Федеральное агентство по образованию
Государственное образовательное учреждение
высшего профессионального образования
«Омский государственный технический университет»

А.Н. ФЛОРЕНСОВ

ОПЕРАЦИОННЫЕ СИСТЕМЫ

для программиста

Учебное пособие

ОМСК 2005

УДК 004.4(075)
ББК 32.973.73-018.2я73
Ф 73

Рецензенты: В. Т. Гиль, канд. техн. наук;
М. Ф. Шакиров, канд. техн. наук

Флоренсов А. Н.
Ф 73 Операционные системы для программиста; Учеб. пособие. Омск: Изд-во ОмГТУ, 2005. -240 с.

Рассматривается программирование, зависящее от особенностей конкретных широко используемых операционных систем. Изучаются средства создания и управления программными процессами и нитями внутри них, а также разнообразные средства обеспечения взаимодействия. Ориентировано на понимание роли системных объектов и средств доступа к ним. Изучение ведется параллельно в операционных системах MS Windows и Linux. Содержит множество небольших примеров программ.

Предназначено для подготовки студентов и бакалавров направления «Информатика и вычислительная техника» по дисциплине «Операционные системы».

Печатается по решению редакционно-издательского совета ОмГТУ.

УДК 004.4(075)
БК 32.973.73-018.2я73
Ф 73

Редактор В.А. арколева
ИД №06039 от 12.10.2001 г.
Сводный темплан 2005 г.
Подписано в печать 1.08.05. Формат 60´84 1/16. Отпечатано на дупликаторе.
Усл. печ. л. 15,0 Уч.-изд. л. 15,0. Тираж 200 экз. Заказ

Издательство ОмГТУ. 644050, г. Омск, пр-т Мира, 11
Типография ОмГТУ

ВВЕДЕНИЕ

С учетом широчайшего использования компьютеров в современной жизни операционные системы (ОС) целесообразно рассматривать с нескольких различных сторон. Во-первых, со стороны пользователя, не умеющего программировать. Таких пользователей подавляющее большинство, и наиболее употребительные ОС рассчитаны именно на них. Управление компьютером для этих пользователей в значительной степени подобно управлению аппаратурой телевизора с помощью дистанционного пульта. В обоих случаях присутствует множество возможностей, как основных, так и более тонких. Управление компьютером на уровне пользователей можно назвать уровнем домашних хозяек, хотя за ними традиционно закрепилось название “чайников”. Для них выпускается соответствующая литература, иногда использующая указанную терминологию. Пользовательский уровень управления компьютером основан на двух типах средств: действиях с помощью манипулятора мыши и на задании действий в пунктах диалоговых окон. Последние действия обычно выполняются также с помощью мыши или, что реже, набором кратких текстов в полях окон диалога. Для огромного функционального многообразия современной ОС набор таких действий чрезвычайно велик и практическое освоение их оказывается достаточно трудоемкой задачей. Заметим, что практики, владеющие значительной частью этого набора, пользуются всеобщим уважением среди непрофессиональных пользователей компьютеров.

Другая важная сторона, с которой рассматриваются операционные системы, – это внутреннее строение и разработка ОС. Детальное изучение внутреннего строения нужно в основном только разработчикам таких систем. В предшествующий период (в 70-х и 80-х годах XX века) этой стороне уделялось большое внимание. Оно было частично обусловлено сложностью практического использования ОС предыдущих поколений, когда даже от прикладного программиста требовались не очень элементарные представления о внутреннем функционировании ОС. Последнее было связано с чрезвычайно высокой стоимостью компьютерных систем и вызывало необходимость всеми средствами повысить эффективность использования компьютеров (час времени работы на компьютере стоил порядка 100$.) Практиче-

ски все книги по операционным системам, написанные в указанные годы, рассматривали операционные системы именно с этой стороны.

К настоящему времени ОС настолько усложнились внутри, что изучение их строения является поистине титанической задачей. Отдельные попытки “укрупненно” описать их строение являются чисто декоративными и практически бесполезными как для поверхностных пользователей, так и для большинства достаточно грамотных программистов. Такое описание вынужденно концентрируется на одном выбранном типе ОС. Поэтому на данной традиционной стороне описания ОС останавливаться не будем, оставим его заинтересованным для самостоятельного овладения, т.к. по объему оно многократно превышает возможности данного курса.

Наконец, третьей стороной изучения ОС является рассмотрение ее со стороны программных функций, вызываемых как подпрограммы. Причем имеются в виду вызовы подпрограмм, доступные людям, которые не являются разработчиками этой ОС. Совокупность таких программных функций принято называть программным интерфейсом пользователя или API (Application Program Interface). Программный интерфейс пользователя является границей между внутренней частью ОС и программами, использующими средства ОС. Он достаточно подробно описывается в руководствах программисту по операционной системе. Именно этой стороне изучения операционных систем и будет в основном посвящено учебное пособие. При этом для понимания действий отдельных функций API будем частично рассматривать некоторые особенности реализации функций внутри ОС. Такая необходимость появляется, когда функции API используют некоторые внутрисистемные структуры данных, без описания строения и взаимосвязи которых действия функций понять невозможно.

Для содержательного освоения предмета без значительного увеличения числа учебных и демонстрационных задач используется параллельное рассмотрение нескольких ОС. Изложение материала ведется на основе двух типов наиболее распространенных в настоящее время ОС: семействе Windows (Windows 2000, Windows NT, Windows 95 и Windows 98) и Unix. Кроме того, в отдельных примерах будут упоминаться технические решения, использованные в операционной системе OS/2. Это позволяет продемонстрировать, не только то, как следует применять специфические средства конкретной ОС (последнее важнее на этапе совершенствования в программировании), но и то, что требуется отобразить в конструкциях программы для решения простой конкретной задачи, использующего мощные средства многозадачной ОС. В конечном счете такой подход нацелен на формирование понятийной базы для полноценного использования операционных систем при разработке программ.

1. ОСНОВНЫЕ ПОНЯТИЯ

1.1. Понятие операционной системы

Операционной системой (ОС) называют комплекс программ, обеспечивающих взаимодействие прикладных программ и пользователей с аппаратурой компьютера. Аппаратура современных компьютеров настолько сложна и многофункциональна, что управление ею непосредственно из прикладной программы катастрофически сложная задача. Не касаясь малопонятных в начальном знакомстве проблем, можно отметить сложность программного управления современными внешними устройствами. Это управление требует понимания действий десятков управляющих кодов отдельного устройства. В конечном счете требует глубокого понимания функционирования сложного электронного или электромеханического устройства. Ясно, что решение соответствующих задач управления прикладному программисту и пользователю приходится перекладывать на разработчика операционной системы.

Среди многообразия функций операционной системы можно выделить главные:

1) обслуживание информации, хранимой в компьютере;
2) программное управление устройствами компьютера;
3) обеспечение простого информационного диалога компонентов ОС с пользователем или прикладной программой;
4) обеспечение эффективного использования вычислительной системы путем совместного задействования общих ресурсов или одновременного выполнения нескольких прикладных программ.

По причине небольшого числа используемых операционных систем классификация ОС основана на наличии или отсутствии существенных особенностей из небольшого набора последних.

По возможности обеспечивать одновременную работу многих пользователей различают *однопользовательские* и *многопользовательские* ОС. По способности выполнять одновременно более одной задачи их классифицируют как *многозадачные* или *однозадачные*. По числу процессоров, управляемых ОС, различают *однопроцессорные* и *многопроцессорные*. Относительно числа отдельных компьютеров ранее различали одномашинные и многомашинные ОС. К настоящему времени термин многомашинные ближе всего по значению к признаку *сетевые* ОС, хотя исторически и не совпадает с ним. Кроме того, различают *аппаратно зависимые*, в более позднем обозначении *зависимые от платформы* операционные системы, и *мобильные* ОС. Наиболее общим делением является разбиение на *универсальные* и *специализированные*. Последние по существу почти полностью совпадают с управляющими ОС. Основной задачей управляющих является обслуживание аппаратных и технологических комплексов в автоматическом режиме. Поэтому характерными особенностями таких ОС является включение в них средств, обеспечиваю-

щих особо высокую надежность функционирования и быстроту реакции на отдельные виды внешних событий, воздействующих на вычислительную аппаратуру электрическими сигналами специального назначения. Конструирование управляющих ОС является предметом особого внимания разработчиков современного технологического оборудования, военной и космической техники.

До недавнего времени ОС по типу используемого в них интерфейса с пользователем разделяли на текстовые и графические. К настоящему времени все современные ОС включают графический интерфейс либо как основной (в системах фирмы Microsoft), либо как дополнительную интерфейсную оболочку (в системах Unix).

1.2. Системные соглашения для доступа к функциям ОС

К настоящему времени в России относительно широко используются следующие типы операционных систем: Windows 9x, Windows NT, различные клоны семейства Unix. К первому типу относятся ОС Windows 95, Windows 98 и Windows Millenium. Последние модификации типа Windows NT называются Windows 2000, Windows XP и Windows Server 2003. Операционная система Unix в основном представлена в России клоном BSD (Berkley Software Distribution) и различными модификациями ОС Linux (Red Hat, Debian, Slackware, Mandrake и т.д.). Привлекательным достоинством ОС Linux является ее бесплатность для пользователей. (Строго говоря, с точки зрения профессионалов, Linux не является разновидностью Unix, но их различия касаются только внутреннего построения этих ОС, с точки же внешнего взаимодействия с программами в текущем курсе изучения их можно считать одинаковыми.)

Особое место до недавнего времени занимала OS/2. С одной стороны, она оказалась жертвой конкурентной борьбы с фирмой Microsoft, с другой – это единственная ОС, являющаяся одновременно и современной, и простой для профессионального изучения. Практическое неудобство OS/2, проявившееся в последние годы, заключается в прекращении поддержки ее производителями видеокарт, в результате чего оказывается невозможным использование режимов высокого разрешения и частоты на современных видеосистемах с этой ОС. Операционная система OS/2 по своей функциональной структуре занимала место между простой MS DOS и современными Windows, причем по простоте ближе к первой, а по возможностям – ко вторым. Поэтому изучение OS/2 есть простейший и кратчайший путь, чтобы разобраться в функционировании операционных систем. Многие технические решения, использованные в OS/2, многократно проще, чем примененные в Windows, причем при тех же функциональных возможностях.

Изучение лишь одного из известных технических решений приводит к малограмотности инженера. Поэтому изучение операционных систем в данном книге

будет базироваться на рассмотрении основных решений для двух операционных систем: Unix и Windows. В отдельных случаях будут упоминаться соответствующие средства для OS/2. Заметим, что в OS/2, как правило, имеют место простые и прозрачные для начального знакомства решения, но практическое значение этой системы приближается к нулю. В Windows мы сталкиваемся с самыми громоздкими и часто очень не безопасными в использовании средствами, но практическое использование ее чрезвычайно широко среди неквалифицированных масс. В Unix знакомимся с классическими решениями, относительным недостатком которых является только некоторый архаизм, неизбежно заложенный в начале 70-х годов. (Unix – самая древняя из современных ОС, но не теряющая свое совершенство для профессионалов.)

Программный интерфейс пользователя для современных ОС описывается исключительно на языке С. Причинами этого являются как заложенные в основных ОС мобильность (Unix, Windows), так и удобство С в качестве языка системного программирования. Заметим, что более ранние ОС имели API, описываемый исключительно на ассемблере, что крайне мешало переносу таких ОС на иные типы процессоров и повышало трудоемкость разработки самих ОС. Тем не менее, критические части современных ОС целесообразно записывать на ассемблере. (Заметим, что даже в громоздкой системе программирования MS Visual C++ содержится более двух сотен килобайтов ассемблерных текстов для подпрограмм, критически влияющих на производительность приложений.) Поэтому в данном пособии все изложение будет основываться на языке С. (Заметим, что описываемое на языке С всегда может быть описано и использовано с помощью языка ассемблера. Такой вариант применяться не будет, так как не предполагается знакомство читателей с программированием на ассемблере.)

Во всех изучаемых здесь ОС обращение к системным функциям (системным подпрограммам), т.е. программным функциям операционной системы выполняется путем вызова этих подпрограмм по имени. (Иной способ, основанный на механизме прерываний, требует использования ассемблера для его записи в исходных программах.) Таким образом, для использования этих функций необходимо найти описание прототипа системной функции, задать в соответствии с ним аргументы и записать синтаксически верный вызов. При внимательном чтении описания функции можно даже достичь семантически правильного задания системного вызова.

1.3. Особенности разработки программ в базовых ОС

Имея в виду учебный характер данного пособия, познакомим читателя с простейшими типовыми приемами разработки программ на языке Си для базовых операционных систем. Эти приемы основываются на вызове средств программирования в режиме командной строки. При таком программировании использу-

ется текстовое окно, часто размером во весь экран, на котором отображается все, что непосредственно вводится с клавиатуры. Программист набирает команды управления операционной системой или вызова служебных программ нажатием клавиш клавиатуры и нажимает клавишу Enter.

Как это ни странно, отказ от полноэкранных систем программирования и графического интерфейса значительно ускоряет разработку простейших программ, нужно лишь обладать необходимым минимумом знаний, чтобы командовать компьютером на этом уровне управления. Более сложные и громоздкие средства разработки на самом деле эффективны только для достаточно больших и сложных программ, а также для малоквалифицированных программистов, которым они в диалоговом режиме помогают разобраться в ряде типовых ситуаций. Лишь отладка программ с помощью специализированных отладчиков требует более мощных и громоздких средств.

Для программирования в операционной системе Windows основными современными средствами программирования на языке Си являются соответствующие программные продукты фирм Borland (более современное название Inprise Inc.) и Microsoft. Современные компиляторы первой из них называются BCC32.EXE, а второй – CL.EXE Исходный текст программы на языке Си может быть подготовлен в любом текстовом редакторе, формирующем чистый текст без всяких атрибутов и стилей. В Windows для этого подойдет редактор "Блокнот" (Notepad.exe), но лучше использовать файловую инструментальную оболочку типа FAR manager или подобную ей. Рекомендуется для простейших программ, использующих программные функции операционной системы, задавать расширение исходных файлов программ, обозначаемое буквой "c", так что типовой пример может быть наименован как prim.c. Создание исполняемой программы с помощью программы BCC32.EXE запишется в командной строке как

bcc32 prim.c

а с помощью программы CL.EXE запишется в командной строке как

cl prim.c

Если использовать систему программирования Watcom фирмы Sybase, приказ на создание исполняемой программы запишется в командной строке как

wcl386 prim.c

Читатель, видимо, уловил правило приказов на создание исполняемых программ: в командной строке записывается имя основной программы системы разработки, за которым через пробел (пробелы) записывается имя исходной программы. При повторных вызовах программ, уже вызывавшихся из командной строки, достаточно один (или более) раз нажать на клавишу клавиатуры (что вызывает в командную строку предыдущую или более раннюю использованную команду), затем следует нажать клавишу Enter (в командных оболочках для вызова предыдущей выполненной команды может использоваться другая управляющая клавиша).

Этот простейший прием в случае ошибок в исходной программе приводит к выводу их непосредственно на текстовый экран (экран консоли), что при многочисленных ошибках может быть неудобно разработчику. Для вывода сообщений об ошибках в файл, имя которого выбрано разработчиком, любую из приведенных выше команд надо дополнить фрагментом вида

>имяфайладляошибок

Например, вызов компилятора BCC32 с помещением информации об ошибках в файл myerrs.lst для исходного текста программы prima.c запишется как

bcc32 prima.c >myerrs.lst

Для использования отладчика подготовительные действия несколько усложняются. Вызов компилятора BCC32.EXE должен содержать опцию (ключ) v, которой должен предшествовать символ дефиса или наклонной черты (символ - или /), так что наш пример приобретает вид

bcc32 -v prim.c

В системе разработки Microsoft отсутствует отладчик, отделенный от интегрированной системы разработки Visual Studio, поэтому желающие могут использовать всю эту систему.

В системе разработки Watcom опцией задания отладочной информации служит буквосочетание d2, поэтому соответствующий вызов компилятора должен иметь вид

wcl386 -d2 prim.c

Для разработок программ в операционной системе Linux служит свободно распространяемый компилятор с общим именем gcc. Его вызов для компиляции исходной программы prim.c задается в командной строке в виде

gcc prim.c

Неожиданностью для программистов, начинающих разработки под Unix, является отсутствие исполняемого файла вида prim.exe. Дело в том, что расширение exe для исполняемых файлов в этой системе не является употребительным, хотя никак и не запрещается. Более того, автоматически создаваемый компилятором исполняемый файл имеет имя a.out (достаточно непривычно для типовых пользователей операционных систем от MS). Чтобы задать любое желаемое имя исполняемого файла, следует воспользоваться опцией -o, которая призвана определять имя выходного (output) файла и за которой через пробел (пробелы) должно следовать это имя. В нашем примере следует набрать командный текст

gcc -o prim.exe prim.c

В стандартной настройке ОС Unix для запуска программы, как правило, недостаточно набрать в командной строке ее имя. Обычно вызов программы из текущего оглавления задается в виде

./имяпрограммы

Причины этого, связанные с особенностями файловой системы и стремлением увеличить защищенность ОС от несанкционированного доступа, будут рассмотрены позже.

Для последующего использования отладчика в вызове компилятора следует задать опцию -g, так что общий вызов будет иметь вид

```
gcc -o prim.exe  -g  prim.c
```

Сам отладчик имеет в этой системе имя gdb, а вызов на выполнение программы под управлением отладчика будет записываться в виде

```
gdb  prim.exe
```

Данный отладчик несет многие консервативные следы предыдущих поколений операционной системы Unix, и его изучение является отдельной темой.

Все разработки в операционной системе Windows требуют подключения заголовочного файла windows.h, в операционной системе Unix в простейших случаях бывает достаточно ограничиться подключением заголовочного файла stdio.h или unistd.h – в современных версиях этих операционных систем.

Более детальная информация будет даваться при рассмотрении примеров.

1.4. Командный интерфейс пользователя в ОС

При работе в режиме разработки и запуска программ, непосредственно использующих системные функции операционной системы, наиболее практичен командный интерфейс. Его относительным недостатком является требование минимальной программистской культуры по использованию основных средств ОС. Эта культура включает навыки управления через основные команды ОС, навигацию по базовой файловой системе ОС и, возможно, использование типовых командных оболочек (коммандеров).

Подобные средства в операционных системах Windows предполагаются известными читателю. Для операционных систем, относящихся к клонам Unix и Linux, краткая информация будет изложена далее. Прежде всего следует иметь в виду, что минимальный набор команд текстового режима во всех упомянутых ОС можно считать одинаковым. К этому набору относятся команды *cd*, *more* и *dir* (последняя является дополнительной командой Linux, в то время как базовая команда вывода оглавления в Unix несколько другая и имеет имя *ls*). Практически все команды, используемые в Windows, присутствовали и в OS/2 (из-за преемственности последней к MS DOS). Для копирования файлов в Unix служит команда *cp*, а для перемещения – команда *mv*.

На первых шагах обучения можно не сосредотачиваться на запоминании команд, специфических для новых читателю операционных систем, но проще использовать типовые командные оболочки. Так, в Linux подобной оболочкой служит вспомогательная программа Midnight Commander, запускаемая командой

(именем программы) *mc*. Во всех упомянутых оболочках нет необходимости явно использовать и помнить написание основных команд, так как эти оболочки поддерживают управляющие клавиши, в частности, клавиши F1 – F10, вызывающие точно те же функции над файлами, что и в Norton Commander.

Для оболочки *mc* надо иметь в виду, что обновление содержимого панели инструментального окна не производится автоматически (как это имеет место в Windows). Например, после компиляции исходного текста программы в панели оболочки *mc* не видно появление исполняемого файла. Обновление панели (аналогично другим базовым оболочкам) происходит здесь по нажатию комбинации Ctrl--r.

Исполняемый файл может быть запущен из панели *mc* непосредственно нажатием клавиши Enter при выделенном в панели названии этого файла. При этом не требуется никаких дополнительных уточнений, относящихся исключительно к командному режиму.

Следует иметь в виду, что в файловых системах Unix нет логических дисков (как понятий и объектов манипуляций). Все файлы собраны в единое связное дерево. Сама файловая система Unix налагает (незаметно для начального пользователя) существенные ограничения на несанкционированный доступ. В частности, обычному пользователю нельзя создавать собственные каталоги нигде, кроме собственного каталога и его подкаталогов (а также внутри каталога /temp). Удаление и изменение файлов в других каталогах ему также не позволяется. При использовании составных имен файлов (абсолютных и относительных) в Unix для разделения имени каталогов и каталога от собственного имени служит символ прямая наклонная черта (символ /) вместо обратной разделительной черты в Windows и OS/2.

Существенной особенностью вывода текстов в Unix на текстовый экран (экран консоли) является выполнение собственно вывода только после поступления на вывод символа конца строки (управляющего символа '\n' в языке Си). Если же начинающий программист забывает это сделать, то вывод осуществляется только после завершения программы. При разработке программ в Unix приходится часто пользоваться управляющей комбинацией Ctrl-C для принудительного прекращения программы, запущенной с той же консоли. Параллельно и независимо друг от друга может быть несколько консолей. Для переключения между ними в Linux служат клавиши Fn (где n – номер консоли), нажимаемые после клавиши Alt.

После запуска операционной многопользовательской ОС (типа Windows NT и Unux) система предлагает ввести системное имя (обозначение) пользователя и затем пароль для входа. В простейшем случае пароль может быть снят (отменен), но ввод системного имени требуется во всех случаях.

