепенного захвата соседних участков. Кольчатая горлица (*Streptopelia decaocto*) еще 50 лет назад в России не встречалась, в настоящее время ее ареал простирается до Поволжья и Ленинградской области.

Иногда расселение животного происходит скачкообразно, путем преодоления какого-либо препятствия. Например, в 20-х гг. прошлого столетия стайка кочующих белок, преодолев безлесную тундру северной части Камчатского полуострова, проникла в его центральные горно-лесные районы и быстро их заселила. Иногда расселение животных происходит пассивно, например морскими течениями, бурями и пр. В некоторых случаях расселению животных способствует человек, завозя их (сознательно или случайно) в новые районы. Так, например, кролики, крысы, домовые мыши, собаки, кошки и другие были завезены людьми в самые различные страны мира. Но обычно границы ареалов более или менее постоянны. В настоящее время чаще наблюдается сокращение ареала вследствие антропогенного воздействия. Так, сплошной ранее ареал бобра (*Castor fiber*), охватывавший большую часть Европы и Северной Азии, в прошлом веке из-за интенсивного промысла оказался разделенным на ряд небольших изолированных участков.

Если вид встречается на всем протяжении ареала, то такой ареал называется сплошным. Часто при этом в пограничных районах ареал разбивается на достаточно обособленные участки. Бывает прерывистый ареал, когда между отдельными его участками имеются промежутки настолько значительные, что любой контакт между разделенными ими популяциями вида исключается. Например, голубая сорока (*Cyanopis cyanea*) обитает на Дальнем Востоке России, в Китае и, кроме того, в Испании и Португалии. Размеры ареала у разных видов могут сильно различаться; некоторые животные обитают только на очень ограниченном пространстве (на острове), другие распространены очень широко — на нескольких материках, занимая на них громадные пространства. Виды животных, встречающихся только в данном районе, называются эндемиками. Иногда ареалы эндемиков узко ограничены. Особенно богаты эндемиками территории, изолированные географически или экологически. Так, небольшие рыбы голомянки (*Gasterosteus aculeatus*) — эндемики озера Байкал, где встречается 76 % эндемиков. В фауне островов, расположенных в океанах вдали от материков, как правило, много эндемичных форм. Это объясняется тем, что обычно фауна подобных островов формируется из иммигрантов, в разное время проникших на них с соседних материков. Эволюция этих иммигрантов, адаптация к местным специ-

фическим условиям и привела к образованию на океанических островах эндемичных форм животных.

!

Группы организмов (например, воробьинообразные), распространенные практически по всему земному шару (точнее — по всей суще), называются космополитами.

У тех животных, которые ведут оседлый образ жизни, ареал по сезонам года заметно не меняется. Но у тех, которые предпринимают дальние сезонные миграции, он может претерпевать значительные изменения в течение года. Так, песец (*Alopex lagopus*) летом, в период размножения, живет только в зоне тундр Крайнего Севера; ареал его в это время года спускается к югу лишь до северной границы лесов. К зиме же часть песцов откочевывает далеко к югу, в зону тайги; поэтому зимой южная граница ареала песца проходит значительно южнее, чем летом.

Еще сильнее меняются по сезонам года области распространения перелетных птиц. У этих видов имеется два ареала — летний (гнездовой) и зимний (зимовочный). Нередко они удалены друг от друга на тысячи километров.

Ареал любого вида животного изменяется также исторически как в силу эволюции животного, так и изменений среды за длительный отрезок времени.

Животный мир отдельных зон России. Территория России расположена в пределах Голарктической области, поэтому для фауны нашей страны характерны все черты животного мира этой области. Но различные зоны России населяют далеко не одни и те же животные. Зональность в их распределении связана с природными зонами Евразийского материка. Каждой ландшафтной зоне его свойственна характерная для нее фауна.

Зона тундр занимает самый север России. Ее безлесные пространства начинаются недалеко от нашей западной границы и тянутся узкой полосой вдоль северной части Кольского полуострова, затем несколько расширяются, достигая в ширину выше 500 км, и простираются до восточных границ, занимая всю территорию к северу от Колымского хребта, т. е. весь северо-восток Азии, а также острова Северного Ледовитого океана, исключая Камчатку.

Для зоны тундр характерен суровый арктический климат, с длительной морозной, но относительно малоснежной зимой и коротким, прохладным летом. Для тундры характерны сильные ветры, крайне плотный снеговой покров и большая облачность. Болотистые, песчаные или каменистые почвы тундры покоятся на мощном слое вечной мерзлоты. Растительность почти не возвышается над поверхностью почвы, она состоит из влаголюбивых трав (осок, злаков), мхов и лишайников.

Животный мир тундры своеобразен (рис. 290). Здесь обитает сравнительно немного видов животных, приспособившихся к суровым местным условиям. Преобладают наземные и норные животные. Многие из них на зиму приобретают белую окраску под цвет снега. Из млекопитающих характерны северный олень, волк, песец, белый медведь (по