Сразу после загрузки операционной системы Unix и ввода системного имени для персонального пользователя ему автоматически предоставляется текстовый

режим работы с ОС (возможен вариант установки Linux, где пользователь сразу же попадает в графический режим, но этот вариант нами рассматриваться не будет). В Windows NT пользователь после ввода пароля попадает в графическую оболочку и требуются некоторые дополнительные усилия, чтобы добраться до текстового интерфейса командного режима. Проще всего сделать это двумя следующими способами. Первый предполагает выбор через главное меню пункта "Командный режим", после чего на экране появляется текстовое окно. Второй способ заключается в активизации командной оболочки, подобной FAR manager, ярлык которой присутствует на рабочем столе или в главном меню. Активизация производится обычным щелчком мыши. Текстовое окно при этом появляется как подложка панелей командной оболочки.

1.5. Информация об ошибках системной функции

Проблемой, почти никогда не возникающей вне использования системных средств, является сама принципиальная выполнимость вызываемой функции. Когда в прикладном программировании задается управляющая конструкция или выражение, то заданные ими действия всегда выполняются по заложенным в них правилам. При обращении же к системным функциям возможные ситуации не столь однозначны. Чтобы их понять, достаточно вспомнить работу с файлами в прикладном программировании (работа с файлом, по существу, неизбежно требует использования средств именно операционной системы). При семантически верном задании открытия файла, он может быть и не открыт, например, когда для чтения открывают файл, который не существует. Значительная часть системных функций совсем не обязательно обеспечивает запрашиваемое действие! Поэтому в API вводятся вспомогательные средства, позволяющие программе обнаружить, выполнилось ли в действительности запрашиваемое действие.

Наиболее традиционно эта проблема решается в Unix. Большинство системных функций Unix имеют вид

int *имяфункции*(*список_аргументов*),

возвращая в качестве основного значения целое число. Это число может нести значение, определяющее результат действия функции, но одно из значений (обычно -1), зарезервировано за условным кодом ошибки. В описании системной функции обязательно присутствует информация, описывающая ошибочные ситуации, которые могут возникать при выполнении функции, и соответствующие им значения *кода возврата* (возвращаемого значения). Описание кодов ошибочных ситуаций в

Unix совершенно обязательно в справочной информации по любой системной функции.

Заметим, что само получение справочной информации по некоторой системной функции в Unix вызывается командой *man* текстового режима. Эта команда является сокращением слова manual и задается в виде

man *имяфункции*

Если по такому вызову появится информация об одноименной команде, то следует использовать вызов справки в форме

man 2 *имяфункции*

Справочная информация появляется в текстовом окне консоли и имеет несколько архаичный вид (без возможности использования гипертекста).

Другой формой контроля ошибок в Unix является специальная переменная *errno*. Для основных функций операционной системы Unix код ошибки неявно заносится в эту специальную глобальную переменную, описанную в заголовочном файле *errno.h*. Такой прием дает принципиальную возможность анализировать значение этой переменной после обращения к системной функции. Но делать это следует сразу же после системного вызова, так как следующий системный вызов, в свою очередь, изменяет эту переменную.

Практическое использование значений переменной ошибок *errno* достигается с помощью вспомогательных функций *perror()* и *strerror()*. Первая из них имеет прототип

void perror(const char *s);

а вторая имеет прототип

char *strerror(int errnum).

Вторая функция возвращает указатель на текст, описывающий ошибку, а первая непосредственно отображает в стандартном потоке ошибок этот текст, причем предваряемый любой пользовательской текстовой информацией, которая содержится в аргументе *s*. Заметим, что к настоящему моменту в операционной системе Linux имеется более двухсот кодов ошибок, которые отображаются этими функциями. Следует также отметить, что указанные функции могут использоваться для локальной диагностики ошибок и в других операционных системах, а именно для программ, написанных на языке С. Такая возможность имеется даже в MS DOS, она присутствует и в OS/2, и в Windows. Но во всех этих последних ОС такая возможность является вспомогательной и не гарантируется для всех системных функций, которые используют свои собственные способы самодиагностики. (В MS DOS и Windows функции, основанные на глобальной переменной *errno*, позволяют определить только около 50 типов ошибок).

В операционной системе OS/2 системные функции делились на несколько функциональных групп, определяемых префиксом их имени. Основные функции имеют имена вида Dos*Собственноеимя*. Функции обслуживания дисплея (видео-

системы) имели имена вида Vio*Собственноеимя*, для обслуживания клавиатуры использовались функции с именами Kbd*Собственноеимя*, а для обслуживания мыши – с именами Mou*Собственноеимя*.

Все основные системные функции возвращали в этой ОС в качестве основного значения функции величину типа APIRET, а функции видеосистемы, клавиатуры и мыши – типа APIRET16. Принято строгое правило, что нулевое значение кода возврата однозначно определяет безошибочное выполнение системной функции. Все иные значения типа APIRET и APIRET16 дают код ошибки. Коды ошибки были описаны в заголовочном файле

bseerrs.h

систем программирования для OS/2, а их описание приведено в одном из файлов справочной системы. Обязательным в этой справочной системе являлось приведение всех возможных кодов ошибки с соответствующими комментариями. Отдельным пунктом справочной подсистемы, являлся пункт с наименованием Errors, которые содержал удобное для программиста соотнесение номеров кодов ошибок их содержательному толкованию.

Наиболее экзотической является получение информации об ошибках в MS Windows. Во первых, отсутствует какое-либо подобие систематичности в системных функциях. Возвращаемые значения системных функций могут быть описаны как VOID, BOOL, HANDLE, PVOID, LONG или DWORD. При использовании типа BOOL возвращаемое значение 0 обозначает ситуацию ошибки (категорически не рекомендуется [13] проверять это значение на TRUE). При типах HANDLE, LONG или DWORD ситуацию ошибки дает значение -1 или 0 в зависимости от конкретной функции. Функции с возвращаемым значением PVOID индицируют ошибку значением NULL.

Для получения же собственно кода ошибки в MS Windows программисту приходится принимать немалые дополнительные усилия. Если по возвращаемому значению системной функции определяется, что ошибка есть, следует немедленно вызывать специальную функцию GetLastError(), которая не имеет аргументов и возвращает значение типа DWORD. Функция GetLastError() возвращает последнюю ошибку, возникшую в ходе выполнения программы (точнее нити программы). Именно это 32-битное значение дает код ошибки. Собственно коды ошибок, общие для всех системных функций, содержатся в заголовочном файле WinError.h.

Отличительной и не очень приятной особенностью MS Windows является отсутствие информации о возможных кодах ошибки для конкретных функций. Известный специалист и один из редакторов журнала "Microsoft Systems Journal" Дж. Рихтер в книге [13] пишет буквально следующее: "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, тако-

го 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 различным 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."

Числовые коды ошибок, возвращаемые функцией GetLastError(), достаточно сложно для разработчика соотнести с наименованием ошибки. Если требуется распознавание вида ошибки при автоматическом выполнении программы, то разработчики этой ОС предлагают для использования специальную функцию FormatMessage.

В конечном счете эта функция делает почти то же самое, что и функция strerror в Unix, но сложность использования функции FormatMessage неизмеримо больше. Прежде всего она имеет прототип

```
DWORD FormatMessage(DWORD dwFlags,   LPCVOID lpSource,
   DWORD dwMessageId,    DWORD dwLanguageId,
   LPTSTR lpBuffer,    DWORD nSize, va_list *Arguments).
```

Здесь не будет рассматриваться все многообразие ее возможностей, которые в большинстве ситуаций излишни (от функции требуется лишь выдать текст, называющий причину ошибки). Главным по значению является третий ее аргумент *dwMessageId*, который и задает распознаваемый код ошибки. Если не использовать "хитрых" возможностей динамического выделения памяти под буфер текста, то адрес буфера для текста задается в аргументе *lpBuffer*, а его размер в аргументе *nSize*. Важным в применении является и первый аргумент, который дает возможность использовать не только системные, но и пользовательские коды ошибок. Для системных ошибок в аргументе *dwFlags* должно присутствовать логическое слагаемое FORMAT_MESSAGE_FROM_SYSTEM. Мощные потенциальные возможности заложены в аргументе *dwLanguageId*, призванном обеспечить многоязыковую поддержку сообщений, но к настоящему моменту для России работающим является только простейшей вариант языка по умолчанию. (В Windows 9x неправильно функционирует даже вариант, принудительно запрашивающий англоязычный американский текст.) Работающий вариант задается макросом MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT). Оставшиеся без рассмотрения аргументы следует брать нулевыми. В результате типовой вызов данной функции имеет вид

```
len=FormatMessage(FORMAT_MESSAGE_FROM_SYSTEM,
     NULL, k, // k - номер ошибки, возвращенный функцией GetLastError()
```

MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT),
txtmess, sizeof(txtmess), NULL);

где имя *txtmess* определено предварительно в описании вида char txtmess[300]. Возвращаемое функцией значение дает число символов в тексте, сформированном ею в заданном буфере.

Функция FormatMessage обладает возможностью возвращать порядка тысячи различных наименований ошибок на языке текущей локализации (т.е. при использовании русифицированной версии Windows – на русском языке). Заметим кстати, что в более точных терминах программный интерфейс MS Windows называется Win32. Существенной особенностью рассматриваемой функции оказывается использование возвращаемых текстов сообщений об ошибках, представленных в кодировке для графического режима. Исторически сложилось так, что тексты, записанные не с помощью латинского алфавита, имеют различное представления в графическом и текстовом режиме. (Для англоязычных стран и пользователей эта особенность совершенно незаметна и часто может быть даже неизвестной.)

Для других стран профессиональное использование программирования для ОС типа Windows требует учета указанной особенности. Решение возникающих при этом проблем обеспечивается парой функций преобразования из одной формы представления в другую. Эти функции задаются упрощенными прототипами

BOOL CharToOem(char *textsource, char *textresult),

BOOL OemToChar(char *textsource, char *textresult).

При использовании указанных функций следует иметь в виду, что разработчики условным буквосочетанием Char в названии функций обозначают кодировку графического режима, а обозначением Oem в названии – кодировку текстового режима. Таким образом функция CharToOem задает преобразование текста из кодировки графического режима Windows в текстовый режим, а функция OemToChar – преобразование текста из кодировки текстового режима в графический режим.

Наиболее совершенные средства обработки ошибок были заложены в OS/2, но рассматривать мы их не будем.

2. ПРОГРАММНЫЙ ДОСТУП К ФАЙЛОВОЙ СИСТЕМЕ

2.1. Понятия дескрипторов, идентификаторов и хэндлов

Операционная система управляет множеством информационных объектов, необходимых для осуществления ее функций. Простейшей разновидностью этих объектов, присутствующих даже в самых примитивных ОС, являются файлы. В многозадачной ОС число типов таких объектов приближается к десяти, а вместе с типами объектов графической оболочки их оказывается еще больше. Мы будем изучать внутренние объекты операционной системы постепенно, отталкиваясь от понятия файла. Напомним, что файлом называется именованный информацион-

ный объект, хранимый в компьютере даже тогда, когда последний выключен. Иначе говоря, файл – это информационный объект длительного хранения, имеющий наименование, легко доступное пользователю, и который может быть перенесен с одного внутреннего места хранения в другое, в частности с одного компьютера на другой или просто на внешний носитель информации (например дискету).

С файлами программист встречается с первых своих шагов, но ему не заметно, как собственно операционная система справляется с обслуживанием файлов. В то же время с этих первых шагов ему становится известно, что для использования файла в программе его нужно *открыть*. Широко известным, хотя часто не замечаемым фактом, является то, что в цивилизованном обществе управлять людьми и экономическими объектами оказывается возможным только с помощью учетной информации, в качестве которой используются паспорта и документы. Аналогично этому во внутреннем мире операционной системы для управления теми же файлами создается и используется учетная информация, которую называют File Control Block или другим аналогичным образом. Терминологическое обозначение может меняться от одной ОС к другой, так как зависит от произвола создателей такой истемы. Для нас главное, что управление файлами при их "внутрикомпьютерной жизни" использует аналоги паспорта. Когда файл просто записан на дискете или жестком диске, в такой учетной информации нет необходимости.

По существу открытие файла и заключается в создании управляющего блока. Для ясного представления желательно знать, что содержится в этом блоке. Там, как минимум, содержится информация о следующем месте чтения из файла или записи в файл. (Напомним, что чтение из файла и запись в него автоматически выполняются последовательно, как на магнитную ленту, от использования которой файлы и произошли.) Программистам известно, что место следующего обращения к файлу можно изменить с помощью специальной функции, обычно называемой SEEK или каким-либо производным от этого слова. В действительности такие функции приводят к изменению параметра именно в управляющем блоке файла, но вовсе не в самом файле. Другим параметром управляющего блока файла служит вид доступа, задаваемый при его открытии (будет ли использоваться файл для чтения, для записи или обеих этих операций). В действительности управляющий блок файла может содержать и, как правило, содержит информацию о месте размещения файла или начале его размещения, позволяющую быстро добраться до его содержимого, но такая информация более важна для внутренних процедур работы с файлами и мы на ней не будем заострять внимание.

Управляющий блок файла – это служебная структура данных, необходимая для управления файлом средствами операционной системы. У вдумчивого программиста возникает вопрос, где же находится эта информация. Возможны два принципиальных решения его: эти структуры данных могут находиться в пределах самой исполняемой программы (ее области данных) и описываться в ней либо раз-

мещаться в служебных областях, доступных только операционной системе. Первое решение использовалось в ранних ОС и помимо дополнительных усилий от программиста (по заданию экземпляров таких структур) оно имело такой недостаток, как возможность непреднамеренного изменения данны структур в результате ошибок программиста и его программы. Поэтому от него, как более трудоемкого и, главное, менее надежного решения, позже отказались. (В частности, в первых версиях MS DOS были системные функции работы с файлами с помощью FCB блоков, которые существовали и позже, но настойчиво не рекомендовались для использования.)

Практическая реализация второго подхода к размещению управляющих блоков файлов появилась вначале в ОС Unix и базируется на понятии хэндла. В этом подходе управляющие блоки размещаются в служебной области операционной системы, причем, как правило, в виде таблицы, и для доступа к ним используется номер строк такой таблицы (порядковый номер управляющего блока во внутренней таблице управляющих блоков для выполняемой программы). Этот номер строк таблицы управляющих блоков и называют *handle*. Иным наименованием для хэндла, использовавшимся как раз в Unix, служит слово *дескриптор* или (в русском переводе) *описатель*. Термин "дескриптор", хотя и является традиционным в операционной системе Unix, на самом деле вызывает ложные аналогии. В действительности его целесообразней применять как синоним управляющего блока, так как именно в последнем содержится учетная информация, описывающая файл и необходимая для выполнения файловых операций. Для операций над файлом при обращении к системной функции в качестве одного из параметров вызова используется значение такого хэндла. По этому значению подпрограмма ОС обращается к соответствующей строке таблицы управляющих блоков и использует информацию из управляющего блока, отвечающего этой строке.

Таким образом, хэндл с внешней (поверхностной) стороны – просто некоторый специальный номер для выполнения операций с конкретным открытым файлом. Более существенным его пониманием, важным для более сложных ситуаций использования, должно быть осознание, что на самом деле этому номеру – хэндлу однозначно соответствует строка специальной таблицы управляющих блоков, которая и дает необходимую информацию для оперативной работы с файлом. Заметим, что английское слово handle переводится на русский язык как ручка, рукоять, но в этот "родном" значении оно не привилось в технической информатике (оказывается удобней использовать чисто фонетический перенос исходного названия вместо применения оборотов русского технического языка вроде "возьмем ручку файла", хотя для англоязычного слушателя и читателя этот оборот звучит именно так).

В переводной технической литературе по программированию встречается перевод термина handle словом идентификатор. (Заметим, что последнее слово

также не русского происхождения.) Этим закладывается некоторая сумятица в умы начинающих программистов, так как сложившееся использование термина identificator относится к принципиально иным информационным объектам. *Идентификатор* позволяет однозначно обозначить некоторый объект. Хэндлы же различных по содержанию и собственному названию файлов могут быть одинаковыми. Причем одинаковыми в один и тот же момент работы компьютера. Такие ситуации имеют место, когда в компьютере выполняется несколько программ одновременно (несколько программ начали свое выполнение, но еще не завершились). В этих выполняемых программах очень часто оказываются одинаковые значения хэндлов (как номеров строк текущих таблиц дескрипторов) по той причине, что для каждой выполняемой программы (точнее вычислительного процесса) операционная система создает и поддерживает свой собственную таблицу дескрипторов файлов. Поэтому одинаковые номера строк таких таблиц содержат различную информацию.

В некоторых ОС термин *идентификатор* (правда для более сложных объектов, чем дескриптор файла) используется независимо от термина *хэндл* и обозначает действительно уникальное в данной операционной системе условное обозначение сложного объекта. Позже идентификаторы сложных системных объектов будут рассматриваться при изучении параллельных процессов в современных операционных системах.

2.2. Ввод и вывод в стандартные файлы.

Разработчики операционной системы Unix кроме "запрятывания" управляющих блоков файлов ввели еще одно новшество, связанное с этими блоками. Так как управляющие блоки файлов описывает и создает сама операционная система, то оказалось возможным поручить ей создавать управляющие блоки некоторых файлов перед запуском программы, точнее в ходе такого запуска. Кандидатами на автоматическое открытие стали обобщенные файлы, связанные с клавиатурой и экраном. Начинающие программисты должны помнить, что и в системах программирования на языках Паскаль и Си для операционных систем MS Windows имеется возможность открывать файлы не только по указанному собственному имени файла, но и использование специальные обозначения для некоторых устройств, в частности, для клавиатуры и экрана монитора (имеющих для этой цели служебное наименование CON). В свое время разработчиками было осознано, что в операционную систему можно предусмотрительно заложить открытие файлов для этих устройств. Возникает вопрос, как программист для создаваемой программы должен узнать значения хэндлов, которые образует ОС при таком предусмотрительном открытии. Было принято простое (и теперь, кажется, очевидное) решение: так как никакие файлы для запускаемой программы до подобного открытия не могли

быть открыты по инициативе последней, то можно поручить ОС разместить дескрипторы файлов (управляющие блоки) в фиксированных позициях внутренней таблицы для таких дескрипторов. Поэтому оказалось возможным зарезервировать определенные номера хэндлов за автоматически открываемыми файлами для консоли и экрана.

Разработчики Unix пошли дальше и сняли ограничение, что эти файлы со стандартным размещением дескрипторов должны быть всегда связаны с клавиатурой и экраном. Решили, что они связываются всегда за исключением таких случаев, когда специальными деталями строки вызова программы не указанно что-то другое. Более того, решили, что два стандартно открываемых дескриптора файлов это слишком ограничено, и зарезервировали для этих целей три. В традиционных обозначениях этим дескрипторам соответствует стандартный ввод, стандартный вывод и стандартный вывод ошибок. Для обозначения высокоуровневого доступа к этим дескрипторам через буферизованный ввод-вывод, осуществляемый с помощью указателей типа FILE*, в Unix приняты символические обозначения stdin, stdout и stderr. Фактическое значение хэндлов, соответствующих этим указателям, для доступа через перечисленные дескрипторы стандартного ввода и вывода в Unix – всегда постоянное и, составляет, соответственно 0, 1, 2.

Таким образом, с первых же строк программы для ввода данных в нее можно пользоваться функциями ввода с помощью хэндла, задав его принудительно константой 0. (Такая возможность есть в Unix, OS/2, MS DOS и некоторых других ОС.) Аналогично для вывода можно воспользоваться функциями вывода с помощью хэндла, задав его константой 1. Построив программу с помощью этих допущений и обозначений, в дальнейшем программу можно вызывать по ее имени, и тогда ввод и вывод в этих программных функциях будут осуществляться с клавиатуры и на экран. Если же пользователь такой программы хочет передавать в нее данные вместо клавиатуры из файла с именем *имяфайлаввода*, то достаточно при вызове программы в командной строке написать

имяпрограммы <имяфайлаввода

При желании получить вывод вместо экрана в некоторый файл *имяфайлавывода*, достаточно задать командную строку в виде

имяпрограммы >имяфайлавывода

Наконец приказ, переназначить ввод с клавиатуры на файл *<имяфайлаввода*, а вывод на файл *имяфайлавывода*, запишется в командной строке текстом

имяпрограммы <имяфайлаввода >имяфайлавывода

Подобная универсальность использования программы с файловым вводом и выводом поистине замечательна, так как без каких-либо ухищрений программиста и минимальными программными текстами обеспечивает массу потенциальных возможностей.

При начальном знакомстве с новой операционной системой, поддерживающей соглашения о стандартном вводе и выводе на основе фиксированных хэндлов для целей ввода и вывода текстов достаточно познакомится только с системными функциями чтения из файлов и записи в файл.

В Unix для этих целей предназначены универсальные системные функции с прототипами

unsigned int read(int handle, void* buffer, unsigned int len),

unsigned int write(int handle, void* buffer, unsigned int len).

Здесь аргумент handle – хэндл файла, второй аргумент buffer - адрес буфера для чтения или соответственно записи данных, последний аргумент len – запрашиваемое для чтения или задаваемое для записи число байтов. Функции возвращают число байтов, которые им удалось передать при вводе или выводе, в частности, функция read возвращает число байтов, прочитанных из файла. Это число может не совпадать с запрошенным, если ввод осуществляется с клавиатуры и завершается символом Enter либо при вводе из обычного файла обнаруживается конец файла.

Следующий пример, приведенный в листинге 2.2.1, демонстрирует простейшее применение этих функций в демонстрационной Unix программе, которая выдает приглашение на ввод текста, и вводимый далее текст выводит с примечанием, что именно он введен в программу.

```
#include <stdio.h>
void  main()
{int cb;
 char buffer[100]="It was readed ... ";
   write(1,"Vvedite\n",8);
   cb=read(0, buffer+18, 80);
   cb += 18;
   write(1, buffer, cb);
}
```

Листинг 2.2.1. Программа для Unix

В операционной системе OS/2 для обозначения стандартного ввода и вывода также использовались фиксированные значения хэндла, равные 0 и 1, а сами универсальные системные функции ввода и вывода файлов имели прототипы

APIRET DosRead(HFILE hFile, VOID *buffer, ULONG len, ULONG *pactualen);

APIRET DosWrite(HFILE hFile, VOID *buffer, ULONG len, ULONG *pactualen);

Здесь аргументы *hFile* задавали хэндлы файлов, с которыми выполняются операции чтения или записи, *buffer* – адреса буфера данных для чтения или записи, аргумент *len* – число запрашиваемых при вводе или задаваемых для вывода байтов, а *pactualen* являлся указателем на длинное слово данных для выдачи (в качестве вспомогательного результата) числа переданных байтов.