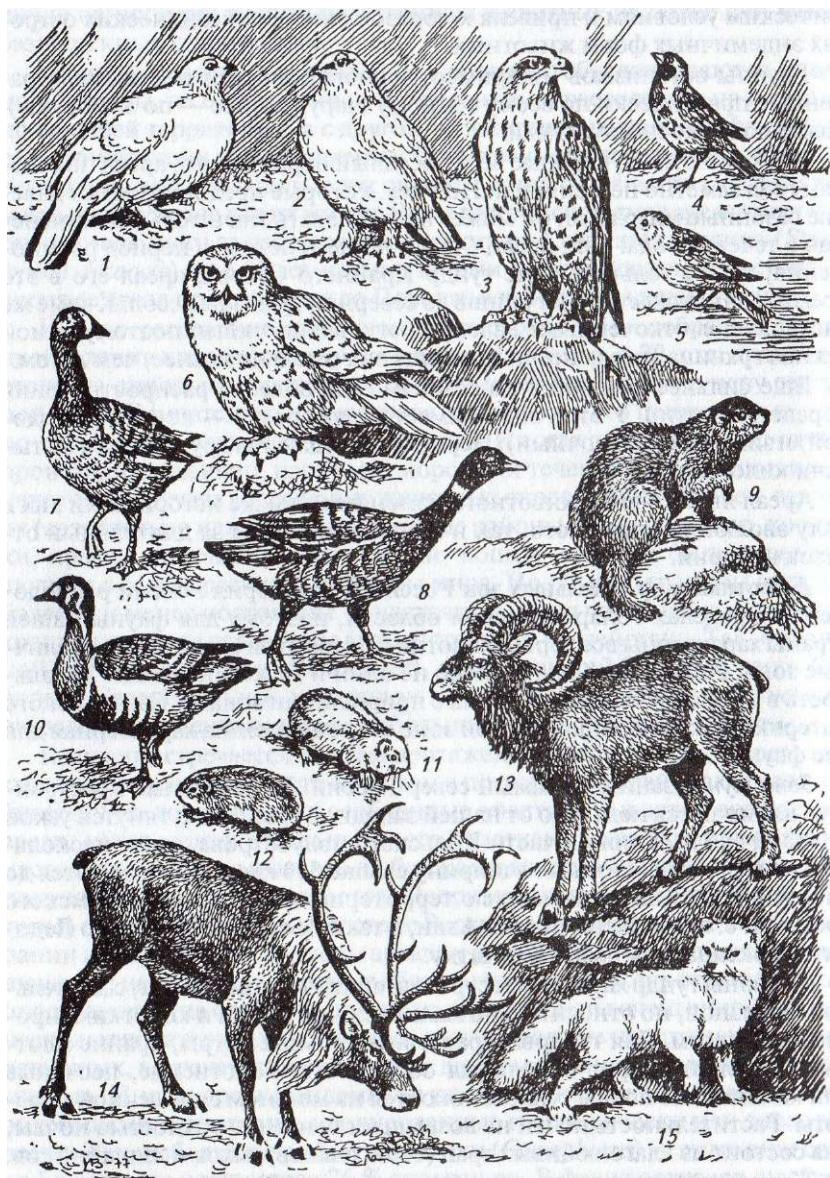


Рис. 290. Характерные представители Голарктической области (зоны тундры):
 1 — тундровая куропатка; 2 — белая куропатка; 3 — мохноногий канюк; 4 — лапландским подорожник; 5 — пуночка; 6 — белая сова; 7 — белолобая казарка; 8 — черная казарка. 9 — черношапочный сурок; 10 — краснозобая казарка; 11 — копытный лемминг; 12 — обский лемминг; 13 — снежный баран; 14 — северный олень; 15 — песец

побережью и на островах), своеобразные грызуны лемминги, полевки. Заходят лисица, росомаха.

Из птиц для этой зоны наиболее типичны белая и тундряная куропатки, белая сова, подорожники. Условия существования зимой настолько суровые, что все живое старается покинуть тундру. Многие обширные участки тундры поражают своей безжизненностью. Весной тундра оживает — прилетают на гнездовья пурпурные, гуси, кулики, сапсаны. На скалах морских побережий расположены птичьи базары, где гнездятся чистики, чайки, поморники, гагары. Пресмыкающиеся и земноводные в тундре отсутствуют.

Междуди зоной тундры и зоной лесов лежит полоса лесотундры, фауна которой образована представителями животного мира обеих зон.

Зона лесов тянется широкой полосой от западной границы России до побережья Тихого океана. К югу она спускается примерно до линии Брянск — Тула — Рязань — Н. Новгород — Казань — Уфа — Екатеринбург — Тюмень — Новосибирск — Алтай. Климат зоны лесов не столь суровый, более умеренный, чем климат зоны тундр. Зимы здесь короче, мягче и многоснежнее. Лето более продолжительное и теплое. Вечная мерзлота характерна только для северо-восточных частей зоны. Прежде большая часть зоны была покрыта лесами различного типа. Но в настоящее время значительные площади преобразованы в антропогенные ландшафты.

Животный мир зоны лесов богаче видами, чем животный мир зоны тундр. Обширная лесная зона делится на подзону хвойных лесов (тайги) и подзону смешанных и широколиственных лесов.

Первая занимает всю азиатскую часть зоны лесов и север Европы до линии Петрозаводск — Вологда — Киров — Казань. Тайга однообразна и достаточно пустынна, ее фауна сложилась в недавнее время (в геологическом смысле) из южных выходцев (рис. 291). Для тайги характерны лось, рысь, бурый медведь, росомаха, соболь, колонок, бурундук, заяц-беляк, белка, летяга, глухарь, рябчик, гоголь, большой улит, желна, филин, кедровка, сычи, клесты, синицы и другие животные. Для зоны смешанных и широколиственных лесов (рис. 292) типичны благородный олень, косуля, кабан, лисица, барсук, лесная куница, норка, бобр, лесная соня, тетерев, вяхирь, ястребы, ушастая сова, вальдшнеп, зеленый дятел, кукушка, иволга, зяблик, дрозды и другие обитатели.

В последнее столетие в зоне лесов вслед за вырубкой лесных массивов расселилось много животных, характерных для сельскохозяйственных ландшафтов, — еж, черный хорек, заяц-русак, серая куропатка, перепел и др.

Пресмыкающиеся в зоне лесов немногочисленны (гадюка обыкновенная, уж обыкновенный, ящерицы — живородящая, прыткая). Из земноводных — некоторые виды лягушек (остромордая, сибирская), жаб (обыкновенная), тритонов (обыкновенный), сибирский углозуб.

К югу от зоны лесов тянется полоса лесостепи, где обитают представители как лесной, так и степной зон.

Зона степей тянется через юг европейской части России на восток до предгорий Алтая. Европейско-азиатская степь — самобытная об-



Рис. 291. Характерные представители Голарктической области (зоны тайги):
 1 — трехпалый дятел; 2 — женна; 3 — глухарь; 4 — кедровка; 5 — свиристель; 6 — бородатая неясыть;
 7 — белокрылый клест; 8 — мохноногий сыч; 9 — соболь; 10 — колонцовый белка;
 11 — лось; 12 — летяга; 13 — бурундук; 14 — кабарга; 15 — кабарга; 16 — росомаха



с. 292. Характерные представители Голарктической области (зоны смешанных и широколиственных лесов):

- лесная куница; 2 - черный хорек; 3 - садовая соня; 4 - полчок; 5 - иволга-
белая; 6 - землеройка; 7 - квакша; 8 - зеленый дятел; 9 - дубонос; 10 - европейская неясыть;
11 - белка, 12 - зубр; 13 - европейская косуля

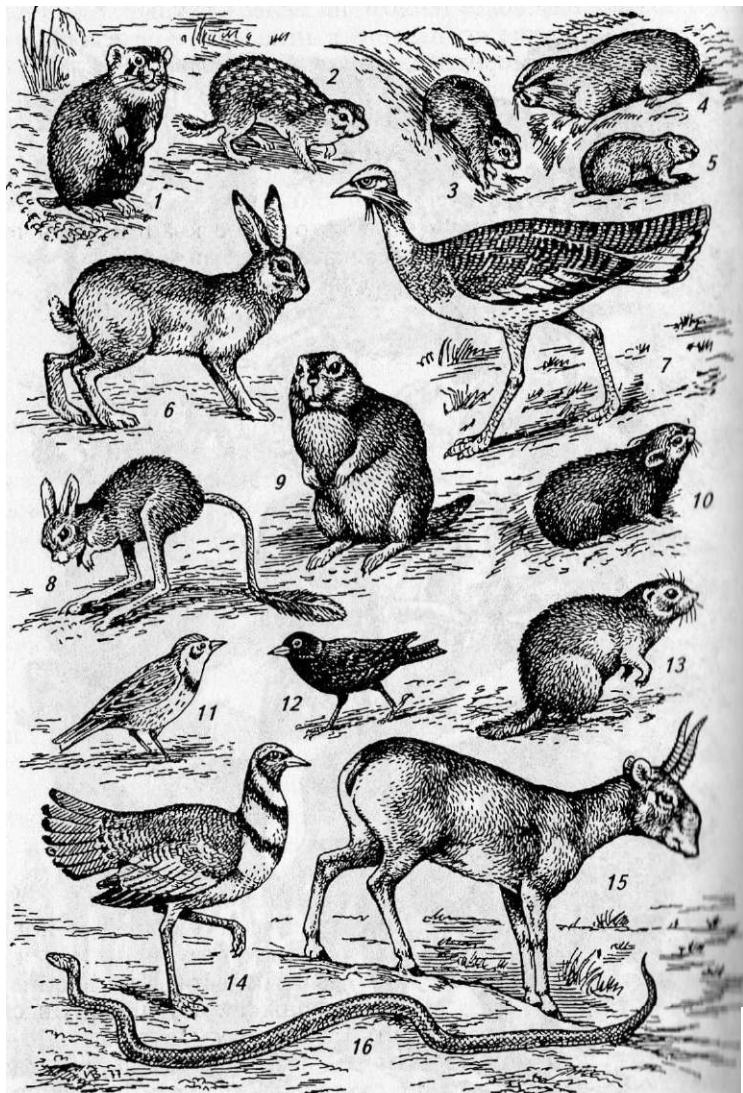


Рис. 293. Характерные представители Голарктической области (зоны «чети «I» |
 1 — обычновенный хомяк; 2 — крапчатый суслик; 3 — степная мышонка, / • |
 5 — степная пеструшка; 6 — заяц-русак; 7 — дрофа; 8 — большой тушка... и " |
 10 — степная пищуха; 11 — степной жаворонок; 12 — черный жаворонок, Г' »
 щекий суслик; 14 — стрепет; 15 — сайгак; 16 — полоз желтобрюхий

масть с особым, свойственным только ей растительным и животным миром (рис. 293).

Климат здесь еще более теплый, но менее влажный, чем в зоне лесок, нередко дуют сухие жгучие ветры. Зима, хотя и не особенно снежная, сопровождается морозами, вышагами и метелями. Для степной зоны характерно отсутствие лесов, за исключением искусственных посадок. Ландшафт здесь открытый. Прежде почти вся зона была покрыта целинными ковыльными или ковыльно-полынными степями. Но в наше гоящее время они почти полностью распаханы и вовлечены в хозяйственный оборот. Поэтому ныне для зоны характерен так называемый сельскохозяйственный ландшафт, в котором бескрайние поля чередуются с пастбищами, лесными посадками, садами и поселениями людей. Самое лучшее время здесь — весна. Быстро становится тепло, снег сходит, и степь превращается в настоящий цветник, появляется бесчисленное множество луковичных растений.

Прежде в целинных степях паслись стада сайгаков, диких лошадей — тарпанов. Сейчас тарпаны истреблены, а сайгаки оттеснены в зону полупустынь и пустынь. Особенно многочисленны в степной зоне различные грызуны: сурки, суслики, тушканчики, хомяки, полевки и мыши. Все степные грызуны роют норы, большинство на зиму впадает в спячку. Многие из них являются серьезными вредителями растениеводства. Достаточно многочисленны в степях зайцы-русаки и степные виды птиц, из хищных зверей — волки, лисы, корсаки, степные хорьки.

Из типично степных птиц встречаются дрофа, стрепет, журавль-красанка, коростель, перепел, степной орел, степной лунь, кобчик, степные жаворонки. Для степных животных характерна однообразная серовато-желтая окраска под цвет почвы и выгоревшей растительности.

В степной зоне значительно больше, чем в лесной, различных пресмыкающихся — гадюка степная, уж водяной, узорчатый полоз, ящерицы (ящурка разноцветная, агама степная), черепаха болотная и др. Земноводные малочисленны — жаба зеленая, квакша обыкновенная.