В операционных системах клонов Windows использование стандартных файлов для программиста значительно усложняется. Дело в том, что эти операционные системы не предоставляют автоматически в программу фиксированные при разработке значения хэндлов для стандартного ввода и вывода. При необходимости программист может запросить такие значения принудительно, причем заранее неизвестны числовые значения этих хэндлов. Для такого запроса служит специальная функция API Windows, имеющая прототип

HANDLE GetStdHandle(DWORD nStdHandle),

где аргумент используется обычно в виде символической константы, определенной в заголовочном файле. Для аргумента этой функции возможны три значения, описанные в заголовочном файле winbase.h как

```
#define STD_INPUT_HANDLE     (DWORD)-10
#define STD_OUTPUT_HANDLE    (DWORD)-11
#define STD_ERROR_HANDLE     (DWORD)-12
```

Функция GetStdHandle может и не вернуть запрошенное значение хэндла, тогда она возвращает значение константы INVALID_HANDLE_VALUE, в действительности равной числу -1 или, точнее, определяется с точностью до перехода к соответствующему типу как (HANDLE)-1. В общем случае после вызова функции GetStdHandle следует обязательно проверять, удалось ли получить запрошенный хэндл.

После получения значений хэндлов для стандартного ввода и вывода их можно использовать в универсальных системных функциях файлового ввода и вывода для этой операционной системы. Такие функции имеют здесь прототипы

```
BOOL WINAPI ReadFile(HANDLE hFile,  void* buffer, DWORD len,
                                    DWORD *pactualen,   OVERLAPPED *pOverlapped);
BOOL WINAPI WriteFile(HANDLE hFile,  void* buffer, DWORD len,
                                    DWORD *pactualen,   OVERLAPPED *pOverlapped);
```

причем первый из аргументов этих функций задает хэндл (для наших ближайших целей – хэндл файла), второй – *buffer* – определяет место (буфер) для считываемых данных, аргумент *len* задает число байтов, которые запрашиваются для ввода или задаются для вывода. Аргумент *pactualen* – возвращаемый параметр, он определяет (после выполнения системной функции) сколько байтов было действительно прочитано (собственно аргумент задается адресом места для записи возвращаемого значения). Последний аргумент *pOverlapped* имеет достаточно специальный характер и в нашем изложении не будет использоваться, а его значение будет зада-

ваться как NULL. Число прочитанных байтов может быть меньше числа запрошенных, если при чтении запрошенных обнаружен конец файла.

Следующий пример для Windows, приведенный в листинге 2.2.2, демонстрирует простейшее применения рассмотренных системных функций Windows для решения уже рассмотренной примитивной задачи.

```
#include <windows.h>
#include <stdio.h>
void main()
{char buffer[100]="It  was readed . . .";
 DWORD actlen;
 HANDLE hstdin, hstdout;
 BOOL rc;
        hstdout = GetStdHandle(STD_OUTPUT_HANDLE);
        if (hstdout = = INVALID_HANDLE_VALUE) return;
        hstdin = GetStdHandle(STD_INPUT_HANDLE);
        if (hstdin = = INVALID_HANDLE_VALUE)  return;
        rc=ReadFile(hstdin, buffer+18, 80, &actlen, NULL);
        if (!rc) return;
        actlen += 18;
        WriteFile(hstdout, buffer, actlen, &actlen, 0);
  getchar();
}
```

Листинг 2.2.2. Программа для Windows

Непосредственно видно, что программа Windows для решения той же самой простейшей задачи получается заметно сложнее, чем для других ОС. Кроме собственно запроса и проверки получения хэндлов стандартных ввода и вывода в этой программе оказывается практически необходимым дополнительно приостанавливать ее работу на вспомогательном вводе произвольного символа. Такой прием оказывается вынужденным, если данную программу запускать под управлением операционных систем типа Windows 9x. В этих ОС отсутствует постоянно задаваемое текстовое окно консольного вывода, и поэтому запуск консольного приложения из графического режима после выполнения программы приводит к свертыванию временно созданного для нее окна, и пользователь не успевает просмотреть окончательные результаты, выдаваемые такой программой. Приостановка выполнения программы с помощью функции getchar(), входящей в стандартную библиотеку языка Си, требует подключения заголовочного файла stdio.h. Заметим, что подобная программа, ориентированная исключительно на разновидности Windows NT, может использоваться без вспомогательного оператора getchar().

При использовании в программах на языке Си стандартного вывода нужно учитывать следующую существенную особенность. Ряд функций вывода из стандартной библиотеки языка Си используют так называемый *буферизованный вывод*. К этим функциям относятся все те программные функции, которые явно или неявно используют для обозначения файлов указатели типа FILE*. В частности, к ним относятся функции printf() и fprintf(). Для более эффективной работы в общем случае эти функции осуществляют собственно вывод только после заполнения внутреннего буфера или явным указанием его *опустошения* (flush). Такая внутренняя логика функционирования приводит иногда к тому, что начинающий программист получает на выводе не совсем тот результат, который ожидает (или даже совсем не тот!).

Так совместное использование вызовов функций WriteFile и printf в программе для Windows может приводить к тому, что в результирующий файл попадают вначале все данные от выполняемых функций WriteFile, а только затем – от функций printf. Этот эффект легко заметить, если использовать переадресацию стандартного вывода в следующей программе из листинга 2.2.3.

```
#include <windows.h>
#include <stdio.h>
int main()
{HANDLE hout;
 DWORD actlen;
  hout=GetStdHandle(STD_OUTPUT_HANDLE);
  printf("Один...\n"); // fflush(stdout);
  WriteFile(hout,"Два...",6,&actlen,0);
  printf("Три\n");  // fflush(stdout);
  WriteFile(hout,"Четыре...",9,&actlen,0);
  return 0;
}
```

Листинг 2.2.3. Совместное использование буферизованных и небуферизованных функций

Если результат трансляции программы из листинга 2.2.3 будет иметь имя prim223.exe и его запустить в виде командной строки

prim223.exe >kkk

то после запуска программы в результирующем файле kkk будет находиться текст

Два...Четыре...Один...

Три

Это совсем не то, что можно было бы ожидать, глядя на последовательность операторов вывода в программе.

Для исправления ситуации следует использовать функцию принудительного опустошения внутреннего буфера, которая имеет имя fflush и должна в качестве аргумента содержать указатель на соответствующую структуру типа FILE. Для стандартного вывода этот указатель описан в заголовочном файле stdio.h и имеет обозначение stdout. В примере рассматриваемой программы такое дополнение заготовлено в закомментированных продолжениях строк с операторами вывода printf. Убрав эти комментарии (символы //), следует проверить получающийся после этого результат трансляции исходного текста, запустив его аналогичным образом с переадресацией стандартного вывода в файл.

Практическая рекомендация совместного использования буферизованных и небуферизованных функций вывода в простейших случаях заключается в непременном использовании оператора fflush(stdout) за каждым оператором с функцией printf.

При программировании в Unix опустошение буфера при выводе на экран обеспечивается указанием управляющего символа конца строки. Если же выполняемая последовательность операторов задает вывод в буферизованный вывод экрана без управляющих символов '\n', то на экране этот текст появляется только после завершения всей программы! Альтернативой использованию указанных управляющих символов может быть также вызов описанных выше функций fflush(stdout).

При выводе же программой собственно в файл, для Unix характерны те же особенности, что уже рассмотрены на примере Windows. Это значит, что и в Unix при сочетании в одной программе функций небуферизованного вывода и буферизованных функций вывода следует использовать операторы fflush(stdout) для организации желаемой последовательности данных в файле результата. В простейших случаях следует каждый вывод через функцию printf сопровождать последующим оператором fflush(stdout).

2.3. Базовые средства использования файлов

Для выполнения операций над файлом, если он отличен от стандартного, файл необходимо открыть. При открытии, как уже пояснялось, операционная система создает управляющий блок для открытого файла, размещая его в области данных, недоступной прикладной программе, и возвращая в место вызова функции открытия хэндл для последующей работы с файлом.

После того, как исчезнет потребность в использовании открытого в программе файла, его следует *закрыть*. Результатом закрытия файла по существу является ликвидация управляющего блока для этого файла. Но такая ситуация имеет место только тогда, когда этот управляющий блок использует только одна выполняемая программа. Позже в связи с изучением программ, одновременно выполняющихся

с помощью операционной системы, познакомимся, когда и как возникают подобные ситуации, и что происходит в общем случае с управляющими блоками при выполнении системных функций закрытия файла. Рассмотрим вначале базовые средства операционных систем типа MS Windows.

Для получения хэндла открываемого файла в них используется функция CreateFile, которая предназначена и для собственно создания и, в частности, для открытия уже существующего файла. Заметим, что в этих ОС имеется два варианта функции создания и открытия файлов, отличающихся последней дополнительной буквой – A или W. Первый вариант отвечает использованию кодирования символов по стандарту ANSI, а второй – по стандарту UNICODE. Второй вариант позволяет использовать множество разнообразных алфавитов и для кодирования каждого символа задействует не один (как в ANSI), а два байта. Этот последний вариант очень перспективен, но на текущий момент не получил преимущественного распространения, поэтому будем использовать исключительно более консервативный вариант ANSI. Практически дополнительные символы кодировки (A или W) оказываются явно существенными только при записи программ на ассемблере. При трансляции программ, написанных на языке Си, предкомпилятор учитывает неявно задаваемый вид кодировки символов и выполняет соответствующее преобразование имени функции без последнего дополнительного символа в один из подразумеваемых вариантов. Для задания такого неявного варианта используются специальные константы, задаваемые компилятору.

Функция CreateFile имеет 7 аргументов, первым из которых является имя открываемого файла, вторым – код желаемого доступа к файлу, третьим – код режима разделяемого использования файла, далее следует адрес атрибутов защиты файла (мы не будет использовать эти довольно не простые возможности и этот аргумент всегда будем полагать равным NULL, т.е. сообщать ОС об отсутствии информации о защите файла). Пятый аргумент задает поведение ОС при открытии файла (диспозицию), шестой – атрибуты файла, а последний имеет специальный характер и рассматриваться нами не будет (будем указывать значение этого аргумента как NULL). Функция CreateFile при удачном выполнении возвращает значение хэндла файла, а при ошибке выдает вместо него значение, задаваемое символической константой INVALID_HANDLE_VALUE. На языке Си прототип функции функция CreateFileA записывается в виде

```
HANDLE CreateFile(LPCSTR pFileName, DWORD DesiredAccess,
   DWORD ShareMode,   LPSECURITY_ATTRIBUTES pSecurityAttributes,
   DWORD CreationDisposition, DWORD FlagsAndAttributes,
   HANDLE hTemplateFile);
```

где *pFileName* задает место имени файла (в терминах языка СИ обозначает *указатель* на имя файла), *DesiredAccess* – код желаемого доступа, *ShareMode* – код режима разделения работы с файлом, *pSecurityAttributes* – указатель на атрибуты за-

щиты файла, *CreationDisposition* – код действия над файлом во время выполнения данной функции, *FlagsAndAttributes* – флаги атрибутов, *hTemplateFile* – хэндл файла шаблона с расширенными атрибутами.

Атрибуты защиты не используются в ОС типа Windows 9x и поэтому параметр *pSecurityAttributes* должен задаваться в этих ОС значением NULL. В ОС типа Windows NT нулевое значение этого параметра равносильно указанию использовать атрибуты защиты по умолчанию. С учетом нетривиальности использования внутрипрограммной защиты будем всегда считать, что этот и подобные параметры задаются по умолчанию, и брать в качестве их значений NULL.

Параметр *FlagsAndAttributes* задает атрибут открываемого файла. Обычный (нормальный) файл имеет атрибут, равный 0; файл доступный только для чтения – атрибут, равный 1; скрытый файл задается атрибутом 2, системный файл – атрибутом 4. В качестве этого параметра можно использовать (чаще всего так и делают) символическую константу FILE_ATTRIBUTE_NORMAL, определенную в заголовочном файле. Для кодирования доступа к открываемому файлу в параметре *DesiredAccess* служат две символических константы GENERIC_READ и GENERIC_WRITE, задающих соответственно разрешение на чтение и запись в файл. Они могут быть использованы совместно, путем объединения (по операции логического ИЛИ) в одном параметре *DesiredAccess*, или раздельно – по необходимости.

Совместное использование файла задается в параметре *ShareMode* символическими константами FILE_SHARE_READ и FILE_SHARE_WRITE, которые также можно при необходимости комбинировать в одном параметре. Для задания действий с файлом в параметре *CreationDisposition* служат символические константы CREATE_NEW, CREATE_ALWAYS, OPEN_EXISTING, OPEN_ALWAYS, TRUNCATE_EXISTING, которые нельзя комбинировать в одном значении параметра *CreationDisposition*, а следует использовать порознь. Константа CREATE_NEW приводит к тому, что если файл, заданный в функции CreateFile уже существует, то функция возвращает ошибку. Константа CREATE_ALWAYS требует создания файла всегда, даже взамен существующего, при этом содержимое старого файла теряется. Константа OPEN_EXISTING требует открыть только существующий файл, если же при этом файла с указанным именем не существует, то функция возвращает ошибку. Константа OPEN_ALWAYS приводит к тому, что существующий файл открывается, а если файл не существовал, то он создается. Константа TRUNCATE_EXISTING приводит к следующим действиям: если файл существует, то он открывается, после чего длина файла устанавливается равной нулю, содержимое старого файла при этом теряется; если же файл не существовал, функция CreateFile возвращает ошибку. Все перечисленные константы описаны в заголовочных файлах.

Для закрытия файла используется функция CloseHandle, назначение которой значительно шире, чем просто функций закрытия файла в других ОС. Функция эта имеет прототип

BOOL CloseHandle(HANDLE hObject),

где хэндл его управляющего блока для закрытия доступа к файлу должен задаваться в качестве аргумента функции. Логическое значение, возвращаемое функцией, позволяет определить, удалось ли закрыть хэндл.

Следующая программа, приведенная в листинге 2.3.1, демонстрирует пример использования в операционной системе Windows функции открытия файла для дальнейшего чтения из этого файла. В этой программе осуществляется ввод пользователем некоторого текста с клавиатуры и запись его с поясняющим текстовым префиксом в файл, имеющий имя myresult.txt.

```
#include <windows.h>
void main()
{char buffer[100]="It  was readed ";
 DWORD len, actlen;
 HANDLE hstdin, fhandle;
 char fname[ ]="myresult.txt";
 BOOL rc;

        len = strlen(buffer);   // вычисляет длину текста в буфере
       hstdin = GetStdHandle(STD_INPUT_HANDLE);
       if (hstdin = = INVALID_HANDLE_VALUE) return;
       fhandle=CreateFile(fname, GENERIC_WRITE, 0, 0,
                    CREATE_ALWAYS, FILE_ATTRIBUTE_NORMAL, 0);
       if (fhandle = =INVALID_HANDLE_VALUE) return;
       rc=ReadFile(hstdin, buffer+len, 80, &actlen, NULL);
       if (!rc) return;
       WriteFile(fhandle, buffer, len+actlen, &actlen, NULL);
       CloseHandle(fhandle);
}
```

Листинг 2.3.1. Программа для Windows

В начале программы функция GetStdHandle выдает хэндл стандартного ввода, который будет позже использоваться для ввода данных. Затем выполняется создание файла с именем *myresult.txt*. Файл создается для записи в него, что определяет константа GENERIC_WRITE в параметре доступа. Кроме того, он создается как недоступный для других процессов после открытия в данной программе, что задается нулевой константой в четвертом параметре, где для одновременного чтения

или записи другими процессами следовало указывать константу FILE_SHARE_READ или FILE_SHARE_WRITE. Файл создается всегда, что диктует константа CREATE_ALWAYS, с целью заменять при последующем запуске старое содержимое, и с нормальными атрибутами, которые определяет константа FILE_ATTRIBUTE_NORMAL в параметре атрибутов.

В операционной системе Unix для открытия файла служит функция с прототипом

int open(char* filename, int access_mode, mode_t permission),

которая возвращает в случае удачного своего выполнения хэндл открытого файла или значение -1 в случае ошибки. Первый аргумент функции задает имя открываемого файла (в общем случае относительное или абсолютное имя файла в Unix). Второй аргумент определяет режим доступа к открываемому файлу для процесса, вызывающему функцию open. Этот аргумент задают обычно комбинацией одной из символических констант O_RDONLY, O_WRONLY, O_RDWR с комбинацией *флагов модификаторов доступа*. Комбинация записывается как логическое объединение соответствующих констант. Модификаторы задаются обозначениями O_APPEND, O_CREAT, O_EXCL, O_TRUNC.

Константы O_RDONLY, O_WRONLY, O_RDWR определяют соответственно открытие файла только для чтения, открытие только для записи и, наконец, открытие для записи и чтения. Модификатор O_APPEND задает добавление данных в конец файла, модификатор O_CREAT указывает создать файл, если он не существует. Модификатор O_EXCL используется только вместе с флагом O_CREAT, если оба они установлены, а указанный файл уже существует, выполнение функции open завершается неудачей. Модификатор O_TRUNC приводит к тому, что если файл существует, то его содержимое отбрасывается и устанавливается размер файла равным нулю. При отсутствии этого флага сохраняется все содержимое исходного одноименного файла, открытого функцией open, а вывод в файл осуществляется как бы перекрытием этой существующей информации в нем. (Имеется еще пара модификаторов, которые здесь не рассматриваются.)

Аргумент *permission* необходим только в том случае, когда в аргументе *access_mode* установлен флаг O_CREAT, т.е. файл создается. Аргумент *permission* задает права доступа к файлу для его владельца, членов группы и всех остальных пользователей. В более простых вариантах использования операционной системы Unix, тип этого аргумента задается просто как int. (Общий случай описывается стандартом POSIX и определяется в заголовочном файле sys/stat.h.) Значение *permission*, указанное в вызове функции, модифицируется значением системной переменной *umask* процесса, вызывающего данную программу. Значение этой переменной задает права доступа, которые автоматически исключаются для всех файлов, создаваемых данным процессом. (Переменную эту для процесса можно опросить и изменить с помощью системного вызова, имеющего прототип

mod_t umask(mode_t new_umask).

Аргументом функции *umask* служит новое (устанавливаемое) значение маски прав доступа, а возвращает функция старое значение.)

Для каждого создаваемого файла принимается значение кода доступа, получаемое как permission & ~umask_value, где umask_value – текущее значение маски прав доступа. Обычно в качестве прав доступа создаваемому файлу задают восьмеричное значение 0600, что определяет права создателю файла на чтение и запись, а для всех остальных не разрешается никакой доступ к этому файлу. (Правая восьмеричная цифра – код прав для всех остальных, следующая цифра – код прав для членов той же группы, третья справа восьмеричная цифра – код прав для владельца, которым автоматически становится создатель файла. Отдельные биты в восьмеричной цифре кода доступа определяют: старший право записи, следующий бит – право чтения, а бит, отвечающий числу 1, задает право выполнения.)

Для закрытия файла в Unix служит функция с прототипом

int close(int handle).

Следует отметить, что использование функций работы с файлами (открытия, закрытия и других, которые будут рассмотрены дальше) требует подключения заголовочного файла fcntl.h.

Получить доступ к уже имеющемуся управляющему блоку открытого файла можно через другое значение хэндла, чем было получено при открытии файла. Практическое применение множественные хэндлы файлов имеют при использовании объектов более сложных чем простые файлы, в частности, для использования каналов передачи данных, которые будут рассматриваться в одной из последних глав данного пособия. Для понимания связей между файлами, хэндлами и управляющими блоками файлов рассмотрим начальные элементы этих возможностей прямо сейчас.

Для получения другого хэндла к уже открытому файлу в Unix предназначена функция с прототипом

int dup(int hsource)

В Windows для тех же целей можно использовать частный вариант системной функции DuplicateHandle, который предварительно рассмотрим в виде

```
BOOL DuplicateHandle(GetCurrentProcess(),
   HANDLE  hsource,       // handle to duplicate
   GetCurrentProcess(),   HANDLE  *htarget, // address of duplicate handle
   0, FALSE, DUPLICATE_SAME_ACCESS)
```

На самом деле системная функция DuplicateHandle обладает огромными и часто небезопасными возможностями, но мы используем ее в наиболее безопасном виде – для действий с системными объектами, предназначенными только для текущей программы. В этом ее применении только второй и четвертый параметры служат для подстановки переменных из конкретной программы, причем в четвертом

параметре должен быть задан адрес такой переменной. Последняя переменная предназначена для получения хэндла, возвращаемого функцией при ее удачном выполнении.