Зона пустынь и полупустынь занимает пространства в Калмыкии и Нижнем Поволжье. Эта зона отличается засушливым климатом. Почвы скудные — сероземы, пески, солончаки и солонцы. Растительность не образует сплошного покрова. Местами имеются заросли пустынных кустарников. Большая часть зоны занята злаково-полынной полупустыней, служащей переходной полосой между степью и пустыней. Есть и участок настоящей пустыни. Животный мир этой зоны крайне ориентирован (рис. 294). Из копытных характерны сайгаки (преимущественно держатся в полосе полупустыни). Из хищных млекопитающих обитают степные кошки, волки, корсаки и другие виды. Многочисленны грызуны — тушканчики, песчанки, суслики. Из пустынных птиц наиболее характерны дрофа, стрепет, жаворонки. Многочисленны различные пресмыкающиеся — агамы, круглоголовки, полозы и прочие виды. Земноводных здесь мало. Почти все звери пустынь — наземные и норные животные, последние зимой обычно впадают в спячку. Многие из них имеют покровительственную окраску песочного цвета.

У 2 /

Рис. 294. Характерные представители Голарктической области (зоны пустынь и поясов щебя и щебя)
1 — тонкопалый суслик; 2 — саксаульная сойка; 3 — гребнепалый тушканчик; 4 — мицтый;
5 — ушастая круглоголовка; 6 — белобрюхий рябок; 7 — каракал; 8 — барханша;
9 — ушастый еж

Особый зоогеографический район нашей страны представляет собой южная часть Приморского края. Здесь сибирская тайга и гуарничики Китая сходятся вплотную. Смешение южных и северных видов животных образует своеобразные сообщества. В ряду с характерными для таежной зоны животными обитают тигр, леопард, белогрудый медведь, харза, минхурский заяц, фазан, дальневосточная черепаха и другие, характерные для широколиственных и смешанных лесов маньчжурского типа.

Зоогеографически выделяются также высокогорные области Гималаев и Южной Сибири. На высокогорных альпийских лугах Кавказа пасутся стада горных козлов — туров и серн, держатся кавказский горный козел.

(горная индейка) и кеклик (каменная куропатка), белоголовый сип, клушица и другие животные.

На высокогорных лугах среди скал и ледников гор Южной Сибири (Алтая и Саян) можно встретить сибирского козерога, архара, снежного барса, снежного барса, красного волка, сурков, пищух, алтайского улара, кеклика, скалистого голубя и других представителей высокогорной фауны.

В настоящее время, когда природа нашей страны быстро меняется и результате хозяйственной деятельности человека, меняется также и характер животного мира различных областей. Вслед за продвижением на север земледелия расселяются в том же направлении и многие животные, жизнь которых связана с сельскохозяйственными угодьями. В населенных пунктах поселяются животные синантропы — крысы, мыши, голуби, скворцы, воробы и др.

Фауна России (сохранение биоразнообразия). Повсюду наблюдающиеся изменения окружающей среды ведут к сокращению экологически необходимого видового разнообразия. Серьезные изменения природы России, как и во всем мире, под воздействием антропогенных факторов делают охрану природных ресурсов и разработку путей рационального природопользования задачей государственного значения.

Животный мир России в целом отличается бедностью видового состава. Ограниченностю видов животных в России определяется, в частности, северным положением страны. Ее большую часть (свыше 75 %) занимают сравнительно однообразные ландшафты таежной и тундревой зон с бедным видовым составом.

Животный мир России изучен еще не достаточно хорошо. В первую очередь это касается фауны беспозвоночных животных. В настоящее время можно говорить только о приблизительном числе видов беспозвоночных фауны России — 130—150 тыс.; среди них простейшие 6,5 тыс., мезозой — 19, губки — 350, кишечнополостные — 450, плоские черви — 1,9 тыс., круглые черви — 2 тыс., немертины — 100, колчватые черви — 1 тыс., форониды — 5, мшанки — 500, плеченогие — 23, моллюски — 2 тыс., ракообразные — 2 тыс., паукообразные — 10 тыс., насекомые — 100 тыс., иглокожие — 280, щетинкочелюстные — 10, погонофоры — 19, полуходкоподые — 3, что составляет менее 10 % их мирового разнообразия. Основу составляют насекомые (97 % всех видов).

Фауна позвоночных животных России достаточно хорошо исследована и насчитывает 1513 видов, принадлежащих к семи классам: млекопитающие — 320, птицы — 732, рептилии — 80, земноводные — 29, рыбы пресных вод — 343 и 9 видов круглоротых. Кроме того, в морях, омывающих Россию, встречается около 1,5 тыс. видов морских рыб. Таким образом, фауна позвоночных России включает около 3 тыс. видов, что составляет менее 7 % мирового разнообразия. Представительна фауна круглоротых — 45 % всех видов. Наибольшее видовое разнообразие характерно для регионов Северного Кавказа, юга Сибири и юга Дальнего Востока. Относительно высокое — для центральных и южных районов европейской части страны в зонах широколиственных лесов и степей.

Круглоротые представлены в России девятью видами. Состояние всех видов миног, обитающих в европейской части страны, вызывает серьез-

ные опасения, и они нуждаются в охране. Промысловое значение имеют каспийская (*Caspiomyzon wagneri*) и речная (*Lampetra fluviatilis*) миноги.

Фауна рыб России разнообразна и еще относительно слабо изучена, это относится, в частности, к таким отрядам, как, например, лососеобразные (*Salmoniformes*), карпообразные (*Cypriniformes*). Фауна рыб составляет около 7 % мировой ихтиофауны. Среди пресноводной фауны достаточно много эндемиков. По числу эндемиков лидирует бассейн озера Байкал. Наибольшее видовое разнообразие характерно для ука занного региона и бассейна реки Амур.

Современное состояние целого ряда видов, подвидов вызывает серьезное опасение как в связи с нарушением состояния водной среды (загрязнение водоемов, зарегулирование стока рек), так и в связи с высоким уровнем промысла, включая браконьерство. Это в полной мере относится практически ко всем видам осетровых. В России сосредоточены основные мировые запасы рыб этого семейства и значительная часть ресурсов лососевых и карловых рыб. Рыбный промысел занимает одно из важнейших мест в экономике страны. К наиболее важным и экономическому плану относятся, в первую очередь, все осетровые, большая часть лососевых, ряд окуневых и карловых рыб.

Фауна амфибий России составляет всего 0,6 % мирового разнообразия этого класса (29 видов). Эндемичных видов нет. Около 15 % видов занесены в Красную книгу России. Экономическое значение земноводных невелико.

Фауна рептилий России также немногочисленна (80 видов), что определяется достаточно суровыми климатическими условиями на большей части территории, и составляет приблизительно 1,2 % мирового разнообразия пресмыкающихся. Эндемичных видов нет. Наибольшее видовое богатство наблюдается на юге Дальнего Востока и на Кавказе. Около 15 % видов относятся к редким и находящимся под угрозой исчезновения. И более половины видов рептилий отмечены на территориях заповедников.

Хозяйственное значение большинства видов связано с их коммерческой ценностью на мировом рынке диких животных. Это в первую очередь ставит под серьезную угрозу черепах и змей.

Фауна птиц России достаточно хорошо изучена (732 вида) и составляет 7,6 % мировой орнитофауны при практически полном отсутствии эндемичных видов. Подавляющее число видов (515) — гнездящихся и 27 видов гнездится только в пределах России. Среди гнездящихся лишь 83 % видов встречается на территориях заповедников. Наиболее многочисленные отряды воробьинообразных (*Passeriformes*), ржанкообразных (*Charadriiformes*) и гусеобразных (*Anseriformes*). Наибольшее экономическое значение имеют водоплавающие птицы — утки (Лпы), гуси (*Anser*) и курообразные (*Galliformes*), являющиеся важнейшими объектами охоты. Тревогу вызывает, в первую очередь, состояние гусеобразных (*Anseriformes*), гнездящихся в тундровой, лесотундровой, лесной и степной зонах, а также журавлеобразных (*Gruiformes*).

Млекопитающие — 320 видов, что составляет около 7 % общего числа видов этого класса. Териофауна не отличается высоким эндемизмом.

мом. Наиболее богатвидами отряд грызунов (Rodentia). Около 23 % видов млекопитающих занесено в Красную книгу России.

К наиболее редким относятся ряд видов китов и подвиды крупных кошек из рода *Pantera*. Следует особо отметить, что некоторые из видов, которые редки в странах Европы, на территории России широко распространены и имеют высокую численность, например бурый медведь и волк.

В морях и внутренних водоемах России постоянно обитает или встречается во время миграций 56 видов морских млекопитающих, в том числе 40 видов китообразных, 15 — ластоногих и представитель семейства куных — калан. Около 50 % видов морских млекопитающих и некоторые их локальные популяции внесены в Красную книгу России.

Среди природных ресурсов России видное место занимают охотничье-промысловые животные. По их запасам Россия занимает первое место в мире: в нашей стране обитает около 70 видов охотничих зверей, около 150 видов пернатой дичи, более 250 форм промысловых рыб. Использование этих природных богатств дает нашей стране большое количество различных пищевых продуктов и технического сырья, используемых для нужд населения.

Среди них наибольшее значение имеют широко распространенные и многочисленные виды копытных — лось (*Alces alces*), косуля (*Capriocoris capriolus*), кабан (*Susscrofa*), благородный олень (*Cervus elaphus*); хищных — волк (*Canis lupus*), рысь (*Lynx lynx*), бурый медведь (*Ursus arctos*); около 20 видов пушных зверей — соболь (*Malreszibolina*), лесная куница (*M. martes*), лисица (*Vulpes vulpes*), песец (*Alopex lagopus*) и другие виды.

Фауна России менялась и меняется, главным образом беднея в результате смены ландшафтов, среди обитания, хозяйственной деятельности человека. Можно вспомнить о судьбе богатейшей фауны русских и южносибирских степей: исчезновение под влиянием распашки и разведения скота степных популяций тарпанов, куланов, зубров, оленей, кабанов, медведей, населявших еще относительно недавно междуречья Днепра, Дона, Волги; совсем недавнее коренное преобразование степных районов в результате «освоения целины». Природное фаунистическое богатство нашей страны подвергалось порой хищнической эксплуатации, приведшей к его заметному истощению. В течение XIX и XX вв. в нашей стране в результате перепромысла и отсутствия охраны были и почти полностью истреблены морские выдры — каланы, речные бобры, выхухоли, зубры, сайгаки, сильно сократились запасы соболя, европейской норки, выдры и ряда других зверей. Во многих районах страны наблюдалось резкое уменьшение количества пернатой дичи и сокращение уловов рыбы.

В последние годы ведется большая работа по перестройке использования ресурсов животного мира, когда эксплуатация запасов различных животных сочетается с широкими мероприятиями по их охране и восстановлению численности. Человек постепенно утрачивает сознание того, что природа безгранично возобновляема. Даже обширные территории заповедных мест не застрахованы от вреда, наносимого

природе за пределами этих границ. Например, изменения климата захватывают и охраняемые территории, что приводит к смещению экологического баланса.

Для сохранения природного ландшафта в разных странах приняты законы, регламентирующие охоту, ловлю и другие виды деятельности человека в целях уменьшения ущерба, наносимого территориям и животным. Но эти законы действуют только там, где наряду с этим создаются и органы контроля за их осуществлением.

Международные усилия по сохранению биоразнообразия продолжаются всего около 100 лет. В 1902 г. в Париже рядом стран была под писана Международная конвенция по охране птиц, которую можно считать первым международным соглашением по охране биоразнообразия. В 1948 г. был создан Международный союз охраны природы (МСОП, IUCN) — международная неправительственная организация при ЮНЕСКО с консультативным статусом, которая объединяет различные организации почти 150 стран мира.