2.4. Многопользовательская блокировка файлов

Непосредственное использование в Unix только функций открытия, закрытия, записи в файл и чтения из файла не обеспечивает полных возможностей совместного использования файлов. Для управления совместным доступом к открытым файлам Unix содержит дополнительные средства, обеспечиваемые через системную функцию fcntl. Функция эта имеет прототип

```
int  fcntl(int handle, int cmd, . . .);
```

Параметр *cmd* задает операции, которые необходимо выполнить над файлом, указанным аргументом *handle*. Третий аргумент этой функции, который может быть указан после *cmd*, зависит от конкретного значения операции *cmd*. Имеется целый ряд возможных значений для аргумента операции функции fcntl, но большинство из них не будет рассматриваться здесь. Ограничимся только значениями, относящимися к взаимной блокировке файлов и их записей. В этом случае функция fcntl приобретает вид

```
int  fcntl(int handle, int cmd, struct flock *ldata);
```

Для этого варианта имеем три основных значения аргумента *cmd*. Они обозначаются символическими константами F_SETLK, F_SETLKW и F_GETLK. Структуры flock данных описываются в заголовочном файле в виде

```
struct flock
{short   l_type;      // тип блокировки
 short   l_whence;    // код базовой позиции для отсчета смещения
 off_t   l_start;     // смещение относительно базовой позиции
 off_t   l_len;       // сколько байтов находится в заблокированной области
 pid_t   l_pid;              // PID процесса, который блокировал файл
};
```

Сами символические константы операций блокировок обозначают следующее. Константа F_SETLK задает попытку применить блокировку к файлу и немедленно возвратить управление (либо же отменить блокировку, если это действие задано значением *l*_type, равным константе F_UNLCK). При неудачной попытке наложения блокировки с операцией, заданной константой F_SETLK, функция fcntl возвращает код -1. Константа F_SETLKW приказывает попытаться применить блокировку к файлу и приостановить работу, если блокировка уже наложена другим процессом. Константа F_GETLK делает запрос на получение описания блокировки, отвечающей данным в аргументе *ldata* (возвращаемая информация

описывает первую блокировку, препятствующую наложению блокировки, которая задается структурой *ldata*).

Код базовой позиции для отсчета смещения задает либо начальную позицию в файле (константой SEEK_SET), либо текущую позицию в нем (константой SEEK_CUR), либо позицию конца файла (константой SEEK_END).

В пояснение описанных деталей отметим, что средства Unix позволяют управлять совместным доступом не только ко всему файлу, но и к отдельным участкам его. В частности, разрешить другим процессам только читать отдельные участки файла, а другим, может быть, разрешить изменять их. Поэтому указанные средства содержат не только указание разрешения вида доступа или соответственно запрета (блокировки) доступа, но и описание участка, к которому относится такой запрет. Именно с этой целью в состав структуры *flock* включены поля *l_whence*, *l_start*, *l_len*, причем поле *l_len* задает длину участка, на который распространяется действие конкретного вызова функции *fcntl*; поле *l_start* задает начальное смещение рассматриваемого участка относительно точки позиции, указанной параметром *l_ whence*. Последний, в свою очередь, значениями констант задает отсчет позиции относительно начала файла, относительно текущего положения указателя позиции в нем или относительно конца файла. В качестве указанных констант могут быть взяты соответственно значения 0, 1, 2 или именованные константы SEEK_SET, SEEK_CUR, SEEK_END из заголовочного файла stdio.h. Поле *l_len* может быть задано числом 0, тогда блокировка распространяется от указанного в структуре начала файла до его конца (причем при дальнейшем увеличении файла в процессе работы с ним эта блокировка распространяется дальше – до текущего конца файла).

В структуре данных *flock* тип блокировки, указываемый полем *l_type*, задается символическими константами, предварительно описанными в заголовочном файле. Эти константы следующие: F_RDLCK – установить на указанную в структуре область данных блокировку чтения; F_WRLCK – установить на указанную в структуре область данных блокировку записи; F_UNLCK – снять блокировку с указанной области. Функция *fcntl* может быть использована многократно для различных участков файла и для них с ее помощью могут быть установлены, изменены или сняты определенные виды блокировок.

Блокировка по чтению (константой F_RDLCK) может быть установлена только в том случае, если файл, на который она устанавливается текущей программой, открыт для чтения (т.е. с режимом O_RDONLY или O_RDWR). Блокировка по записи (константой F_WRLCK) может быть установлена только, если файл, на который она устанавливается текущей программой, открыт данной программой для записи (т.е. с режимом O_WRONLY или O_RDWR).

Блокировка чтения, задаваемая константой F_RDLCK, просто предотвращает установку другими процессами блокировки записи. Несколько процессов могут

одновременно выполнять блокировку чтения для одного и того же участка файла. Блокировка по чтению может использоваться, чтобы предотвратить обновление данных, не скрывая их от просмотра другими процессами. Блокировка записи, задаваемая константой F_WRLCK, предотвращает установку другими процессами блокировок чтения и записи для файла. Для заданного участка файла может существовать только одна блокировка записи одновременно. Блокировка записи может использоваться для запрещения просмотра участков файла при выполнении обновлений.

Блокировка по чтению на участок файла может устанавливаться независимо различными процессами, но блокировку по записи может установить только один процесс. Попытки других процессов установить такую блокировку, пока она не снята процессом, установившем ее, оказываются неудачными. Эта неудача проявляется в том, что при использовании операции SETLK функция fcntl возвращает значение -1, а при использовании операции SETLKW в функции fcntl процесс приостанавливается до снятия блокировки другим процессом.

Заметим, что блокировка посредством вызова функции *fcntl* действует только, если ее использование систематически применяется всеми программами, обращающимися к файлу. Если некоторая программа обращается к тому же файлу операциями *read* или *write* без предварительных операций *fcntl* в попытках установить блокировку, то, к сожалению, никаких блокировок, предохраняющих несогласованное использование файла, не реализуется.

Следующая программа, приведенная листингом 2.4.1, демонстрирует использование функций открытия файла, закрытия файла, записи в файл, задание ограничения совместного доступа и аналогична по выполняемым действиям программе в листинге 2.3.1, написанной для другой ОС.

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

int main()
{char buffer[100]="Было прочитано ";
 int len;
 int fhandle;int fhandle;
 char fname[ ]="myresult.txt";
 struct flock lock={F_WRLCK, SEEK_SET, 0, 0};

       fhandle=open(fname, O_WRONLY | O_CREAT | O_TRUNC, 0600);
       fcntl(fhandle, F_SETLKW, &lock);
       write(1, "Vvedite\n",8);
       len=read(0, buffer+16, 80); // записываем вводимое после начального текста
```

```
        write(fhandle, buffer, 16+len);
        sleep(10);
        lock.l_type=F_UNLCK;
        fcntl(fhandle, F_SETLK, &lock);
        close(fhandle);
}
```

Листинг 2.4.1. Программа ограничения доступа к файлу в Unix

В этом примере значение поля *l_type* структуры данных *lock* типа *flock* задается равным константе F_WRLCK, т.е. указывается блокировка при записи данной программой, которая равносильна запрету как по чтению, так и по записи для других процессов. Начальный адрес участка задается на основе начальной позиции в файле (константа SEEK_SET в поле *l_whence*) со смещением в *l_start*, равным нулю, т.е. указывается, что управляемый участок начинается с самого начала файла. Значение поля *l_len* задано нулевым, что равносильно указанию распространения действий до конца файла. Таким образом указано, что воздействию запрета на доступ для других процессов подлежит весь файл. Рассматриваемый запрет на запись осуществляется вызовом функции fcntl(fhandle, F_SETLKW, &lock) сразу после открытия файла. Если в последующие моменты времени до снятия этой блокировки другой процесс выполняет аналогичную функцию fcntl(fhandle, F_SETLKW, &lock), то он приостанавливается на этой функции до тех пор, пока блокировка не будет снята.

После записи в файл функция write задает отмену запрета на запись, установленного данным процессом. С этой целью выполняется функция fcntl(fhandle, F_SETLK, &lock), в параметре *lock* которой задано поле *l_type*, равное константе F_UNLCK, т.е. указывается отмена блокировки (того же участка, что и раньше был указан в этой структуре).

Заметим, что если не требуется принимать мер по управлению совместным использованием файла, то в программе для Unix достаточно просто не использовать функцию *fcntl* для этих целей. В частности, для упрощения рассматриваемого примера можно было отбросить из программы вызовы функции *fcntl*, описание и установки полей экземпляра *lock* структуры *flock*. В программах же для операционных систем MS Windows и OS/2 во всех случаях приходится использовать те же функции, что и в более сложном случае, причем с заданием всех их параметров.

В операционных системах Windows для блокировки участков файлов предназначена функция, имеющая прототип

```
BOOL LockFile(HANDLE  hFile,
      DWORD  FileOffsetLow, DWORD  FileOffsetHigh,
      DWORD  nNumberOfBytesToLockLow,
```

DWORD nNumberOfBytesToLockHigh).

Данная функция при удачном выполнении блокирует доступ к указанному в ней участку для всех других процессов. Эта функция предназначена в общем случае для работы с очень большими файлами, размер которых превосходит числа, задаваемые в формате DWORD (более чем 32-битные двоичные числа, что соответствует границе в 4Гбайта). Поэтому для задания смещения блокируемого участка используются два аргумента *FileOffsetLow* и *FileOffsetHigh*, задающие младшую и старшую части такого смещения. Аналогичным образом для задания числа байтов в блокируемом участке предназначены два аргумента *nNumberOfBytesToLockLow* и *nNumberOfBytesToLockHigh*, также задающие младшую и старшую части этого числа.

Удалось или нет заблокировать запрошенный участок, программа определяет по возвращаемому функцией LockFile логическому значению. При неудачном задании блокировки выполняемая программа не приостанавливается сама по себе, но для этого требуется выполнение дополнительных операторов, вызывающих системные функции, которые рассмотрим впоследствии. (Простейшим, но не эффективным решением являются многократные попытки успешно выполнить функцию блокировки LockFile до тех пор, пока она не вернет значение, свидетельствующее о успешно установленной блокировке.)

Обратной к функции LockFile является системная функция UnlockFile, которая имеет прототип

```
BOOL UnlockFile(HANDLE hFile,
   DWORD FileOffsetLow, DWORD FileOffsetHigh,
   DWORD cbUnlockLow, DWORD cbUnlockHigh),
```

где аргументы по своему назначению совпадают с уже рассмотренными в функции LockFile. Данная функция снимает блокировку с указанного в ней участка, давая тем самым свободный доступ к нему, если это, конечно, согласуется с общими режимами доступа для других процессов, определяемыми аргументом *ShareMode* в функции открытия файла CreateFile.

Кроме двух рассмотренных функций, предназначенных для управления блокировкой отдельных участков в Windows, эта ОС содержит несколько позже добавленные в нее функции, которые дают возможности расширенного управления выборочной блокировкой. Это – функции LockFileEx и UnlockFileEx, которые имеют прототипы

```
BOOL LockFileEx(HANDLE hFile,   DWORD Flags,
                DWORD dwReserved,
                DWORD nNumberOfBytesToLockLow,
                DWORD nNumberOfBytesToLockHigh,
                LPOVERLAPPED lpOverlapped)
```

и

```
BOOL UnlockFileEx(HANDLE  hFile,   DWORD  dwReserved,
                  DWORD  nNumberOfBytesToUnlockLow,
                  DWORD  nNumberOfBytesToUnlockHigh,
                  LPOVERLAPPED  lpOverlapped).
```

По существу эти расширенные функции очень похожи на только что рассмотренные, но позволяют более сложное управление блокировкой за счет параметра Flags, в котором может находиться одна из следующих констант:

LOCKFILE_FAIL_IMMEDIATELY

LOCKFILE_EXCLUSIVE_LOCK

или их объединение. Первая из этих констант задает немедленное возвращение из системной функции при невозможности установить блокировку. При отсутствии этой константы происходит переход к состоянию ожидания, если заданный участок или его часть уже заблокированы другим процессом. Вторая константа дополнительно задает исключающую блокировку. При отсутствии последней константы блокировка является разделяемой между процессами, т.е. может быть установлена более чем одним процессом.

В двух последних функциях значение смещения рассматриваемого участка файла размещается внутри структуры данных, задаваемых последним аргументом. Как видим, полноценное блокирование участков файлов в ОС типа Windows задается в действительности еще сложней чем в Unix.

2.5. Установка произвольной позиции в файле

Следующей по значимости для работы с файлами после уже рассмотренных системных функций является функция позиционирования в файле. Эта функция обозначается в MS Windows именем SetFilePointer и именем *lseek* в операционной системе Unix. Функция позиционирования позволяет организовать прямой доступ к данным в файле путем указания места в файле, с которого будет выполняться следующее чтение или запись.

Функция позиционирования SetFilePointer для MS Windows на языке Си имеет прототип

```
DWORD WINAPI SetFilePointer(HANDLE hFile, LONG ib,
                  LONG* pDistanceToMoveHigh,   DWORD metod);
```

где *hFile* – хэндл позиционируемого файла, *ib* – число байтов, на которое будет перемещена текущая позиция в файле относительно точки отсчета (младшие 32 бита), *metod* – метод отсчета смещения в файле, pDistanceToMoveHigh – адрес старшего слова размещения количества байт, на которые требуется переместить текущую позицию в файле (при использовании файлов меньше 4 Гбайт этот параметр не используется – задается как NULL). Результирующая позиция после перемещения выдается функцией в качестве значения типа DWORD. При ошибке

функция возвращает значение -1. Метод отсчета задается символическими константами FILE_BEGIN, FILE_CURRENT, FILE_END, которые определены в заголовочном файле.

Следующий пример, представленный листингом 2.5.1, демонстрирует использование позиционирования файла в MS Windows.

```
#include <windows.h>
#include <stdio.h>
void main()
{char buffer[100]="It  was readed ";
 int len;
 DWORD cb, cbw1;
 HANDLE hstdout, fhandle;
 char fname[ ]="myresult.txt";
 BOOL rc;
        len = strlen(buffer);
        hstdout = GetStdHandle(STD_OUTPUT_HANDLE);
        if (hstdout = = INVALID_HANDLE_VALUE) return;
        fhandle=CreateFile(fname, GENERIC_READ, FILE_SHARE_READ, 0,
                OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, 0);
        if (fhandle= = INVALID_HANDLE_VALUE) return;
        rc=SetFilePointer(fhandle, 10, 0, FILE_BEGIN);
        if (rc= =NULL) return;
        rc=ReadFile(fhandle, buffer+len, 80, &cb, NULL);
        if (!rc) return;
        cb += len;
        WriteFile(hstdout, buffer, cb, &cbw1, NULL);
        CloseHandle(fhandle);
        getchar();
        return;
}
```

Листинг 2.5.1. Программа для Windows

В начале программа получает от ОС хэндл стандартного вывода вызовом функции GetStdHandle. Затем выполняется открытие файла с доступом по чтению и нормальными атрибутами, причем в режиме доступности по чтению другим процессам. Действия по открытию задаются флагом OPEN_EXISTING, т.е. файл, указанный именем myresult.txt, должен уже существовать, иначе возникает ошибка, обнаруживаемая следующими командами. Предполагается использовать эту программу после выполнения программы, описанной листингом 2.3.1, которая и

должна была создать файл с именем myresult.txt. Затем указатель текущей записи в файле устанавливается на 10 байт от начала файла, что выполняется вызовом функции SetFilePointer. Далее выполняется чтение из этого файла – с той позиции, которая была только что установлена, и прочитанный текст выводится на стандартный вывод функцией WriteFile. Файл закрывается, а программа завершается.

В операционной системе Unix для установки текущей позиции в файле служит функция с прототипом

off_t lseek (int handle, off_t pos, int whence),

где тип данных *off_t* совпадает в простейшем применении Unix с типом *int* (в общем случае тип данных *off_t* определяется в заголовочном файле sys/types.h). В качестве значения параметра *whence* следует использовать символические константы SEEK_CUR, SEEK_SET, SEEK_END, определяющие соответственно отсчет изменения позиции от текущего положения, от начала файла и от его конца. Второй параметр – *pos* определяет относительное смещение позиции в файле от указанной параметром whence точки отсчета. Результирующая позиция файла выдается как значение функции lseek. В случае неудачи функция возвращает значение -1.

Для использования больших файлов в современных версиях операционных разновидностей Unix предназначена модификация базовой функции позиционирования, имеющая прототип

__off64_t lseek64 (int handle, __off64_t pos, int whence).

Она отличается от своей предшественницы только применением 64-битных значений в качестве смещения в файле, задаваемого вторым аргументом и возвращаемым значением с употреблением соответствующих типов данных для таких значений. Можно и неявно использовать 64-битные смещения в обычных функциях позиционирования lseek (с описываемым обобщенным типом данных off_t), если перед заголовочными файлами программы определить символическую константу __USE LARGEFILE64 (которая имеет два символа подчеркивания в начале имени), задав такое определение в виде

#define __USE LARGEFILE64

Следующий пример, представленный листингом 2.5.2, демонстрирует использование позиционирования файла в Unix и по своим действиям аналогичен программе из листинга 2.5.1.

```
#include <stdio.h>
#include <string.h>
#include <fcntl.h>
int main()
{char buffer[100]="Было прочитано ";
  int len, cb, cbw1;
  int hstdin=0, hstdout=1, fhandle;
```

```
 char fname[ ]="myresult.txt";
 struct flock lock={F_RDLCK, SEEK_SET, 0, 0};
        len = strlen(buffer);
        fhandle=open(fname, O_RDONLY);
        if (fhandle = = -1) exit(1);
        lseek(fhandle, 10, SEEK_SET);
        fcntl(fhandle, F_SETLKW, &lock);
        cb=read(fhandle, buffer+len, 80);
        cb += len;
        cbw1=write(hstdout, buffer, cb);
        lock.l_type=F_UNLCK;
        fcntl(fhandle, F_SETLK, &lock);
        close(fhandle);
}
```

Листинг 2.5.2. Программа для Unix

В этой программе сначала открывается файл с именем myresult.txt, причем только для чтения (флаг открытия O_RDONLY), затем позиционируется функцией *lseek* относительно начала (параметр SEEK_SET) на десятый байт. Потом устанавливается блокировка по чтению от данного процесса на весь файл, из файла читаются данные, которые приписываются к тексту в массиве buffer. Полученный текст выводится функцией *write* в стандартный файл вывода, снимается блокировка с файла, он далее закрывается, и программа завершается.

Упражнения

1. Разработать программу для Linux, которая открывает файл для записи, вводит со стандартного ввода несколько десятков различных символов и запоминает их в файле, открытом программой.

2. Разработать программу для MS Windows, которая открывает файл для записи, вводит со стандартного ввода несколько десятков различных символов и запоминает их в открытом программой файле.

3. Разработать программу для Linux, которая открывает для чтения файл, созданный в упр. 1, получая значение хэндла. Затем по этому первому хэндлу функцией dup() строится его дубликат – второй хэндл этой программы. Далее выполняется еще одно открытие того же исходного файла для чтения, причем получается третий хэндл. Функцией lseek выполняется позиционирование на 10-й байт открытого файла, используя для этого первый из полученных ранее хэндлов. Затем выполняются три чтения из файла по 7 байтов, используя по очереди все три ранее полученных хэндла и выводя полученные от файла данные через стандартный вы-

вод. Каждый такой вывод следует завершать дополнительным выводом символа перевода на новую строку. В заключение требуется вывести числовые значения всех трех используемых хэндлов и выполнить их закрытие. Результаты вывода объяснить.

4. Разработать программу для MS Windows, которая открывает для чтения файл, созданный в упр. 1, получая значение хэндла. Затем по этому первому хэндлу функцией DuplicateHandle() строится его дубликат – второй хэндл этой программы. Далее выполняется еще одно открытие того же исходного файла для чтения, причем получается третий хэндл. Функцией SetFilePointer выполняется позиционирование на 12-й байт открытого файла, используя для этого первый из полученных ранее хэндлов. Затем выполняются три чтения из файла по 8 байтов, используя по очереди все три ранее полученных хэндла и выводя полученные от файла данные через стандартный вывод. Каждый такой вывод следует завершать дополнительным выводом символа перевода на новую строку. В заключение требуется вывести числовые значения всех трех используемых хэндлов и выполнить их закрытие. Результаты вывода объяснить.

5. Разработать программу для MS Windows, осуществляющую вывод на "стандартный файл вывода" содержимого некоторого текстового файла, с запретом совместного чтения другими процессами и организованной задержкой перед прекращением запрета на несколько секунд. Перед открытием файла и после попытки его открытия и анализа ситуации выдавать поясняющие сообщения на экран, отражающие существо происходящего. Запустить в двух экземплярах программу на выполнение, используя для этого либо различные экземпляры командных оболочек, либо системную программу Проводника (Explorer) и пронаблюдать поведение программы. При запущенной на выполнение программе попытаться открыть используемый текстовый файл с помощью системной программы просмотра содержимого файлов (из командной оболочки или иным способом). Результаты объяснить.

6. Разработать программу для Linux, осуществляющую вывод на "стандартный файл вывода" содержимого некоторого текстового файла, с запретом совместного чтения другими процессами и организованной задержкой перед прекращением запрета на несколько секунд. Перед открытием файла и после попытки его открытия и анализа ситуации выдавать поясняющие сообщения на экран, отражающие существо происходящего. Запустить в двух экземплярах программу на выполнение, используя для этого различные виртуальные консоли. Переключение между консолями выполнять нажатием комбинаций клавиш Alt-Fn, где n – номер консоли (находится в пределах от 1 до 6). Пронаблюдать поведение программы. Результаты объяснить.

3. ПРИНЦИПЫ ПОСТРОЕНИЯ ОС

3.1. Модульная структура построения ОС

Построение такой сложной системы, как операционная, невозможно без использования принципа модульности. Этот принцип состоит в том, что сложная система при проектировании разбивается на множество частей, относительно мало связанных друг с другом и достаточно просто взаимодействующих друг с другом. Практическое разбиение проектируемой системы на такие части-модули больше искусство чем наука, но для достаточно сложных систем такой путь оказывается единственным, чтобы справиться с совокупной сложностью всей системы.

Проектирование программных систем составляет отдельную и очень важную область практического программирования и изучается обычно в курсах технологий программирования. Операционные системы – уникальные произведения человеческого ума и труда, но после того, как они уже созданы, к ним достаточно просто подходить со стороны их модульного строения. (Это строение в процессе разработки может изменяться, но после завершения разработки воспринимается как логическое и установившееся.)

Подобно строению большинства программных систем, строение операционных систем определяется в первую очередь их функциями. Основные функции ОС, рассматривавшиеся в гл. 1, наряду с другими, более частными, но необходимыми, вызывают разбиение ОС на следующие подсистемы, характерные для всех многопрограммных ОС.

Этими подсистемами являются:

- файловые системы;
- подсистема управления устройствами;
- подсистема управления оперативной памятью;
- подсистема управления процессами;
- подсистема интерфейса с пользователем;
- подсистема управления задачами и диспетчеризацией;
- подсистема ведения времени и обслуживания таймеров.

Такое деление является самым грубым и приблизительным, но уже оно позволяет выделить достаточно различные и в значительной степени автономные части. Заметим, что кроме перечисленных основных частей в ОС, как правило, выделяют так называемое *ядро* ОС (OS kernel). Через ядро как через всеобщий диспетчер осуществляется передача управления от отдельных запросов на системные функции к частным подсистемам. Ядру обычно поручается и создание управляющих блоков для системных информационных объектов. Подробней понятие ядра будет рассматриваться в одном из следующих разделов.

3.2. Использование прерываний в ОС

Построение всех операционных систем в конечном счете основывается на понятии *прерывания* и его реализации как работающего механизма в совокупности аппаратных и программных средств.

Детальное изучение механизма прерывания возможно только на нижнем уровне детализации программных действий, когда в таком описании используются особенности архитектуры, т.е. логического строения и функционирования аппаратуры компьютера, достаточных для программиста. (Практически архитектура - это точное описание границы между детальным программным и аппаратным обеспечением.) Такое детальное описание использует понятие машинных команд и регистров программиста. К сожалению, подобные описания требуют детализации, которая хорошо излагается только с помощью изобразительных средств языка ассемблера. Изучение последнего представляет само по себе немалый труд.

Поэтому подойдем с другой стороны, частично описательной, но в основном содержательно принципиальной, описывая особенности механизма прерывания с помощью аналогий и близких понятий.

Прерывание – это аппаратно-программное средство, которое предоставляет возможность вызова программ не по имени или указателю, а по *числовому номеру*. Читатели должны быть хорошо знакомы с использованием подпрограмм на языке Си или Паскаль. Подпрограмма некоторых действий описывается отдельно от места ее использования, а затем там, где следует выполнить действия обычной подпрограммы, на языке высокого уровня записывается ее *вызов* путем указания имени и, если нужно, фактических аргументов. Обобщенно это может быть описано на метаязыке в виде

<вызвать подпрограмму> *имя подпрограммы*

Заметим, что после трансляции программы вызовы подпрограмм могут преобразовываться в другую форму, которая явно или неявно использует указатель на место размещения подпрограммы, аналогично тому, как в языке Си применяется доступ к данным через указатель на них. При всей распространенности и привычности для программиста такой подход вызова подпрограмм неприменим для более простой электронной аппаратуры. Аппаратура, по сложности соизмеримая с простым микропроцессором, в лучшем случае может переслать куда-то небольшое число или сигнал. Заметим, что прерывания – это механизм, который предназначен в первую очередь для вызова подпрограмм сигналами непосредственно от аппаратуры.

Если вспомним назначение рассмотренных ранее хэндлов, являющихся как числовые значения небольшими числами, то легко сообразить, что тот же подход может быть использован для перехода от небольших чисел, посылаемых аппара-

турой, к действительным значениям указателей (адресам) подпрограмм. Достаточно где-то в пределах операционной системы построить таблицы, в строки которых записать такие указатели на начало подпрограмм. Правда, дополнительно к таким таблицам нужно дать операционной системе какие-то средства по автоматическому преобразованию номеров вызова подпрограмм через эту таблицу. Эти автоматические средства вкладываются в аппаратуру современных процессоров (уже более 40 лет!) и называются *механизмом прерываний*.

В простейшем случае приказ запуска механизма прерывания записывается прямо в программе. Следует заметить, что никакие языки высокого уровня не позволяют записать непосредственно такой приказ, он допустим только на уровне машинных кодов и команд ассемблера. Подобный вызов называют *программным прерыванием*. В частности архитектура процессоров типа Intel на языке ассемблера записывает такой приказ в виде

INT *номер*

Программные прерывания – это как бы использование чужих по назначению средств в своих программистских целях. На самом деле это очень мощное и используемое внутри ОС средство, но к нему вернемся несколько позже.

Прерывания дают операционным системам неоценимый механизм для вмешательства реального времени и реальных внешних процессов в формализованный мир последовательной обработки информации.

В частности, нажатие клавиши клавиатуры может быть произведено пользователем в любой момент – и через долю секунды и через час, если он займется или отвлечется чем-то другим. В течение всего этого времени (достаточно большой продолжительности для компьютера) целесообразно поручить компьютеру делать что-то более существенное чем просто ждать нажатия на клавищу. Ему можно поручить выполнять другую программу, проигрывать аудизапись и т.п. Прерывание явилось тем средством, с помощью которого внешняя к процессору аппаратура запускает подпрограмму обслуживания событий, вызванных этой аппаратурой.

В состав всех современных компьютеров входит так называемый *таймер*, который через небольшие периодические интервалы времени посылает сигнал, вызывающий через механизм прерывания специальную подпрограмму *обработки прерываний от таймера*. Такие сигналы и вызовы обработки прерываний возникают многие десятки раз в секунду. Человек уже не может уследить за такой частотой появления сигналов, но компьютер между двумя последовательными такими сигналами выполняет многие миллионы внутренних операций и команд.

Для нас существенно, что при возникновении аппаратного прерывания оно может прийтись на любое место выполняемой в этот момент компьютерной программы. Получается, что выполняется обычная программа, которая не требует никаких собственных обращений к подпрограммам, и вдруг неожиданно для нее

происходит переход на чужую подпрограмму обработки прерывания, не предусмотренную в выполняемой. Новичок скажет: "Ну и очень плохо, все испортится!". Ничего подобного! Подпрограмма обработки аппаратного прерывания выполнится, совершит заложенную в нее обработку (например, обновить изображение часов на экране) и "вернет управление" в место прерванной программы. При этом прерванная программа продолжится точно с того места, где она была прервана и притом точно с теми же данными, которые были получены в упомянутом месте. Как технически реализуется запоминание места прерываемой программы, возврат в прерванное место и восстановление ситуации с оперативно обрабатываемыми данными можно понять только на уровне архитектуры и изобразительных средств ассемблера. Для текущего пониманая существенно, что особенности технического механизма прерывания, кроме собственно перехода к подпрограмме обработчика, включают и средства восстановления ситуации для абсолютно точного возвращения ситуации на момент прерывания.

Механизм прерывания можно образно представить следующей картиной. Вы читаете книгу, и вдруг неожиданно раздается телефонный звонок (аналог сигнала от таймера или другой аппаратуры). Вы прерываете свое основное занятие, запомнив место, где приостановили чтение текста, и беретесь за телефонную трубку, а после завершения разговора возвращаетесь к прерванному месту. Заметим, что на основе подобной картины можно представить себе и "вложенные прерывания", когда среди такого телефонного разговора раздается звонок во входную дверь, и вы идете открывать или узнавать, что нужно звонившему, а затем возвращаетесь к прерванному разговору, после завершения последнего – к продолжению чтения.

3.3. Управление системными ресурсами

Ресурсами (в переводе с родного французского языка на русский) называют средства или запасы, используемые для существования, а также источники материальных средств и доходов. Для существования процесса обработки информации в компьютере оказываются необходимыми средства, которые принято обобщенно называть *ресурсами*. Ресурсом является процессор компьютера, ресурсом служит оперативная память, без использования которой невозможно выполнить программу, к ресурсам относят устройства ввода, без которых не может выполняться конкретная программа.

С учетом ограниченности любых технических средств, в частности компьютерных, и их значимости для любого вычислительного процесса, операционная система вынуждена управлять ими. Оказывается, что львиную долю работы ОС можно представить как управляющие действия над ресурсами вычислительной системы. В частности, на первый план при этом выступают такие действия как

создание управляющих блоков для контроля за ресурсами, организация контролируемого доступа к ресурсам для различных процессов, распределение и перераспределение ресурсов, обеспечение согласованного или монопольного использования частями ресурсов.
Значительная часть управляющих блоков информационных объектов внутри ОС как раз и предназначена для ведения распределения и перераспределения ресурсов.
Вычислительные ресурсы в первом приближении могут быть подразделены на *монополизируемые* и *разделяемые*. К монополизируемым ресурсам относится клавиатура. Действительно трудно придумать, как можно с одной клавиатуры направлять вводимые символы в различные единицы вычислительной работы, если явно не перенаправлять средствами ОС вводимые символы от одной единицы к другой. Монополизируемым является и процессор компьютера.
К разделяемым ресурсам относятся магнитные запоминающие устройства, читать с которых или записывать на которые могут почти одновременно несколько независимо выполняемых программ.
Более сложным является сочетание монопольного и разделяемого режимов доступа к ресурсу. Таким примером, далеко не тривиальным, служит оперативная память, которая в современных компьютерах сложным образом перераспределяется между многими процессами, причем часть ее используется совместно многими процессами одновременно, а другие части монопольно перераспределяются отдельным процессам.

3.4 Строение ядра операционной системы

Среди программных компонентов ОС особую роль играет так называемое *ядро* ОС. Ядром ОС называют такую ее явно выделяемую часть, которая отделяется от прикладных программ и других компонентов ОС с помощью аппаратно-программных средств. Эти средства предоставляют современным программам как минимум два уровня прав при выполнении машинных команд: супервизорный и пользовательский. В архитектуре Intel86 заложено даже четыре уровня прав по выполнению машинных команд, называемые *кольцами защиты* от нулевого до третьего. (Заметим, что полностью эти возможности колец защиты практически избыточны и не используются ни в одной из современных ОС.)
Можно описательно сказать, что ядро ОС – это ее глубоко запрятанная центральная часть, в которой выполняются самые ценные действия по управлению компьютером. Сложившийся способ добраться до программных действий ядра представляет собой использование прерываний. При срабатывании аппаратурных средств входа во внутреннюю (аппаратно реализованную) процедуру обра-

ботки прерывания среди прочего происходит смена уровней прав, причем при возврате в прерванную программу права последней восстанавливаются.
Ядро воплощает в себе множество целей проектирования, из которых основными являются обычно ясность, совместимость, переносимость и живучесть. Цель построения ядра должна быть настолько ясной, насколько это возможно в рамках реальных интеллектуальных способностей разработчиков. Ясность построения обеспечивает возможность модифицировать операционную систему, не внося при этом новых ошибок. Практически именно ясность обеспечивает в перспективе живучесть ОС, хотя и не совпадает с последней. Неясно построенные и реализованные в исходной программе части становятся опасным источником непредусмотренного поведения при последующих и неизбежных усовершенствованиях ОС. В то же время ясность неизбежно конфликтует с быстродействием. Когда явно становится видным, что в каком-то компоненте ОС имеется потенциальный конфликт ясности и быстродействия, то предпочтение отдается быстродействию. Алгоритмы, обеспечивающие высокое быстродействие, как правило, достаточно редко удается ясно и понятно реализовать. Тем не менее профессиональные разработчики всегда ставят перед собой цель даже такие алгоритмы представлять в программе по возможности просто и с большим количеством комментариев, упрощающих дальнейшие изменения этой части исходной программы.
Совместимость операционных систем – это возможность выполнять программы одной операционной системы под управлением другой. Ядро Linux поддерживает совместимость путем выполнения исполняемых файлов классов Java, как будто это обычные исполняемые модули Linux. Именно в ядре реализован механизм, делающий такую поддержку прозрачной для пользователей. Операционная система средствами ядра может обеспечивать совместимость путем автоматической возможности запуска и выполнения исполняемых файлов другой операционной системы. Так, в Windows имеется возможность автоматически запускать приложения для MS-DOS и старых 16-битных версий Windows.
Совместимость с аппаратными средствами принято называть *переносимостью* (мобильностью). Совместимость с аппаратурой видеокарт обеспечивается широкими наборами драйверов к этим видеокартам для каждой из широко используемых ОС. Совместимость с аппаратурой процессора неизбежно требует усилий разработчиков ядра ОС. Операционная система Linux имеет наиболее широкий спектр таких применений и работает с процессорами семейств Alpha, Intel, Motorola 680x0, MIPS, PowerPC, SPARC и рядом других, менее известных в России. Операционная система Windows NT кроме работы на процессорах Intel функционирует с процессорами Alpha, PowerPC, MIPS.
Очень большое значение для операционной системы, если только она не ориентирована исключительно на персональное применение, имеет живучесть и на-

дежность. Живучесть и надежность Linux гарантирует открытый процесс разработки системы, при котором каждая строка исходного кода и каждое изменение в нем за считанные минуты исследуются огромным числом разработчиков по всему миру. Часть из них инициативно специализируется на выявлении затаившихся ошибок. Таким образом большинство нарушений безопасности исправляются в течение нескольких дней, а то и часов. Среди профессионалов наиболее безопасной до сих пор считается ОС OpenBSD – одна из вариаций Unix, в которой основной целью ставилась именно безопасность.

Любое краткое описание строения ядра ОС является неизбежным упрощением, которое теряет многие существенные детали. Поэтому к структурному описанию ОС не нужно относиться как к точному описанию – оно неизбежно приближенное.

В настоящее время укрупненную структуру ОС принято рассматривать либо как многослойную, либо как модульную. В многослойной структуре обычно выделяют следующие слои: системно-пользовательские приложения (Explorer в Windows, Midnight Commander в Linux и т.п.), системные библиотеки, архитектурно-независимую часть ядра и архитектурно-зависимую часть ядра. В более детальном разбиении можно последовательно выделить слой системных сервисов (реализуемых системными библиотеками), файловую систему, слой управления памятью и устройствами ввода-вывода, а также "полуслой" планирования процессов (предыдущий слой взаимодействует с аппаратурой напрямую).

При разбиении на модули обычно выделяют, кроме отдельных системно-пользовательских приложений, системные библиотеки, интерфейс системных вызовов, подсистемы управления процессами, памятью, планировщик процессора и т.п.

Другой подход к описанию ядра является более концептуальным. В соответствии с ним ядро может быть либо *монолитным*, либо использовать *микроядро*. В последнем случае большая часть функций ядра реализуется как процессы (высокоприоритетные системные процессы) с помощью небольшого набора функций микроядра. Функции же последнего представляют собой что-то вроде "детского конструктора" со множеством элементарных и удобных деталей. Системные функции собственно операционной системы реализуются как последовательности вызовов функций микроядра.

При использовании микроядра реализация функций ОС представляет собой множество процессов, которые по свой природе достаточно сильно изолированы друг от друга, в частности, не могут непосредственно использовать подпрограммы из другого процесса. Поэтому взаимодействие между функционирующими частями ОС вынужденно осуществляется через посылку сообщений. Использование микроядра имеет следствием клиент-серверную структуру построения самой операционной системы. При этом клиентами микроядра (или вспомо-

гательного слоя между микроядром и этими клиентами) оказываются системные процессы, среди которых можно назвать следующие: сервер памяти, сервер сети, сервер файлов, сервер графической системы (дисплея), сервер процессов, который непосредственно занимается созданием и операциями над процессами.
Практически функции микроядра предоставляют то общее, что удается выделить в множестве однотипных системных функций у разных ОС. Поэтому, используя микроядро, относительно просто реализовать наборы системных функций не только для одной, но и для набора ОС. Появляется возможность проводить будущие модификации не только полной переделкой отдельных функций ядра, а и путем модификаций вызовов внутренних функций микроядра. Преимуществом микроядра является возможность разрабатывать новые системные модули и добавлять их даже в процессе функционирования самой ОС. Другое достоинство этого подхода заключается в более эффективном использовании памяти, так как неиспользуемые модули просто не загружаются.
Использование микроядра неизбежно влечет дополнительный вызов (чаще вызовы) подпрограмм, реализующих элементарные действия, но обеспечивает большой функциональный запас для модификации операционной системы. Операционная система Windows NT (в отличие от Windows 9x) с самого начала проектировалась как использующая микроядро. Именно на основе элементарных функций микроядра основана реализация совместимости с программами 16-битной Windows, MS-DOS и минимальным стандартном POSIX.
Монолитное ядро – это один большой системный процесс. Вместо отправки сообщений модули монолитного ядра взаимодействуют за счет обычных вызовов подпрограмм.
В Linux ядро в основном – монолитное. Это определилось авторским решением первой версии Linux (Линусом Торвальдсом – Linus Torvalds) которое объясняется тем, что монолитное ядро имеет более простую и ясную структуру, для него не приходится изобретать средства передачи сообщений, разрабатывать методы загрузки внутренних моделей и т.п. По существу утверждается, что асинхронный порядок взаимодействий в варианте с микроядром динамически слишком сложен, чтобы ожидать высокую надежность достижимого программного решения для всех внутренних функций операционной системы. Вариант микроядра, с одной стороны, дает возможность более высокой гибкости модификации ОС, но используемые дополнительные средства взаимодействий между внутренними процессами принципиально снижают его быстродействие. С другой стороны, микроядро очень небольшого объема может размещаться во внутреннем кэше современного процессора, что, в свою очередь, может существенно повысить быстродействие. Именно такое решение использовано в ОС реального времени QNX.

В операционных системах типа Windows на первое место выступают цели удобства для непрофессиональных пользователей (в поздних версиях даже цели удобства для очень малограмотных пользователей) и быстрая реакция ОС при манипуляциях в графической оболочке, которым в жертву приносятся все остальные.

3.5. Структура операционной системы типа Windows NT

Структура операционных систем типа Windows NT укрупнено представлена на рис. 3.1.

На схеме блоки "Клиент Win32" и "Клиент POSIX" представляют пользовательские процессы, использующие ОС, но в саму операционную систему не входят. Слой абстрагирования (Hardware Abstraction Layer) представляет собой совокупность программ, работающих непосредственно с аппаратурой, и его реализация зависит непосредственно от типа используемых процессоров. Этот слой изолирует верхние слои ОС от платформенно-зависимых особенностей аппаратуры.

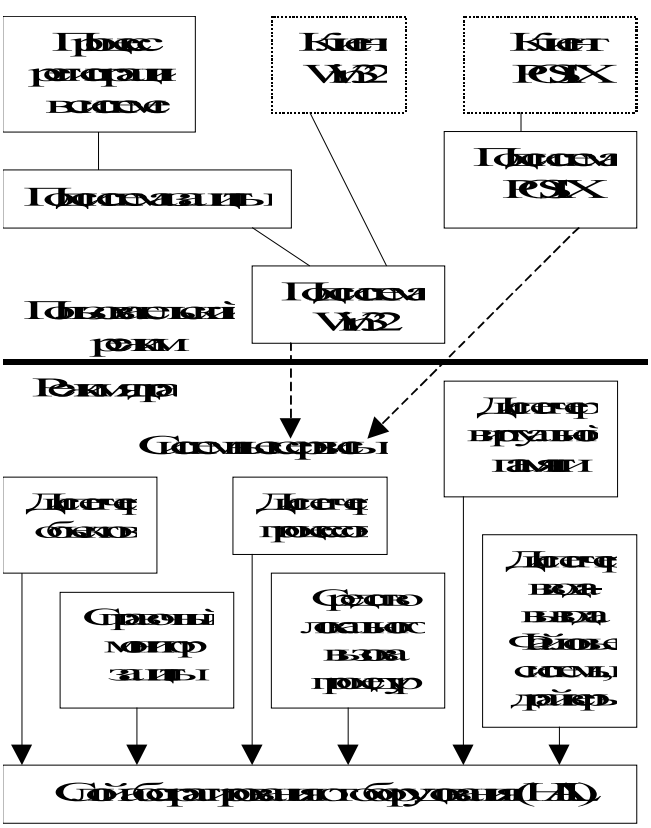


Рис. 3.1. Упрощенная структура ОС типа Windows NT

Структура ОС Windows NT распадается на две существенно отличные части, верхняя по изображенной схеме работает в обычном пользовательском режиме аппаратуры, а нижняя - в состоянии режима ядра. Переход к режиму ядра на

уровне машинных команд осуществляется с помощью прерываний. При этом принципиально меняются возможности использования некоторых операций, в частности появляется возможность непосредственно работать с аппаратурой.
Часть этой ОС, работающая в режиме ядра, за исключением слоя HAL разработчики назвали исполнительной системой (NT executive). В этом наименовании, в частности, отразилась традиция Microsoft, связанная с предшествующей квазиоперационной системой Windows 3.x, в которой основная часть, отличная от графической оболочки, называлась подобным образом.
Все остальные блоки рассматриваемой схемы, за исключением клиентов, HAL и блока "Средство локального вызова процедур", в процессе функционирования представляют собой отдельные процессы, причем процессы исполнительной части и естественно привилегированные. Средства локального вызова представляют собой специализированные средства межпроцессного взаимодействия, ориентированные на взаимодействие процессов самой операционной системы.
Заметим, что в поздних версиях операционной системы Windows NT сервер графической системы, обозначаемый как программный компонент GDI (Graphic Device Interface), перемещен из области пользовательского режима в область ядра. Это сделано, начиная с ОС Windows 2000. Такое решение призвано ускорить видимые операции графического интерфейса, но принципиально снижает надежность всей операционной системы.

4. МНОГОФУНКЦИОНАЛЬНЫЙ КОНСОЛЬНЫЙ ВЫВОД

4.1. Функции управления курсором

Классические языки высокого уровня не содержат средств управления позицией вывода на экране и цветом символов текста. Такая ситуация в значительной степени предопределена тем, что когда создавались эти языки, подобные средства были недоступны по аппаратным причинам. Именно глубокая связь упомянутых возможностей с конкретной аппаратурой и привела к тому, что подобные средства оказались зависимыми от операционной системы. К настоящему времени подавляющее большинство мониторов поддерживают как позиционирование курсора, так и многоцветные изображения, но особенности управления этими возможностями по-прежнему оказываются зависимыми от операционных систем.
С содержательно-функциональной стороны для позиционирования (установки в некоторую позицию) курсора необходимо задать экранные координаты этой позиции. Наиболее просто и естественно это задается в OS/2. Здесь для этих целей служила функция с прототипом

APIRET16 VioSetCurPos(USHORT row, USHORT col, HVIO hvio).

Заметим, что все функции, осуществляющие управление экраном в этой ОС, имели префиксом названия буквосочетание Vio и возвращали 16-битное значение кода возврата, определяющее успешное выполнение функции как нулевое значение, ненулевые значения зарезервированы для кодов ошибок. Аргументами этой функции служили номера строки и столбца, в которые требовалось установить текстовый курсор. Последний аргумент hvio предполагалось использовать при множестве мониторов, подключенных к одному системному блоку компьютера. При единственном мониторе этот параметр должен быть равен 0. (Теоретически его задает хэндл монитора, для которого выполняется позиционирование.)

Обратной функцией к описанной являлась функция с прототипом

APIRET16 VioGetCurPos(USHORT *prow, USHORT *pcol, HVIO hvio).

Она возвращала значения номеров строки и столбца, в которых находится курсор.

Существенно отличается позиционирование курсора в ОС Windows. Здесь функция позиционирования имеет прототип

BOOL SetConsoleCursorPosition(HANDLE hConsOut, COORD pos).

Первым ее аргументом служит хэндл экрана консоли, а второй задает устанавливаемую позицию курсора и представляет собой структуру данных, описанную в заголовочном файле wincom.h как