В 1992 г. в Рио-де-Жанейро принимается Конвенция о биологическом разнообразии. В 2001 г. на Всероссийском форуме по сохранению живой природы России принимаются «Национальная стратегия сохранения биоразнообразия России» и Национальный план действий по сохранению биоразнообразия России.

Для спасения находящихся под угрозой исчезновения видов животных используют различные способы, заключают различные соглашения с международными организациями. Одним из методов сохранения является зоокультура, а именно разведение, в том числе и редких видов, в искусственных условиях. Для этого животных отлавливают и содержат в специальных питомниках или центрах. Могучие зубры (*Bison bonasus*), прежде широко распространенные в лесах Европы, к началу XX в. были почти полностью истреблены. После Великой Отечественной войны они сохранились лишь в зоопарках. Многие страны участвовали в спасении зубра, и ныне стада этих животных вновь пасутся в ряде заповедников. Другой пример подобного рода — судьба антилопы орикс (*Oryx gazella leucoryx*), когда-то распространенной в Саудовской Аравии. Отловленных особей разместили в ряде питомников, поскольку единственная наиболее крупная популяция ориксов в естественные условиях численно была слишком мала для поддержания вида. Но такие работы можно проводить только с видами, которые легко примикиают к неволе и могут размножаться в этих условиях. Этот путь в ряде случаев дает возможность быстрого восстановления численности тех видов животных, которые в некоторых местах естественного ареала уничтожены. Однако переселять можно только те виды, которые ранее уже тут обитали, и только в тех случаях, если местная популяция (≥ 5) этого пополнения не смогла бы выжить.

Все эти мероприятия по охране ценных животных, совмещенные, работой по их расселению в местах прежнего обитания, дали уже вполне ценные результаты. Так, например, еще в XVIII в. соболь населяя всю тайгу Дальнего Востока, Сибири и северо-восточных областей см

ропейской части страны. Нов результате усиленного промысла к началу XX в. соболь во многих местах был истреблен, а в других сохранился лишь в небольшом количестве. Благодаря принятым мерам по охране и расселению в места прежнего обитания численность зверька удалось восстановить в настоящее время до уровня примерно 1,2 млн особей. К началу прошлого столетия речные бобры сохранились в количестве менее 1 тыс. голов в глухих местах Белоруссии, в Воронежской губернии, на Северном Урале и в Туве. В настоящее время колонии бобров можно встретить от западных границ и Карелии до Амура. Численность их настолько возросла, что уже несколько лет во многих областях ведется добывка этих ценных зверей.

Успешно прошло восстановление ареала и численности такого ценного копытного животного, как лось. В результате охраны численность лосей в России достигла 600 тыс. голов, что позволило вести их регулярный промысел. Проводятся работы по увеличению численности кабанов, благородных оленей, косуль в охотничьих хозяйствах центральных и западных областей России.

Широкие работы проведены также в целях расширения ареалов ряда промысловых зверей и птиц отечественной фауны. Так, заяц-русак прежде встречался только в европейской части России и в Зауралье. Но в последние годы он был расселен во многих местах Южной Сибири, в некоторых из них уже стал объектом охоты. Пятнистые олени, дающие ценные панты (молодые рога), обитали только в Уссурийском крае. Теперь их завезли в ряд заповедников и охотничьих хозяйств европейской части России. В некоторых из них хорошо прижились также маралы.

Проводится расселение по охотничьям хозяйствам разных областей России фазанов и других ценных охотничьих птиц.

Обогащение видового состава фауны нашей страны в начале прошлого столетия проводилось и путем акклиматизации животных, завезенных из других стран. Это касается в частности акклиматизации пушных зверей. Как правило, такие работы не принесли ожидаемого положительного эффекта. Успешным можно назвать лишь работы по акклиматизации ценного пушного зверя — североамериканского грызуна ондатры, которая была завезена впервые в нашу страну в 1928 г. И настоящее время она широко расселилась практически на всей территории, ведется ее промысел. Хорошо прижилась в Южной Сибири и на европейской части страны американская норка.

Были проведены удачные опыты завоза некоторых хищных насекомых и клещей для борьбы с различными вредителями сельского хозяйства. Так, были завезены хищные жуки родолия, криптолемус, хилокорус и другие, истребляющие червецов и щитовок, наносящих огромный вред цитрусовым культурам.

В России в первую очередь в охотничьих хозяйствах реализуется программа работ по улучшению условий существования различных 11 чистых животных. Численность последних нередко ограничивается запасами кормов в местах их обитания. Поэтому охотничьи хозяйства

ва проводят подкормку диких копытных, пушных зверей и пернатой дичи в то время года (обычно зимой), когда недостаток кормов ощущается особенно остро. Улучшение кормовой базы зверей и птиц достигается также посадкой или посевами различных растений.

Увеличение численности животных может быть достигнуто также улучшением условий их гнездования или норения. Так, количество уток-крякв на водоемах возрастает, если по берегам поставить особые шалаши или ящики и другие искусственные гнездилища. Численность певчих птиц увеличивается при развешивании дуплянок и скворечников.

Формы охраны природы, применяемые в России, многообразны. Н в различных частях страны, где сохранились особо ценные природные комплексы, организованы государственные заповедники. Первые из них были созданы еще в начале прошлого века, в 1916 г — «Баргузинский» и «Кедровая падь». В 2000 г. было 99 заповедников, самые большие из которых — «Большой Арктический» и «Командорский» имени соответственно площадь 4 169 222 и 3 648 679 га. В заповедниках запрещено добывать животных, рубить лес, пасти скот и чем-либо иным нарушать облик местной природы. В них ведется научная работа, в частности по изучению биологии животных. Заповедники играют большую роль в деле охраны и обогащения нашей фауны. Они являются сохраненными эталонами природы различных ландшафтных районов России.