```
typedef struct _COORD {
   SHORT X;
   SHORT Y;
} COORD, *PCOORD;
```

Ее использование требует от программиста несколько больших усилий. Так, например, установка курсора в третью строку и пятый столбец при работе в OS/2 требует единственной записи вызова функции в виде VioSetCurPos(3, 5, 0), а аналогичное указание позиции курсора для консоли ОС Windows вынуждает предварительно описать экземпляр структуры типа COORD, например в виде

```
COORD pos;
```

а затем задать запись целых трех операторов

```
pos.X=5; pos.Y=3;
SetConsoleCursorPosition(hout, &pos);
```

Обратная к установке функция, позволяющая определить текущую позицию курсора, описывается в Windows прототипом

```
BOOL GetConsoleScreenBufferInfo(HANDLE hConsOut,
   CONSOLE_SCREEN_BUFFER_INFO* pConsoleScreenBufferInfo)
```

и требует использования структуры данных, описанной в заголовочном файле как

```
typedef struct _CONSOLE_SCREEN_BUFFER_INFO {
```

```
    COORD dwSize;
    COORD dwCursorPosition;
    WORD  wAttributes;
    SMALL_RECT srWindow;
    COORD dwMaximumWindowSize;
} CONSOLE_SCREEN_BUFFER_INFO, *PCONSOLE_SCREEN_BUFFER_INFO;
```

В этой структуре данных для рассматриваемых целей поле dwCursorPosition предоставляет информацию о положении курсора.

В операционной системе Unix для управления курсором и некоторых других действий с экраном предназначены *управляющие последовательности*. Идея их использования расширяет управляющие символы, которые в языке Си и Unix служат основным средством управления выводом на экран. Управляющие последовательности определяются стандартом ANSI и называются также ANSI-последовательностями. Их можно использовать и в других ОС, но там часто для этого оказывается необходимым запустить ANSI-драйвер. В MS DOS этот драйвер должен был быть установлен в конфигурационном файле CONFIG.SYS, в OS/2 – запущен как отдельная программа ANSI.EXE.

Управляющие последовательности ANSI начинаются со специального символа с десятичным эквивалентом значения 27. Это код, выдаваемый клавишей Esc, и при записи на языке Си в составе текстовых констант его записывают в виде '\033'. При этом задействована универсальная форма записи произвольных (в том числе явно не изображаемых символов) в виде восьмеричных констант. Вторым символом управляющих последовательностей ANSI является обязательный символ ‘[‘ (открывающаяся квадратная скобка), последним символом управляющей последовательности – латинская буква, детализирующая операцию. Иногда для такой детализации используется и предпоследний символ.

Для задания установки курсора служит управляющая последовательность, записываемая на языке Си как текстовая константа

"\033[*строка*;*столбец*H"

Здесь компоненты *строка* и *столбец* должны быть обязательно заданы десятичными числами и обязательно без дополнительных пробелов. В дальнейшем записывать управляющие последовательности (как принято в документации) будем без подразумеваемых кавычек, а служебный символ \033 условно записывать как *esc*, подразумевая, что это все-таки не три латинских буквы подряд, а один специальный символ. Нумерация позиции считается от 1, так что установка в верхний левый угол экрана требует последовательности *esc*[1;1H .

Для чтения позиции курсора имеется также специальная управляющая последовательность, но использование ее не очень удобно, а главное в ней практически нет необходимости при наличии таких интересных и удобных управляющих последовательностей, какие просто переводят курсор на заданное число строк и

столбцов в вертикальном и горизонтальном направлениях. Для этих целей предназначены управляющие последовательности *esc*[*строка*A, *esc*[*строка*B, *esc*[*столбец*C, *esc*[*столбец*D, последние символы в которых обязательно прописные латинские буквы.

Последовательность *esc*[*строка*A приказывает переместить курсор на заданное в ней число строк вверх, последовательность *esc*[*строка*B – на заданное число строк вниз, *esc*[*столбец*C – на заданное в ней число столбцов вправо, а *esc*[*столбец*D – на заданное число столбцов влево. Если при заданных значениях параметров курсор должен выйти за пределы экрана, то действие управляющей последовательности игнорируется.

4.2. Многократный вывод символов и атрибутов

В современных ОС имеются специальные функции для многократного вывода однотипной информации, иногда облегчающие программисту его работу. Рассмотрим вначале средства многократного вывода одного символа. В OS/2 содержатся две системных функции, выполняющих эту задачу. Одна из них позволяет задавать символ вместе с атрибутом для вывода и называется VioWrtNCell. Другая не содержит информации об атрибуте выводимого символа и называется VioWrtNChar. Для обеих, если при их выполнении достигается конец экрана, вывод прекращается.

Функция VioWrtNCell имеет на языке Си прототип

```
APIRET16  VioWrtNCell(BYTE *pCell, USHORT len,
        USHORT row, USHORT col, HVIO hvio);
```

где *row* – номер строки; а *col* – номер колонки позиции экрана, с которой начинается вывод (в случае одного символа – куда осуществляется вывод); *len* – число повторений вывода символа (сколько раз подряд заданный символ будет выводиться на экран); *pCell* – адрес ячейки из двух байтов, задающий информацию о символе для вывода, причем младший байт должен содержат ASCII код символа, а старший байт – атрибуты вывода.

Функция VioWrtNChar имеет на языке Си прототип

```
APIRET16  VioWrtNChar(BYTE *pChar, USHORT len,
                        USHORT row, USHORT col, HVIO hvio);
```

где *row* – номер строки, а *col* – номер колонки позиции экрана, с которой начинается вывод (в случае одного символа – куда осуществляется вывод), *len* – число повторений вывода символа (сколько раз подряд заданный символ будет выводиться на экран), *pChar* - адрес байта, хранящего символ.

Дополнением к возможностям только что рассмотренных функций служит функция VioWrtNAttr. Она позволяет предварительно установить новые атрибуты для текста или изменить старые атрибуты уже выведенного текста для после-

довательных позиций экрана. Место экрана, начиная с которого следует изменить атрибуты, указывается в вызове функции VioWrtNAttr.

Функция VioWrtNAttr имеет на языке Си прототип

```
APIRET16  VioWrtNAttr(BYTE *pAttr, USHORT len,
                      USHORT row, USHORT col, HVIO hvio);
```

где *row* – номер строки, а *col* – номер колонки позиции экрана, с которой задаются (изменяются) атрибуты, *len* – число мест для символов, в которых изменяются атрибуты, *pAtrr* – адрес байта, хранящего атрибут.

Для многократного вывода одного символа в Windows предназначена функция FillConsoleOutputCharacter, а для многократного вывода одного и того же атрибута – функция FillConsoleOutputAttribute. Эти функции имеют прототипы

```
BOOL FillConsoleOutputCharacter(HANDLE hConsOut,
                CHAR character, WORD len, COORD pos, DWORD* actlen);
BOOL FillConsoleOutputAttribute(HANDLE hConsOut,
                WORD attr, DWORD len, COORD pos, DWORD* actlen);
```

Первые аргументы этих функций задают хэндл экрана консоли (более точно экранного буфера), вторые соответственно – выводимый символ или выводимый атрибут, параметр *len* задает число повторений вывода этого символа или атрибута, а параметр *actlen* – адрес для возврата числа действительно выполненного числа повторений. Это число может отличаться от заказанного в вызове системной функции, если в процессе вывода достигнут конец экрана. Параметр *pos* задает позицию экрана, начиная с которой эти функции выполняют свой вывод.

Для задания атрибутов в Windows можно использовать предопределенные символические константы, которые заданы в заголовочном файле wincon.h. Эти константы имеют названия FOREGROUND_BLUE, FOREGROUND_GREEN, FOREGROUND_RED, FOREGROUND_INTENSITY, BACKGROUND_BLUE, BACKGROUND_GREEN, BACKGROUND_RED, BACKGROUND_INTENSITY, которые говорят сами за себя. Для получения комбинированного цвета с их помощью достаточно несколько требуемых из них соединить символами побитовой операции ИЛИ. Так, например, для задания белового цвета символа следует указать операнд в виде FOREGROUND_RED | FOREGROUND_GREEN | FOREGROUND_BLUE. (Нетрудно видеть, что этот подход достаточно громоздкий для нормального программиста, поэтому неплохо представлять, что значения перечисленных выше констант есть соответственно 1, 2, 4, 8, 16, 32, 64 и 128.)

В Windows имеется еще одна возможность установки цвета для вывода на экран консоли. Ее предоставляет функция с прототипом

```
BOOL SetConsoleTextAttribute(HANDLE houtput, WORD attrib).
```

Эта функция устанавливает цвет, автоматически используемый далее при выводе на экран функцией WriteFile (и также более специализированной функцией WriteConsole), а так жеиспользуемый при отображении символов в процессе ввода функцией ReadFile. Кроме того, такая установка оказывается действующей и на те функции, которые в действительности построены на основе указанных. В то же время эта установка не действует на некоторые другие функции вывода, которые будут рассматриваться далее. (В частности на вывод в произвольную позицию экрана с помощью функции WriteConsoleOutput.)

В операционной системе Unix отсутствуют системные функции многократного вывода символов и атрибутов. Желающие, впрочем, легко могут написать их для себя, опираясь на стандартные средства этой операционной системы.

Для современного программирования непременной особенностью является широкое использование цветовых возможностей мониторов. Достаточно скромные средства для этого в Unix связана с тем, что эта ОС становилась и внутренне развивалась в более ранний период, когда наличие цветного монитора было редкой особенностью.

Проблема использования цвета в Unix была решена с помощью управляющих последовательностей. Для задания цвета используются управляющие последовательности, последним символом которых служит латинская буква **m**. Сама управляющая последовательность имеет вид

esc[*цвет***m**

где компонент *цвет* задается десятичным числом согласно табл. 4.1.

Табл. 4.1.Кодирование цвета в управляющих последовательностях

Цвет символа, фона и, может быть, повышенной яркости можно задавать в одной управляющей последовательности, разделяя их символами "точка с запятой" и обязательно без дополнительных разделяющих пробелов. Например задание ярко-желтого символа на синем фоне, можно получить управляющей последовательностью

esc[1;33;44m

Цвет, установленный управляющей последовательностью, действует до тех пор, пока другой подобной последовательностью он не будет переустановлен. В частности, после завершения работы программы очень удобно вернуться к стандартному цвету использования консоли: светло-серым символам на черном фоне. Для такого перехода проще всего выполнить управляющую последовательность esc[0**m .**

Заметим, что параметры 30-47 соответствуют стандарту 6429 Международной организации по стандартизации (ISO). Нетрудно видеть, что средства Unix обеспечивают достаточно компактные средства управления цветом, но, к сожалению, для начинающих очень не наглядные.

4.3. Вывод в произвольную позицию экрана

Для вывода строки текста в произвольную позицию экрана операционная система OS/2 содержит функцию VioWrtCharStr, которую часто было удобней использовать, чем вначале отдельной функцией позиционировать курсор, а затем выводить текст функцией вывода. Кроме того, иногда возникала потребность выводить изображения символов, которые режим вывода считал управляющими. Во всех этих случаях было целесообразно в OS/2 использовать функцию, которая имела прототип

APIRET16 VioWrtCharStr(PCH text, USHORT len, USHORT row,
USHORT col, HVIO hvio).

В этой функции параметр *text* задавал адрес строки выводимого текста, параметр *len* – длину этого текста в байтах, параметры *row* и *col* давали номера строки и столбца позиции экрана, начиная с которой производится вывод. Если при выполнении функции достигался конец экрана, то вывод прекращался. Число действительно выведенных символов выдавалось как возвращаемое значение функции.

При использовании рассматриваемой функции атрибуты для выводимых символов брались такие, какие ранее были установлены для позиций, куда осуществлялся вывод. Эта установка могла остаться от стандартного режима использования консоли (серый цвет на черном фоне), и задана рассмотренной выше функцией VioWrtNAttr или иными функциями.

В операционной системе Windows для вывода строки текста в произвольную позицию экрана служит функция

```
BOOL WriteConsoleOutputCharacter(HANDLE hConsOut,
                CSTR* text, DWORD len, COORD pos, DWORD* actlen);
```

Параметры, используемые в функции WriteConsoleOutputCharacter, имеют те же назначения, что и одноименные - описанные ранее в предыдущих функциях. Парное сочетание функций FillConsoleOutputAttribute и WriteConsoleOutputCharacter характерно для консольных приложений Windows, в которых требуется выполнять вывод с новым цветом в непоследовательных позициях экрана.

Заметим, что в Windows нет системной функции, объединяющей указанные действия функций FillConsoleOutputAttribute и WriteConsoleOutputCharacter. В операционной же системе OS/2 такая функция была запасена и называлась VioWrtCharStrAtt. Прототип функции описывался на языке Си как

```
APIRET16  VioWrtCharStrAtt(PCH text, USHORT len, USHORT row,
                USHORT col, PBYTE attr, HVIO hvio);
```

Первый аргумент *text* задавал выводимый текст и указывался в вызове своим адресом; второй аргумент *len* должен был задаваться коротким словом и определял длину выводимого текста. Следующие два аргумента определяли позицию начала текста на экране. Третий аргумент *row* определял номер строки на экране, а четвертый *col* – номер колонки для позиции, откуда начинал выводиться текст. Пятый аргумент *attr* определял байт атрибутов для текста и должен был задаваться адресом в памяти, где этот атрибут записан.

В отдельных ситуациях, хотя и довольно редко, требуется осуществлять вывод текста, цветовые атрибуты символов которого должны быть отличными друг от друга. Теоретически такой вывод можно разбить на множество частей, внутри каждой из которых осуществляется вывод участка текста с постоянным для участка атрибутом, но такой прием очень не эффективен, если почти каждый последовательный символ требует другого атрибута. Для решения подобных задач в OS/2 была предназначена функция VioWrtCellStr, которая дает возможность выводить текстовую строку, в которой атрибут для каждого символа задается независимо от других.

Рассматриваемая функция предоставляла наибольшие изобразительные возможности, но для ее использования текст должен был быть размещен в памяти так, что за каждым его символом следует атрибут вывода. Это позволяло разнооб-

разить вывод, задавая для каждого символа свой цветовой вариант, но требовало от программиста дополнительных усилий при таком детальном описании текста. Например, для вывода текста 'Привет', в котором первая буква должна отобразиться синим цветом на ярко-красном фоне, вторая буква – зеленым цветом на черном фоне, третья буква – ярко-бирюзовым цветом на черном фоне, пятая буква – желтым цветом на синем фоне и последняя буква – ярко-синим цветом на желтом фоне, следует этот текст задать описанием вида

```
char celltext[ ]={'П', 0xc1, 'р', 0x2, 'и', 0xb, 'в', 0x1e, 'е', 0x2c, 'т', 0xe9};
```

В операционной системе Windows рассмотренной функции вывода символов вместе с их атрибутами соответствует функция

```
BOOL WriteConsoleOutput(HANDLE hConsOut, CHAR_INFO* cells,
            COORD dwBufferSize, COORD dwBufferCoord, SMALL_RECT*
rect);
```

Возможности этой функции значительно шире, чем будет нами рассмотрено, и позволяют вернуть на экран, не обязательно в то же его место, запомненный ранее в памяти фрагмент текстового изображения экрана с полным учетом цветовых атрибутов. Функция эта в качестве исходной отображаемой информации использует двухмерный массив (а не одномерный как в OS/2) ячеек, хранящих всю отображаемую информацию о символах на экране. Этот массив задается для функции адресом в параметре cells. Его элементы должны быть описаны как имеющие тип CHAR_INFO, который в заголовочном файле задается описанием

```
typedef struct _CHAR_INFO {
   union {
      WCHAR UnicodeChar;
      CHAR   AsciiChar;
   } Char;
   WORD Attributes;
} CHAR_INFO, *PCHAR_INFO;
```

Параметр *dwBufferSize* рассматриваемой функции задает измерения этого массива: сколько в нем строк и сколько столбцов, а параметр *dwBufferCoord* определяет позицию в этом массиве, начиная с которой его элементы выводятся на экран как символы с их атрибутами. Последний параметр *rect* функции задает прямоугольную область экрана, в которую предназначен вывод текста с его атрибутами. Тип этого параметра описана в заголовочном файле как

```
typedef struct _SMALL_RECT {
   SHORT Left;
   SHORT Top;
   SHORT Right;
   SHORT Bottom;
} SMALL_RECT, *PSMALL_RECT;
```

и позволяет определить все четыре вершины используемого прямоугольника. После выполнения функции параметр *rect* возвращает информацию о вершинах прямоугольника экрана, действительно заполненного этой функцией (им может оказаться прямоугольник, отличный от заданного вначале, если в процессе вывода мог произойти выход изображения за пределы экрана).

В операционных системах Windows имеется функция для вывода на экран, не имеющая аналога в OS/2. Это функция WriteConsoleOutputAttribute, предназначенная для вывода последовательности различных атрибутов на экран. Целесообразность выделения подобных действий в отдельную функцию представляется весьма сомнительной, но может быть кто-то из читателей найдет ей удачное применение в своих разработках. (Практически возможности этой функции перекрываются в OS/2 выше рассмотренной функцией VioWrtCellStr.) Прототип функции WriteConsoleOutputAttribute дается описанием

```
BOOL  WriteConsoleOutputAttribute(HANDLE hConsOut,
            WORD* attrs,  DWORD len,  COORD pos, DWORD* actlen);
```

Параметр *attrs* задает массив атрибутов, элементами которого являются слова (а не байты как в OS/2 и MS DOS), содержащие атрибуты. Параметр *len* определяет размер используемого массива (точнее используемой части массива); *pos* - задает позицию экрана, начиная с которой выполняется заполнение атрибутами, параметр *actlen* используется для возвращения числа действительно записанных атрибутов.

В операционной системе Unix для вывода строки текста с различными атрибутами следует использовать управляющие последовательности между отдельными выводимыми символами. Так, для вывода в описанных выше цветах текста "Привет" требуется построение следующей текстовой константы:

```
"\033[1;31;44mП\033[0;32;40mр\033[1;36;40mи\033[33;44mв
\033[31;42me\033[34;43mт\n"
```

В этой последовательности комбинируются отдельные символы текста вместе с требуемыми для них атрибутами. Заметим еще раз, что в Unix достаточно просто – по структуре используемых средств и по размеру исходного текста - решаются задачи, которые в других ОС требуют значительно больше вызовов системных функций. Программы же для Unix часто недостаточно наглядны, что особенно ощутимо для непрофессионалов.

В частности, для очистки экрана Unix содержит специальную управляющую последовательность, которая определяется как

```
esc[2J
```

После очистки экрана, вызванного этой управляющей последовательностью, курсор помещается в левый верхний угол. Кроме очистки экрана имеется еще одна часто удобная последовательность – очистить строку, задаваемую как esc[K . Это

управление удаляет все символы, начиная с позиции курсора до конца строки, включая символ в позиции курсора.

4.4. Ввод данных, размещенных предварительно на экране

Для чтения в OS/2 текстовой информации с экрана служат две функции: VioReadCharStr и VioReadCellStr.

Первая из них читает в программу (т.е. передает в вызывающую программу) только собственно символы с указанного в вызове места экрана, а вторая – для каждого читаемого с экрана символа дополнительно извлекает и его атрибут, так что из экрана извлекается предельно полная информация о тексте.

Прототипы этих функций имеют вид

```
APIRET16  VioReadCharStr(PCH text, PUSHORT len, USHORT row,
                         USHORT col, HVIO hvio);
APIRET16  VioReadCellStr(PCH cells, PUSHORT len, USHORT row,
                         USHORT col, HVIO hvio);
                                  word ptr row, word ptr col, word ptr 0);
```

Эти функции возвращают прочитанный текст в область памяти, заданную адресом в первом аргументе, причем функция VioReadCharStr возвращает только символы текста, а функция VioReadCellStr – последовательность пар "символ-атрибут". Второй аргумент *len* вызова функции, во-первых, задает, сколько символов требуется прочитать с экрана, а, во-вторых, после выполнения системного вызова дает, сколько символов удалось прочитать. Системные функции чтения могут вынужденно прочитать меньше символов с экрана, чем задано для чтения, если при чтении достигнут конец экрана (правый нижний его угол). Для выполнения этого аргумент сделан возвращаемым.

В Windows для чтения информации с экрана имеется даже три функции. Это функция ReadConsoleOutputCharacter – для чтения с экрана только собственно символов текста, функция ReadConsoleOutputCharacter – для чтения с экрана символов текста вместе с их атрибутами и функция ReadConsoleOutputAttribute – для чтения из последовательных позиций экрана атрибутов символов. (Заметим, что в OS/2 нет аналога последней функции.) Перечисленные функции имеют следующие прототипы:

```
BOOL  ReadConsoleOutputCharacter(HANDLE hConsOut,
                         STR* buffer, DWORD len, COORD dwReadCoord,
                         DWORD* actlen);
BOOL ReadConsoleOutput(HANDLE hConsOut,
                         CHAR_INFO* cells, COORD dwBufferSize,
                         COORD dwBufferCoord, SMALL_RECT* rect);
```

BOOL ReadConsoleOutputAttribute(HANDLE hConsOut,
WORD* attribs, DWORD len, COORD pos, DWORD*
actlen);

Вторая из перечисленных функций является обратной к функции с именем WriteConsoleOutput, которая в свою очередь предназначена для размещения на экране информации из двухмерного массива элементов, состоящих из пары (символ, его атрибут). Практически все параметры этих функций совпадают по использованным здесь обозначениям, по их типам и назначению. Вся разница между этими функциями в том, что одна из них переносит информацию из указанного в них прямоугольника экрана в двухмерный массив данных (функция ReadConsoleOutput), а другая осуществляет перенос информации в противоположном направлении: из двухмерного массива элементов – в указанный прямоугольник экрана. В обоих случаях параметр *rect* определяет используемый в описании прямоугольник экрана; параметр *cells* – массив элементов (ячеек для пар символ-атрибут); параметр *dwBufferSize* – как размеры массива, а *dwBufferCoord* – позицию (индекс строки и столбца) того элемента массива, начиная с которого используется массив в конкретном вызове рассматриваемых функций.

Функция ReadConsoleOutputCharacter выполняет чтение только символов в буфер программы с именем *buffer* (или по адресу, который дает указатель *buffer* – в зависимости как конкретно определено это имя), а функция ReadConsoleOutput-Attribute выполняет чтение только атрибутов в массив слов, указанный параметром *attribs*. Число читаемых при этом символов или атрибутов задается параметром *len*, чтение выполняется с начальной позиции экрана, задаваемой параметром с именем *pos*, а действительное число прочитанных символов или атрибутов возвращается с помощью параметра *actlen*.

В операционной системе Unix отсутствуют стандартные средства чтения символов с экрана. Это объясняется тем, что основные принципы работы с Unix формировались еще в то время, когда основным средством вывода информации для отдельного пользователя интерактивной ОС были телетайпы, осуществляющие вывод на бумажную ленту, считывать информацию с которой было невозможно по техническим причинам. Как показывает опыт работы в этой ОС, без средств чтения информации с экрана вполне можно обойтись без снижения потребительских возможностей операционной системы и прикладных программ.

В то же время современные версии Unix, в частности Linux, ориентируясь на возможности современных электронных устройств вывода информации пользователю, предлагают дополнительные виртуальные устройства. Одним из таких устройств является "виртуальный экран консоли", обозначаемый в файловой системе Unix как vcs (сокращение от virtual consoe screen). В связи с замечательными особенностями файловой системы Unix устройства в ней рассматриваются как файлы, только специализированные. Это позволяет использовать операцию откры-

тия файлов для доступа к устройствам. Заметим, что специальные файлы устройств стандартным образом размещаются в каталоге /dev, поэтому полное наименование устройства виртуального экрана консоли следует задавать в виде /dev/vcs.

Практически в настоящее время имеется целый набор виртуальных экранов консолей на весь их возможный набор. Когда желательно иметь непосредственный доступ к виртуальному экрану для текущей консоли, с которой работает программа или оператор, рекомендуется использовать виртуальное устройство, обозначаемое как vcs. Это виртуальное устройство, рассматриваемое как файл, отображает содержимое видеопамяти для экрана данной виртуальной консоли. Каждой позиции экрана в этом файле соответствует один байт. Поэтому при использовании типового разрешения экрана в 25 строк по 80 символов виртуальный экран консоли vcs хранится как файл из 2000 байтов (25x80).

Кроме виртуального экрана консоли vcs, можно использовать расширенное виртуальное устройство, обозначаемое как vcsa, которое позволяет считывать из видеопамяти не только коды символов, но и их атрибуты. На каждую позицию экрана в этой файле имеет два байта: один хранит код самого символа, а второй (следующий) код атрибутов этого символа. Кроме того, в файле vcsa - в его начале размещаются четыре служебных байта, хранящие, соответственно, число строк экрана, число столбцов в нем, а далее номер столбца и номер строки, в которых находится курсор экрана. Поэтому при использовании типового разрешения экрана в 25 строк по 80 символов, виртуальный экран консоли vcsa хранит как файл 4004 байтов (25x80x2+4).

Следующая программа, приведенная листингом 4.4.1, демонстрирует сохранение текущего содержимого экрана в файле с именем myscreen.

```
#include <stdio.h>
#include <fcntl.h>
void main()
{int screena, fhandle;
 int rc;
 char buffer[256];

        screena=open("/dev/vcs",O_RDONLY);
        if (screena= =-1) {printf("Error open virtual screen\n"); exit(0);}
        fhandle=open("myscreen",O_WRONLY|O_CREAT, 0764);
        while ((rc=read(screena,buffer,256))!=0)
                {write(fhandle, buffer,rc);}
        close(screena);
        close(fhandle);
```

}

Листинг 4.4.1. Чтение из виртуального устройства экрана

Использование виртуального экрана консоли, как файла, из которого можно оперативно извлекать содержимое любой позиции экрана, позволяет решать те же задачи, что и функции прямого чтения видеобуфера, применяющиеся в Windows.

Следует только заметить, что для непривилегированных пользователей Linux устройство vcs может оказаться недоступным по чтению, если об этом не побеспокоиться на уровне общесистемных установок. (При недоступном устройстве возвращается ошибка открытия файла.) Чтобы сделать это устройство доступным по чтению любым пользователям следует в режиме привилегированного пользователя (обычно пользователя с именем root) выполнить команду

chmod o+r /dev/vcs

Проверить результат можно с помощью команды Unix, которую следует задать в виде

ls -l /dev/vcs

Она должна выдать сообщение вида

crw----r-- 1 root tty *номер*, *номер дата* /dev/vcs

(Вместо дефисов здесь могут быть (хотя и маловероятно) какие-то другие символы.)

Аналогичное действие следует выполнить и над устройством vcsa, если предполагается использовать последнее.

5. СИСТЕМНЫЕ ФУНКЦИИ ВВОДА ДЛЯ КОНСОЛЬНЫХ УСТРОЙСТВ

5.1. Системные функции ввода текстовых строк

Как ни странно, ввод текстовых строк организуется для пользователей системных функций проще чем ввод отдельных символов. Поэтому с ввода текстовых строк и начнем.

В операционных системах MS Windows для ввода текстовых строк с клавиатуры в консольном режиме предназначена функция ReadConsole с прототипом

```
BOOL ReadConsole(HANDLE hConsInput, VOID* buffer,
                 DWORD len, DWORD *actlen, VOID* reserved);
```

Первый параметр этой функции задает хэндл буфера ввода для консоли. Он в простейших (и большинстве) случаях получается в результате вызова функции GetStdHandle с аргументом STD_INPUT_HANDLE. Параметр *buffer* задает буфер для ввода текстовой строки, а параметр *len* определяет максимальный размер вводимой строки (обычно этот параметр совпадает с длиной буфера *buffer*). Параметр

actlen используется для получения числа – сколько символов было введено в результате выполнения функции. (В число символов входит и завершающий символ '\n'.) Последний параметр вызова функции зарезервирован разработчиками и должен браться равным *NULL*.

Заметим, что рассмотренная функция дает возможность ввода с консоли только в том случае, когда стандартный ввод не переназначен. Если же это условие нарушено, то данная функция возвращает значение FALSE, свидетельствующее об ошибке. При необходимости в Windows осуществлять ввод только с консоли, даже когда стандарный ввод переназначен, следует открыть для ввода файл, обозначаемый специальным именем CON (применив функцию CreateFile) и осуществлять ввод из этого файла.

В операционной системе Unix для ввода текста с консоли, если стандартный ввод не переназначается, можно использовать стандартные функции *gets* и *scanf*. Следует заметить, что использование функции *gets* настойчиво не рекомендуется профессионалами. Дело в том, что она может вернуть текст большей длины, чем предусмотрено программистом в ее вызове. Такая ситуация приводит к катастрофическим последствиям. Единственным эффективным решением указанной проблемы является отказ от применения данной функции. Вместо нее допустимо использовать более общую функцию стандартной библиотеки Си с прототипом

char* fgets(char *s, int size, FILE *stream),

где в качестве последнего параметра можно использовать символическую константу stdin, обозначающую поток стандартного ввода. (В этой функции явно указывается максимальный размер *size* текста, который может быть введен при ее вызове.) В то же время рассмотренная возможность не дает принципиального решения проблемы ввода с консоли, поскольку стандартный ввод может переназначаться в момент вызова программы в Unix.

В Unix для специализированного ввода именно с консоли следует использовать непосредственно устройство консоли как специальный файл. С этой целью нужно открыть доступ к виртуальной консоли, имеющей файловое имя /dev/tty. (В Unix обычно параллельно функционирует ряд виртуальных консолей, обозначаемых именами /dev/tty1, /dev/tty2, /dev/tty3 и т.д., но обозначение /dev/tty задает текущую *управляющую консоль* для исполняемой программы.) Заметим, что в Unix по историческим причинам принято называть консоль словом *терминал*, которое в общем случае обозначает нечто гораздо более общее, в частности, удаленное средство доступа к операционной системе. В данном изложении для единообразия в рассмотрении средств различных ОС будет использоваться термин консоль как приближенное наименование терминала в Unix.

Открытие доступа к терминалу Unix выполняется обычной функцией открытия, например, в виде

hcons=open("/dev/tty",O_RDONLY),

где переменная *hcons* должна быть предварительно описана как целочисленная. После такого открытия можно использовать функцию read(hcons, buffer, *сколько*) для чтения текста непосредственно с клавиатуры (даже если стандартный ввод переназначен).

5.2. Событийно-управляемый ввод

Поведение современных программных систем, активно взаимодействующих с пользователем, определяется не столько последовательностью операторов в исходной программе, сколько внешними воздействиями на компьютер. Такими воздействиями являются нажатия клавиш клавиатуры, перемещение мыши и нажатие ее кнопок, а также более сложные действия, реализуемые обычно также с помощью мыши. Примером такого сложного действия является перетаскивание какого-то графического объекта на экране с помощью мыши (dragging).

Поведение же программ, написанных в классических традициях, определяет сам программист, он при написании таких программ сам решает, что на очередном шаге выполнения будет требовать и ожидать программа: данных от клавиатуры или информации от мыши.

Естественно первый из перечисленных подходов отвечает реальным событиям в реальном мире, порядок которых никому, в том числе программисту, не известен. Поэтому именно он и реализуется в современных программах. Такая реализация потребовала отказа от безоговорочного следования за идеей алгоритма (как последовательности действий предварительно и строго описанной). По существу она оказалась возможной на основе сложного аппаратно-программного механизма, встроенного во все современные компьютеры и называемого механизмом прерываний.

Этот механизм позволяет немедленно реагировать на внешние сигналы путем запуска специально подготовленных программ – обработчиков прерываний, причем выполнявшаяся в момент прерывания программа временно приостанавливается с сохранением всей информации, необходимой для будущего продолжения. После завершения обработки прерываний прерванная программа возобновляет свое выполнение с прерванного места.

Разработка программ обработчиков прерываний очень не простая задача и поэтому ее выполняют разработчики операционных систем. Остальным предоставляется неявно пользоваться этим механизмом. Его использование в прикладных программах основывается на специализированных структурах данных о внешних по отношению к программе событиях.

Само событие, в частности движение мыши, это явление материальной реальности и, чтобы сделать его доступным для программ, внутренние компоненты ОС строят специализированные описания событий, обычно называемые сообщениями

(message) или для выразительности просто событиями (event). Для нас, как программных пользователей операционной системы, существенно, что какой-то ее компонент подготавливает такие специализированные структуры данных для программ, способных их использовать.

Для текстовой консоли Windows, по замыслу разработчиков, теоретически возможны сообщения от нажатия клавиши клавиатуры, от мыши, сообщения об изменении размера окна, о переходе активности к текущему окну или о потере такой активности. Свойство активности визуально отражается изменением цвета заголовка окна и содержательно состоит в том, что только активное окно получает данные от клавиатуры.

В Windows программа для текстового окна может запросить сообщение путем вызова системной функции ReadConsoleInput. Эта функция имеет прототип

```
BOOL  ReadConsoleInput(HANDLE hConsInput,
                       INPUT_RECORD* buffer, DWORD len, DWORD* actlen).
```

Кроме хэндла для указания консольного буфера ввода (в частности хэндла стандартного файла ввода) эта функция содержит адрес буфера, который представляет собой в общем случае массив записей типа INPUT_RECORD для размещения некоторого числа записей сообщений ввода. Размер массива выбирается программистом. Размер этого массива записей задается при вызове в параметре *len*. В простейших случаях массив *buffer* состоит из единственного элемента – для размещения единственного очередного сообщения, и параметр *len* берется поэтому равным 1. В общем случае, когда при вызове функции задается *len*, не равное 1, следует в программе после обращения к ReadConsoleInput проверять, сколько записей о вводе было действительно получено (с помощью параметра *actlen*), и принимать соответствующие действия с учетом этого фактического значения. Заметим, что функция ReadConsoleInput возвращает управление в вызвавшую ее программу только после появления сообщения о вводе. До этого момента вычислительный процесс выполнения программы, содержащей такую функцию, приостановлен (блокирован).

Сообщения как структуры данных типа INPUT_RECORD, получаемые от этой функции, имеют довольно сложное строение. Это строение описывается в заголовочном файле следующим образом:

```
typedef struct _INPUT_RECORD {
   WORD EventType;
   union {
      KEY_EVENT_RECORD KeyEvent;
      MOUSE_EVENT_RECORD MouseEvent;
      WINDOW_BUFFER_SIZE_RECORD WindowBufferSizeEvent;
      MENU_EVENT_RECORD MenuEvent;
      FOCUS_EVENT_RECORD FocusEvent;
```

```
    } Event;
} INPUT_RECORD, *PINPUT_RECORD;
```

Ключевым полем в этой записи является *тип события* EventType, его значения задаются предопределенными константами, описанными как

```
//  EventType flags:
#define KEY_EVENT         0x0001 // Event contains key event record
#define MOUSE_EVENT       0x0002 // Event contains mouse event record
#define WINDOW_BUFFER_SIZE_EVENT 0x0004
                          // Event contains window change event record
#define MENU_EVENT 0x0008 // Event contains menu event record
#define FOCUS_EVENT 0x0010 // event contains focus change
```

Это ключевое поле EventType своим значением определяет более детальное строение сообщения. Если его значение равно KEY_EVENT, то на самом деле в этой универсальной структуре вся остальная часть (обобщенно называемая Event) есть структура типа KEY_EVENT_RECORD. Если же значение типа равно MOUSE_EVENT, то остальная часть есть в действительности структура типа MOUSE_EVENT_RECORD. (Остальные типы сообщений и их структуры в данном изложении рассматриваться не будут.)

Поэтому типовое использование системной функции ReadConsoleInput может быть описано наиболее характерной схемой вида

```
. . .
ReadConsoleInput( hInput, &inpbuf, 1, &actlen);
if (inpbuf. EventType = = KEY_EVENT)
  {обработка события от клавиатуры,
    структура которого представляется в программе обозначением
    ipnbuf.Event.KeyEvent }
if (inpbuf. EventType = = MOUSE_EVENT)
  {обработка события от мыши,
    структура которого представляется в программе обозначением
    ipnbuf.Event.MouseEvent }
. . .    ,
```

в которой предполагается, что использованные информационные объекты данных где-то раньше описаны как

```
HANDLE hInput;
INPUT_RECORD inpbuf;
DWORD actlen;
```

В общем случае рассмотренный фрагмент должен, как правило, находиться внутри цикла, который приводит к многократному запросу сообщений.

Конкретные внутренние поля типов данных для сообщений от клавиатуры и мыши будут рассмотрены несколько дальше.

5.3. Системные функции ввода с клавиатуры

В операционной системе Windows ввод символов даже для консоли организован довольно сложно. Эта организация ввода символов была предварительно рассмотрена в предыдущем разделе. Теперь рассмотрим более детально строение экземпляров структуры типа KEY_EVENT_RECORD. Именно эти структуры данных оказываются в сообщении типа INPUT_RECORD, когда ключевое поле EventType в ней имеет значение KEY_EVENT.

Структура данных KEY_EVENT_RECORD для записи ввода с клавиатуры описана в заголовочном файле как

```
typedef struct _KEY_EVENT_RECORD {
   BOOL bKeyDown;
   WORD wRepeatCount;
   WORD wVirtualKeyCode;
   WORD wVirtualScanCode;
   union {
      WCHAR UnicodeChar;
      CHAR   AsciiChar;
   } uChar;
   DWORD dwControlKeyState;
} KEY_EVENT_RECORD, *PKEY_EVENT_RECORD;
```

В ней содержатся поля как собственно ASCII-кода символа (который обычно и нужен при вводе с клавиатуры), так и дополнительная информация. Поле *bKeyDown* определяет, нажата клавиша (значения TRUE) или отпущена (значение FALSE). Заметим, что для каждого события нажатия или отпускания клавиши формируется свое отдельное сообщение. Поле *wRepeatCount* дает число многократных сигналов от клавиши, формируемых в автоматическом режиме при длительном удержании клавиши (для последовательности таких событий ОС формирует только одно сообщение – для экономии). Поле *wVirtualScanCode* дает сканкод нажатой клавиши, а поле *wVirtualKeyCode* определяет специальный код для управляющих клавиш, принятый в системах Windows. Наконец, поле *dwControlKeyState* комбинацией бит информирует получателя сообщения о текущем – на момент формирования сообщения - состоянии нажатий на специальные клавиши. Коды клавиш для этого поля описаны следующими определениями

```
// ControlKeyState flags
#define RIGHT_ALT_PRESSED   0x0001 // the right alt key is pressed.
#define LEFT_ALT_PRESSED    0x0002 // the left alt key is pressed.
```

```
#define RIGHT_CTRL_PRESSED 0x0004 // the right ctrl key is pressed.
#define LEFT_CTRL_PRESSED       0x0008 // the left ctrl key is pressed.
#define SHIFT_PRESSED          0x0010 // the shift key is pressed.
#define NUMLOCK_ON             0x0020 // the numlock light is on.
#define SCROLLLOCK_ON          0x0040 // the scrolllock light is on.
#define CAPSLOCK_ON            0x0080 // the capslock light is on.
#define ENHANCED_KEY           0x0100 // the key is enhanced.
```

В операционной системе Unix нет специальных средств, отличных от стандарта языка Си, для ввода символов с консоли, а предполагается использовать стандартные библиотеки языка Си. Поэтому для ввода с консоли следует использовать стандартные функции getchar и getch.

Дополнительные возможности в Windows предоставляются путем выборочного задания типов событий, которые будут далее поступать при вызове функции ReadConsoleInput. Этот выбор задается функцией SetConsoleMode с прототипом

```
BOOL SetConsoleMode(HANDLE hConsHandle, DWORD mode);
```

Параметр *mode* задает в ней новый режим управления вводом (или выводом, если в качестве первого параметра *hConsHandle* задан хэндл буфера консольного вывода). Возможные режимы для буфера ввода с консоли задаются константами

```
#define ENABLE_PROCESSED_INPUT    0x0001
#define ENABLE_LINE_INPUT         0x0002
#define ENABLE_ECHO_INPUT         0x0004
#define ENABLE_WINDOW_INPUT       0x0008
#define ENABLE_MOUSE_INPUT        0x0010
```

Если с помощью функции SetConsoleMode задать в качестве режима значение константы ENABLE_MOUSE_INPUT, а другие константы при этом задании режима не использовать, то по запросу событий будут поступать только сообщения от мыши (даже если пользователь и пытается нажимать на клавиши клавиатуры при активном окне текущего приложения). Если же с помощью данной функции установить только режим ENABLE_PROCESSED_INPUT, то будут поступать сообщения лишь от клавиатуры и т.п.

Отключив режим ENABLE_ECHO_INPUT, можно отказаться от использования эха символов, но делать это следует временно – с последующим восстановлением стандартного режима. Поэтому вначале целесообразно запомнить предыдущее значение режима для буфера ввода консоли, для этого действия предназначена функция

```
BOOL  GetConsoleMode(HANDLE hConsHandle, DWORD* pmode),
```

в которой возвращаемое значение текущего режима передается через второй параметр ее вызова.

Как уже пояснялось выше, в Unix для доступа непосредственно к клавиатуре следует выполнить открытие специализированного виртуального устройства /dev/tty. В общем случае универсальная функция read() в варианте использования, читающем по одному символу, может обеспечить посимвольный ввод. Но при этом для программиста возникает проблема, обусловленная тем, что вводимые символы поступают в программу только после нажатия на клавишу Enter, когда используется обычный (стандартный) режим работы консоли. (Информация о нажатии клавиш клавиатуры оказывается недоступен программе до воздействия на клавишу Enter.)

В тех ситуациях, когда программе Unix необходим полноценный посимвольный ввод, следует отменить стандартный режим работы консоли. Для этих целей используются функции с прототипами

int tcgetattr(int htty, struct termios *tsave),

int tcsetattr(int htty, int action, struct termios *tnew),

где параметр htty задает хэндл консоли.

Функция tcgetattr() позволяет получить детальную информацию о текущих режимах консоли, а функция tcsetattr() – установить новые режимы. Получение текущих режимов перед установкой новых настоятельно рекомендуется для восстановления стандартных режимов работы консоли перед выходом из программы (иначе нестандартный режим может существенно нарушить работу последующих программ на данной консоли). Обе функции требуют использования заголовочного файла с именем termios.h. Среди возможных значений параметра *action* функции tcsetattr() чаще всего используется задаваемое символической константой TCSAFLUSH (параметр задает, когда и как будут проведены изменения режима консоли).

Структура типа *termios* содержит множество полей, подробную информацию о которых можно получить в технической документации. С целью настройки режима для одиночного ввода символов с клавиатуры – требуется изменить поля с именами *c_lflag* и *c_cc*. Второе из них представляет собой массив, в элементах которого с индексами, задаваемыми константами VMIN и VTIME, следует установить значения 1 и 0 соответственно. В поле *c_lflag* следует сбросит стандартный режим (обозначаемый константой ICANON) и, как правило, режим автоматического отображения символа (обозначаемый константой ECHO). При этом не следует изменять значения битов, кодирующих другие разновидности режимов. Поэтому последовательность команд, решающая рассматриваемую задачу, имеет вид