В последние годы в нашей стране начато создание сети национальных природных парков, которые, с одной стороны, должны сохранять природу различных районов, а с другой — служить местом массового туризма и отдыха. Первые парки — «Лосинный Остров» и «Сочинский» — были созданы в 1983 г. В 2000 г. национальных парков было уже 35, площадь самого крупного «Югыд ВА» составляет 1 891 704 га.

Применяют и другие формы сохранения биоразнообразия. Для введения становления запасов охотничьих зверей и птиц устраивают охотничьи заказники, в которых временно запрещается либо всякая охота, либо добывание определенных животных. В России их примерно 1,5 тыс

Одним из путей сохранения биоразнообразия является создание и ведение Красных книг. В 1963 г была издана первая Красная книга МСОП (Red Data Book). Два тома представляли собой сводку о 211 таксонах млекопитающих и 312 таксонах птиц. В настоящее время под угрозой исчезновения находятся уже тысячи видов животных. В Красный список МСОП (2000 г.) занесено более 9 тыс. видов живиных. С 1600 г. зарегистрировано исчезновение 484 видов животных.

Красная книга МСОП охватывает животный мир в глобальном масштабе. Необходимым дополнением к Красной книге МСОП стали международные Красные книги. Первая Красная книга СССР появилась в 1978 г.

Решение о создании Красной книги РСФСР было принято в 1978 г., а опубликована она была в 1983 г. В нее было занесено 65 видов млекопитающих, 107 видов птиц, 11 видов рептилий, 4 вида амфибий, 9 видов рыб, 15 видов моллюсков и 34 вида насекомых.

В 1997 г был утвержден новый Перечень объектов животного мира, занесенных в Красную книгу России. В него вошли новые типы и классы животных — кольчатые черви (13 видов), мшанки (1 вид), плечоногие (1 вид), круглоротые (4 вида). Число видов редких млекопитающих увеличилось на 7, птиц — на 14, рептилий — на 10, земноводных — на 4. Список редких рыб стал больше в 4 раза, а беспозвоночных животных — в 3 раза.

Красная книга Российской Федерации является официальным документом, содержащим свод сведений о редких и исчезающих видах животных и растений, а также необходимых мерах по их охране и восстановлению. Число редких и находящихся под угрозой исчезновения видов позвоночных России свидетельствует в целом о неблагоприятном состоянии фауны.

Имеются также региональные книги по некоторым административным единицам. Охрана животных всегда должна быть сосредоточена на конкретных популяциях и территориях.

Идея Красной книги за 50 лет ее существования как никакая другая, имеющая отношение к охране природы, стала популярна и понятна. Многие заповедники и другие охраняемые территории были организованы ради сохранения редких животных и растений, а необходимость сохранения редких видов зачастую диктует принятие тех или иных хозяйственных решений. В конечном итоге именно отношение к проблеме сохранения редких животных и растений фактически стало зеркалом всей государственной деятельности в области сохранения биоразнообразия.

ЛИТЕРАТУРА

- Варне Р., Кейлоу П., Олив П., ГолингД.* Беспозвоночные: Новый обобщенный подход: Пер. с англ. — М.: Мир, 1992.
- Веселое Е. А., Кузнецова О. Н.* Практикум по зоологии. — М.: Высш. шк., 1979.
- Догель В. А.* Зоология беспозвоночных. — М.: Высш. шк., 1981.
- Иванова А. В., Полянский Ю. И., Стрелков А. А.* Большой практикум по зоологии беспозвоночных: В 3 т. — М.: Высш. шк., 1981.
- Иорданский Н. Н.* Развитие жизни на Земле. — М.: Просвещение, 1981.
- Карташов Н. Н., Соколов В. Е., Шилов И. А.* Практикум по зоологии позвоночных. — М.: Высш. шк., 1981.
- Константинов В. М., Наумов С. П., Шаталова С. П.* Зоология позвоночных. — М.: Издательский центр «Академия», 2000.
- Кузнецов Б. А., Чернов А. З., Катонова Л. Н.* Курс зоологии. — М.: Агропромиздат, 1989.
- Кэрролл Р* Палеонтология и эволюция позвоночных: В 3 т. — М.: Мир, 19М
- Левушкин С. И., Шилов И. А.* Общая зоология. — М.: Высш. шк., 1994.
- Лукин Е. И.* Зоология. — М.: Агропромиздат, 1989.
- Наумов Н. П., Карташев Н. Н.* Зоология позвоночных: В 2 т. — М.: Высш. шк., 1979.
- РомерА., Парсонс Т* Анатомия позвоночных: В 2 т. — М.: Мир, 1992.
- Фролова Е. Н., Щербина Т. В., Михина Т. Н.* Практикум по зоологии бессип позвоночных. — М.: Просвещение, 1985.
- Хадорн Э., Венер Р.* Общая зоология. — М.: Мир, 1989.
- Шарова И. Х.* Зоология беспозвоночных. — М.: Гуманитарный издательский центр «ВЛАДОС», 1999.

УКАЗАТЕЛЬ РУССКИХ НАЗВАНИЙ И ТЕРМИНОВ

- Автотрофы 5
Агамы 368
Адолескарии 83
Листообразные 399
Аисты 399
Акрон 162, 167
Актиния 71
Актиноспоридии 48
Аксолотль 354
Акула 313
Аллантоис 305, 356
Альвеококк 94
Амбистома 354
Амбулакральная система 272, 274
Амбоциты 60, 276
Амебы 33
Аметаморфоз 225
Амнион 225, 304, 356
Амфибии 294, 343
— безногие 353
— бесхвостые 353
— хвостатые 353
Амфитrite 130
Анабиоз 105, 116, 339
Анапсида 356, 365
Антеннулы 167
Антилопа канна 430
Анус 271
Апантелес 269
Апикомплексы 24, 38
Аппарат Гольджи 11, 12
— коловоращательный 125
Аппендикулярии 286, 293
Аптерии 398
Ареал 23, 469
Архар 443
Археоптерикс 404
Архозавроморфы 372
Архозавры 356, 370
Арцелла 35
Аскарида свиная 116
— человеческая 116
Аскаридия 120, 121
Аскариды 116, 119
Аскон 59, 60
Аспиды 368
Асцетоспоридии 24
Асцидии 286, 290
АтTRACTАНты 220
Аурелия 68, 69
Аутогеморрагия 222
Аутомия 188, 276
Афеленхиды 104
Афеленхойд рисовый 104, 105
Афелинус 245
Бабочки 250, 253
— равнокрылые 254
— разнокрылые 254
Бабочницы 259
Балантидий 52
Бегемоты 440
Бедро 192, 214
Беззубки 154
Безоаровый козел 443
Беличьи 432
Белка 425
Белуга 334
Белянка капустная 258
Белянки 258
Бессяжковые 208, 209
Бесчелюстные 294, 306
Бесчерепные 286

- Биотический потенциал вида 459
 Биотоп 461
 Биоценоз 461
 — видовая структура 461
 Битиния 84, 148
 Бластодерма 17, 225
 Бластомеры 16
 Бластопор 18, 271
 Бластоцель 17
 Бластула 16
 Блоха человеческая 253
 Блохи 249, 253
 Блошки огородные 252
 Блюдечко морское 147
 Бобр 425
 Богомол грациозный 239
 — обыкновенный 238
 Богомоловые 234, 238
 Божки коровки 252
 Бородавки паутинные 188
 Ботрий 87, 95
 Бражники 258
 Булавоусые 258
 Бульбус 100, 102, 120
 Бурса 98
 Бык 443
 Бычковые 338
 Бычок 339
 Вакуоли пищеварительные 27, 51
 — сократительные 28, 51
 Вараны 368
 Веретеница 366
 Вертлуг 192, 214
 Вертячка овец 93
 Вид 6, 23
 Влагалище 48
 Власоеды 242, 239
 Водолюб черный 251
 Водяные ослики 177
 Волк 434
 Волнянки 256
 Вольвокс 58
 Воробей 397
 Воробышкообразные 404
 Ворона 397
 Восьминогие 160
 Вошь головная 242
 карповая 176
 — лобковая 242
 — платяная 242
 Вредная черепашка 249
 Вторичная полость 18
 Вторичнополостные 18, 19
 Вторичноротые 18, 271
 Вши 241, 238
 Вши китовые 178
 Выдра 435
 Вяхирь 402
 Гага 396
 Гагара 396
 Гадюка 366
 Галлицы 260
 Галлы 103, 109
 Гаметы половые 17, 32
 Гамонт 39
 Ганоидные 311, 334
 Гастропор 18
 Гастроцель 18, 32
 Гаструла 17, 29
 Гаттерия 366
 Геккон серый 366
 — сцинковый 366
 Гекконы 368
 Гелиометра 278
 Гельминты 116
 Гемиметаморфоз 226
 Геммула 60
 Гемоглобин 132, 136, 171
 Гемолимфа 166, 171, 221, 222
 Гессенский комарик 260
 Гетеродериды 105
 Гетеротрофы 5, 12
 Гидранты 66
 Гидроидные 65
 Гидроидные медузы 67, 68
 Гидрорецепторы 219
 Гидроскелет 97
 Гидры 65
 Гиподерма 98
 Гипопус 193
 Гирудин 141
 Гистогенез 227
 Гистолиз 227
 Глаза простые 164, 218
 — сложные 164, 218
 Глазки Гессе 287

- Глохидий 145, 152
Глухарь 396
Гниды 242
Голень 192, 214
Головогрудь 162, 167, 179
Головохордовые 286, 287
Голометаболизм 226
Голубеобразные 402
Голубь 397
Гольян 328
Горбуша 336
Горилла 430
Горлица 403
Горностай 435
Гребешки 154
Гребневики 24, 73
Грегарини 39, 52
Грибы 5
Грифельки 216
Грызуны 431
Губки 24, 59
Губоногие 207
Гуматы 138
Гумус 138
Гусеница 227
Гусеобразные 399
Гусь 396
Дактилогирус 86
Дафния 173
— обыкновенная 173
Двоякодышащие 311
Двукрылые 255, 259
Двупарногие 205, 207
Двустороннесимметричные 24
Двуустка кровяная 85
Двухвостки 209
Дейтомерит 39
Дейтонимфа 193
Дельфин 430
Деляфондии 122
Джейран 430
Диапауза 194
Диаптомусы 175
Дигенетические сосальщики 75
Диффлюгия 35
Длинноусые 259
Долгоносик свекловичный 252
Долгоносики 252
- Дорилаймиды 104
Древнекрылые 234
Дрейссена 154
Дыхальца 165
Дятел зеленый 403
— пестрый 403
Дятлообразные 403
Елец 328
Емкость среды 459
Ехидна 425
Жаба 354
Жабродышащие 162, 167
Жаброногие 173
Жабры 129, 132, 144, 171, 222
Жабы 353, 354
Жаворонок 397
Жало 216
Жвалы 153, 170, 210
Жвачные 438, 440
Жгутики 26, 27, 59
Жгутоконосцы 28
Железы антеннальные 172
— коксальные 135, 184
— максиллярные 172
— паутинные 183, 188
Желтопузик 366
Желуди морские 176
Жемчужница морская 153, 154
— речная 154
Жерех 327
Жестокрылые 178
Животные 5
Жигалка осенняя 263
Жировое тело 223
Жужелицы 250
Жужжалы 261
Жужжалыца 214, 259
Жук колорадский 252
— носорог 251
Жуки 249
— пластинчатоусые 251
— разноядные 251
— хищные 250
— хлебные 251
— щелкуны 251
Журавлеобразные 401
Журавль 397, 401
Журчалки 262
Зайцеобразные 431