```
struct termios sterm, term;
int htty ;
. . .
  htty=open("/dev/tty", O_RDON LY);
  if (htty = = -1) {. . . exit(1);}
```

```
  tcgetattr(htty,&sterm);
  tcgetattr(htty,&term);
  term.c_lflag &= ~(ICANON | ECHO);
  term.c_cc[VMIN]=1;
  term.c_cc[VTIME]=0;
  tcsetattr(htty,TCSAFLUSH, &term);
. . . // использование режима одиночного ввода  символов
. . . // оператором  read(htty, имяодносимвольногомассива, 1);
tcsetattr(htty,TCSAFLUSH, &sterm);
```

В простейшем случае вместо хэндла *htty* виртуального терминала можно использовать хэндл стандартного ввода со значением 0, но только в том случае, когда есть уверенность, что он соответствует именно консоли. (В Unix имеется функция int isatty(int htty), которая позволяет определить, соответствует ли терминалу заданный ее аргументом хэндл, что сообщается возвращаемым значением, отличным от нуля.)

Особо следует остановиться на кодировании управляющих клавиш в терминалах Unix. Если в других рассматривавшихся операционных системах для распознавания таких клавиш предназначен специальный скан-код, то в Unix используются управляющие последовательности кодирования! Управляющие последовательности и для кодирования управляющих клавиш имеют в качестве первых двух символов коды ‘\033’ и ‘[‘. Следующие символы таких последовательностей позволяют различать конкретную управляющую клавишу.

Как правило, клавиши управляющих стрелок имеют трехсимвольные кодирующие последовательности *esc*[A (задает стрелку вверх), *esc*[B (стрелка вниз), *esc*[D (стрелка влево) и *esc*[C (стрелка вправо). Управляющие клавиши от F1 до F5 выдают соответственно четырехсимвольные управляющие последовательности *esc*[[A, *esc*[[B, *esc*[[C, *esc*[[D, *esc*[[E. Управляющие клавиши от F6 до F10 выдают управляющие последовательности *esc*[17~, *esc*[18~, *esc*[19~, *esc*[20~, *esc*[21~, которые состоят из пяти составляющих символов. Управляющие клавиши F11 и F12 выдают аналогичные управляющие последовательности *esc*[23~, *esc*[24~, а клавиши Home, Insert, Delete и End выдают управляющие последовательности *esc*[1~, *esc*[2~, *esc*[3~ и *esc*[4~. (В этих обозначениях буквосочетание *esc* представляет единственный символ с внутренним кодом 27 и записывается в тексте для языка Си как \033.)

Следует заметить, что конкретное кодирование этих последовательностей может зависеть от настройки рабочей версии ОС или от ее текущей модификации. До последнего времени управление терминалами в Unix являлось областью, известной своей несовместимостью между отдельными версиями (клонами) этой операционной системы. К настоящему времени принят стандарт XSI, обеспечивающий

стандартный набор системных вызовов для управления терминалами. Использование этого стандарта обеспечивает совместимость между конкретными версиями Unix, которые его поддерживают.

5.4. Опрос ввода с клавиатуры в программе

В ряде ситуаций при построении программы возникает необходимость получать информацию о том, нажимались ли клавиши клавиатуры во время выполнения данной программы или, иначе говоря, есть ли сообщения от клавиатуры, которые предполагается использовать несколько позже. В архаичной MS DOS для этих целей использовалась библиотека системных функций системно-зависимой версии языка Си, которая содержала функцию *kbhit* (в ориентированном на MS DOS Турбо Паскале аналогичная функция называлась *keypressed*).

В операционной системе OS/2 соответствующая функция (ожидания нажатия на любую клавишу клавиатуры) носила название KbdPeek и имела прототип

```
APIRET16    KbdPeek(KBDKEYINFO  *pkbdi,  HKBD hkbd);
```

причем информация о том нажата ли клавиша, возвращалась в одном из полей структуры данных типа KBDKEYINFO.

В операционных системах Windows проверка нажатия, точнее в более конкретных для этой ОС терминах, проверка наличия сообщений для функции ReadConsoleInput, может проводиться функций PeekConsoleInput, имеющей прототип

```
BOOL PeekConsoleInput(HANDLE hConsInput,
    INPUT_RECORD* buffer,  DWORD len,  DWORD*  actlen);
```

Строение обращения этой функции практически полностью совпадает со строением обращения к функции ReadConsoleInput, и возвращают эти функции практически одно и то же. Различие между ними в том, что функция чтения ReadConsoleInput извлекает (из специальной очереди) очередное сообщение, поэтому следующее по выполнению обращение к этой же функции извлеченное сообщение не найдет (может быть извлечено только следующее сообщение, когда оно появится). Функция же PeekConsoleInput хотя и копирует информацию о событии в свой буфер, но не извлекает ее из очереди. Поэтому повторное выполнение той же функции найдет и извлечет уже использованное значение сообщения. В частном случае, когда сообщения от клавиатуры на момент выполнения функции PeekConsoleInput нет, она возвращает (без ожидания и приостановки процесса выполнения программы) с помощью последнего параметра, что присутствует нулевое число сообщений ввода.

В Windows есть еще одна функция, которую можно использовать для опроса наличия сообщений, но уже без детализации, что это сообщение содержит. Это функция GetNumberOfConsoleInputEvents с прототипом

BOOL GetNumberOfConsoleInputEvents(HANDLE hConsInput, DWORD* number). Она выдает в качестве результата с помощью параметра *number* число сообщений ввода, накопившихся во входном буфере консоли. Если режим ввода выбран так, что к выполняемой программе поступают только сообщения от клавиатуры, то это значение можно использовать как информацию о нажатой клавише.

В операционной системе Unix нет непосредственных аналогов рассмотренных функций опроса нажатия клавиши. Аналогичную содержательную проблему такого специального получения символа из потока данных, чтобы он по-прежнему как бы оставался невостребованным и извлекаемым по следующей операции ввода, разработчики Unix решили очень оригинальным и своеобразным способом. Набор функций посимвольного ввода содержит функцию *ungetc*(int char, FILE *stream), которая возвращает прочитанный символ обратно в буфер ввода. Таким образом рассмотренные функции опроса ввода моделируются в Unix парой последовательных функций *getc*(FILE *stream) и *ungetc*(), между которыми или после которых можно поставить анализ введенного символа. Все же надо отметить, что описанная замена не обеспечивает полной аналогии. В частности, указанные функции относятся к произвольным потокам файловых данных, а не только к консоли.

Иногда, прежде чем опрашивать клавиатуру на предмет нажатия клавиш, нужно отбросить хранящиеся во входном буфере ненужные коды клавиш. Для этих целей служит функция очистки буфера клавиатуры. Она бывает нужна, когда программа начинает выполняться, а во внутреннем буфере, может быть, осталась какая-то символьная информация от клавиатуры, попавшая туда в конце выполнения предыдущей программы.

В операционной системе Windows такая функция имеет прототип

BOOL FlushConsoleInputBuffer(HANDLE hConsInput).

Целесообразно использовать эту функцию в начале программы работы с консолью, чтобы символы введенные до запуска программы в текстовом окне, не повлияли на ее нормальную работу.

5.5. Системные функции мыши для текстового режима

В операционной системе Windows, как уже пояснялось, информация от мыши получается в результате универсального запроса событий для текстовой консоли посредством вызова функции ReadConsoleInput. (В этой ОС мышь является равноправным с клавиатурой устройством, и поэтому сообщения от обоих этих устройств используются чрезвычайно похоже.) При получении указанной функцией сообщения от мыши, поле *EventType* этого сообщения оказывается равным константе MOUSE_EVENT, а комбинированный компонент *Event* является записью типа MOUSE_EVENT_RECORD, который определяется в заголовочном файле описанием

```
typedef struct _MOUSE_EVENT_RECORD {
   COORD dwMousePosition;
   DWORD dwButtonState;
   DWORD dwControlKeyState;
   DWORD dwEventFlags;
} MOUSE_EVENT_RECORD, *PMOUSE_EVENT_RECORD;
```

Компонент *dwMousePosition* этой структуры дает текущую позицию мыши (точнее позицию на момент формирования операционной системой сообщения от мыши) и состоит из X-координаты и Y-координаты курсора мыши. Поле *dwButtonState* выдает код нажатия клавиш мыши и может быть произвольной комбинацией следующих битовых значений:

```
#define FROM_LEFT_1ST_BUTTON_PRESSED  0x0001
#define RIGHTMOST_BUTTON_PRESSED      0x0002
#define FROM_LEFT_2ND_BUTTON_PRESSED  0x0004
#define FROM_LEFT_3RD_BUTTON_PRESSED  0x0008
#define FROM_LEFT_4TH_BUTTON_PRESSED  0x0010
```

Поле *dwEventFlags* позволяет определить двойное нажатие мыши и двигалась ли мышь в момент формирования сообщения. Для представления этой информации отдельными битами в данном поле служат следующие символические константы

```
#define MOUSE_MOVED       0x0001
#define DOUBLE_CLICK      0x0002
```

Поле *dwControlKeyState* совпадает по названию и назначению с соответствующим полем в записи события от клавиатуры и содержит комбинацию битов, отражающих текущее состояние отдельных управляющих клавиш.

В Unix мышь появилась, когда в этой операционной системе уже сложились определенные традиции, и поэтому является в ней достаточно новым компонентом. Следствием является отсутствие единого установившегося стандарта на использование мыши внутри центральных средств Unix.

Поэтому рассмотрим сложившуюся подсистему управления и использования мыши в широко распространенном варианте этой операционной системы, называющимся Linux и имеющим лицензию на свободное некоммерческое распространение. При этом дальнейшее изложение в данном разделе не претендует на универсальность для всех вариаций операционных систем Unix.

Для использования мыши в Linux служит программная подсистема *gpm*. Построена она по технологии клиент-сервер. Основой подсистемы служит *сервер мыши*. В некоторой степени это – аналог драйвера мыши в других системах, но здесь последовательно проводится принцип независимости обслуживающей программы как совершенно отдельного процесса, предназначенного для выполнения

вычислительных услуг, которые специальными сообщениями запрашивают другие процессы.

Перед началом использования мыши процесс должен зарегистрироваться у сервера и передать ему информацию, какие сообщения от мыши интересуют этот процесс. Получаемые сообщения от мыши процесс-клиент может обрабатывать как полностью сам, так и передавать другой процедуре обработчика сообщений от мыши.

Для регистрации у сервера программа процесса должна заполнить структуру Gpm_Connect и выполнить вызов функции Gpm_Open с этой структурой в качестве аргумента. Функция Gpm_Open имеет прототип

```
int Gpm_Open (Gpm_Connect *CONN, int flag),
```

где флаг flag в простейших случаях должен быть задан равным 0. Структура Gpm_Connect описана в заголовочном файле как

```
typedef struct Gpm_Connect {
  unsigned short eventMask, defaultMask;
  unsigned short minMod, maxMod;
  int pid;
  int vc;
} Gpm_Connect;
```

Перед регистрацией необходимо заполнить поля *eventMask*, *defaultMask*, *minMod*, *maxMod*. Остальные поля заполняются автоматически. В простейшем случае для полей *minMod*, *maxMod* берется значение 0. Ненулевые значения этих полей используются только, если клиентом предполагается отдельно обрабатывать комбинации нажатия клавиш мыши в сочетании с нажатыми клавишами Shift, Ctrl и Alt. Особенно важно значение поля *eventMask*, именно оно определяет, какие сообщения будет получать программа клиента. Значение этого поля задается как логическая (побитовая) комбинация символических констант, которые задают типы событий. Основные типы событий обозначаются следующими константами: GPM_MOVE – событие движения мыши при не нажатых ее клавишах; GPM_DRAG – событие *протаскивания*, когда нажаты одна или более клавиш и мышь движется; GPM_DOWN – событие нажатия клавиши мыши; GPM_UP – событие отпускания клавиши мыши; GPM_DOUBLE – используется для уточнения события типа нажатия, отпускания и протаскивания и обозначает наличие быстрого двойного нажатия на клавишу мыши.

Для того чтобы поручить серверу передавать клиенту сообщения о движении мыши, следует включить GPM_MOVE в значение поля *eventMask* при вызове клиентом функции Gpm_Open регистрации у сервера. Если же клиента интересуют только нажатия на клавиши мыши, то следует в eventMask включить только GPM_DOWN. Когда предполагается использовать двойные нажатия клавиш

мыши, следует занести в *eventMask* комбинацию GPM_DOWN | GPM_DOUBLE и т.д.

Поле *defaultMask* используется для указания какие сообщения, формируемые мышью на активной консоли клиента, следует передавать на обработку по умолчанию (стандартную обработку) серверу. Обычно сообщения о движении мыши (код типа события GPM_MOVE) передаются (оставляются) серверу, чтобы именно он занимался рисованием курсора мыши на экране. Если этого не сделать, то либо курсор мыши не будет при движении виден на экране (никто его при этом не рисует), либо в программе клиента следует предусмотреть программные фрагменты рисования такого курсора.

Стандартными рекомендациями по использованию поля *defaultMask* являются следующие. Вариант *~eventMask* поручает выполнять стандартную обработку всех событий, которые не обрабатывает клиент. Вариант 0 обозначает отбрасывание всей стандартной обработки событий.

Функция Gpm_Open при неудаче возвращает значение -1, а при удаче – хэндл для обращения к серверу, который, впрочем, явно почти не используется, так как копия его значения сохраняется в глобальной для клиента переменной *gpm_fd*.

Следует обратить особое внимание, что программу, использующую сервер мыши gpm не удается нормально запустить из-под инструментальной оболочки *mc* (полное имя которой MidnightCommander). При попытке такого запуска возвращается код возврата, свидетельствующий о неудаче. Такие программы следует запускать из командной оболочки, не использующей мышь, в частности, выйдя из инструментальной оболочки *mc* по нажатию управляющей клавиши F10. Если же присутствует необходимость одновременного использования и инструментальной оболочки *mc*, и прикладной программы, запрашивающей обслуживание gpm-сервером мыши, то следует вызывать функцию открытия доступа к такому серверу с последним аргментом *flag*, равным числу -1.

После удачной регистрации программа клиента может запрашивать и получать сообщения от мыши, для этого служит функция Gpm_GetEvent с прототипом

```
int Gpm_GetEvent (Gpm_Event *EVENT)
```

Неявно эта функция использует глобальную переменную *gpm_fd* со значением, установленном при регистрации клиента. Функция возвращает значение 1 при успешном выполнении, -1 – при неудаче и 0 – после закрытия связи с сервером. Единственным аргументом функции является структура данных сообщения Gpm_Event, описываемая как

```
typedef struct Gpm_Event {
  unsigned char buttons, modifiers;
  unsigned short vc;
  short dx, dy, x, y;
  enum Gpm_Etype type;
```

```
  int clicks;
  enum Gpm_Margin margin;
}           Gpm_Event;
```

Поля *x*, *y* этой структуры дают координаты позиции курсора мыши на момент формирования сообщения. Поле *buttons* выдает код нажатия клавиш и является логической комбинацией следующих констант:

```
#define GPM_B_LEFT      4     // Левая клавиша мыши
#define GPM_B_MIDDLE    2     // Средняя клавиша мыши
#define GPM_B_RIGHT     1     // Правая клавиша мыши
```

Такая кодировка заметно отличается от используемой в других из рассматриваемых ОС. Поля *dx*, *dy* дают значения приращения координат мыши в сравнении с предыдущим сообщением от нее. Поле *type* информирует о типе полученного сообщения и может принимать одно из следующих значений: GPM_MOVE, GPM_DRAG, GPM_DOWN, GPM_UP. Значение поля *clicks* дает число нажатий клавиши, зафиксированных в одном событии, и может быть 0, 1, 2 для одиночного двойного и тройного нажатия соответственно. Это поле имеет действенное значение только для событий нажатия, отпускания и протаскивания. Назначения остальных полей в данном изложении рассматриваться не будут, так как реальные возможности современного системного обеспечения намного больше, чем удается изложить в относительно небольшом по объему учебном пособии.

После завершения использования мыши клиентом следует вызвать функцию Gpm_Close(), которая разрывает связь клиента с сервером. Функция эта не имеет аргументов. Заголовочным файлом для программ с использованием подсистемы gpm служит файл gpm.h.

При разработке программ, использующих подсистему gpm, следует иметь в виду необходимость явного подключения библиотеки поддержки этой подсистемы. Такое подключение может задаваться в командной строке дополнительным параметром полного имени библиотеки. Обычно ее полное имя /usr/lib/libgpm.so, так что вызов компиляции исходного текста программы prim1.c будет иметь вид

```
gcc -o prim1.exe  prim1.c  /usr/lib/libgpm.so
```

Другим, более компактным вариантом, является использование вспомогательной опции задания дополнительной библиотеки. Эта опция в общем случае имеет вид

-luникальнаячастьименибиблиотеки,

где *уникальнаячастьименибиблиотеки* получается отбрасыванием префиксной части lib в имени библиотеки. При использовании указанной опции наш пример перепишется в виде

```
gcc   -o prim1.exe   prim1.c    -lgpm
```

Следующий пример, приведенный в листинге 5.5.1, демонстрирует использование рассмотренных функций подсистемы gpm.

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

int main()
{int k, xm, ym, button;
 struct Gpm_Event evnt;
 struct Gpm_Connect conn;

   conn.eventMask = GPM_DOWN; // запрашиваем события нажатия кнопок
мыши
   conn.defaultMask = GPM_MOVE;
   conn.minMod = 0;
   conn.maxMod = 0;
   if (Gpm_Open(&conn,0) = = -1)
     {printf("Error Open Mouse");exit(1);}
   do
    {Gpm_GetEvent(&evnt);
     xm=evnt.x; ym=evnt.y;
     button=evnt.buttons;
     printf("\033[1;60H\033[1;33m"); // позиция на x=60,y=1; цвет YELLOW
     printf("row=%3d col=%3d key=%1d\n", xm, ym, button);
    } while (button!=5); // при GPM_B_LEFT и GPM_B_RIGHT одновременно
   printf("\033[0m");
   Gpm_Close();
   return 0;
}
```

Листинг 5.5.1. Использование мыши в Linux

В программе выполняется регистрация на gpm-сервере с запросом получения всех сообщений от мыши и стандартной обработкой сообщений о движении мыши сервером. После этого запускается цикл опроса событий от мыши. Данные, полученные от сервера, о координатах курсора мыши и кодах нажатия клавиш отображаются в фиксированных позициях экрана желтым цветом, а именно, начиная с 60-го столбца 1-й строки экрана. Выход из цикла происходит по получении сообщения о нажатии одновременно левой и правой клавиш. После чего выполняется восстановление стандартного цвета вывода, закрытие связи с сервером и завершение программы.

Если программист запрашивает получение событий типа GPM_MOVE, то обработчик событий от мыши по умолчанию не занимается прорисовкой изображе-

ния курсора мыши. Поэтому для принудительной прорисовки этого курсора предусмотрены специальные средства. Они состоят из функции Gpm_DrawPointer и макроса GPM_DRAWPOINTER. Указанная функция имеет прототип

Gpm_DrawPointer(short x, short y, int gpm_consolefd),

а макрос требует единственного параметра – адреса экземпляра события, полученного от сервера мыши функцией Gpm_GetEvent (или аналогичной ей).

Упражнения

1. Разработать программу, которая запускает использование мыши и отображает в верхнем правом углу значения текущих координат указателя мыши, завершая работу при одновременном нажатии на обе клавиши мыши (первую и вторую клавиши). Следует разработать два варианта: один для Windows, другой – для Linux.

2. Разработать программу, которая выводит на экран текст из некоторого файла, инициализирует для программы мышь, и при нажатии на клавиши мыши во время нахождения ее указателя на любом изображаемом экраном слове отображает это слово в нижней строке экрана. Причем если клавиша мыши нажата во время движения, то отображение в нижней строке должно выполняться красным цветом, а если при неподвижной мыши – зеленым. Следует разработать два варианта: один для Windows, другой – для Linux.

3. Разработать для Windows подпрограмму реализации функции kbhit или keypass из стандартных библиотек языков Си и Паскаль, выполняя эту реализацию с помощью системной функции PeekConsoleInput.

6. ФАЙЛОВЫЕ СИСТЕМЫ

6.1. Структуры файловых систем для пользователя

Практической основой любой ОС служит ее базовая файловая система. Так основой старенькой MS DOS являлась существующая до сих пор файловая система FAT в двух модификациях: FAT16 и FAT12 (соответственно для жестких и гибких дисков). Основой Windows NT служит файловая система NTFS (NT File System), операционная система OS/2 основана на файловой системе HPFS (High Performance File System). ОС MS DOS даже основную часть своего названия получила от слова Disk (Disk Operation System). Более того, наличие собственной файловой системы в составе ОС непосредственно определяет само существование ОС, так как более простые программные комплексы, используемые в настоящее время только для микропроцессорных систем и содержащие только средства управления устройствами и интерфейс с пользователем, относятся к так называемым мониторам и управляющим программам для микропроцессоров. (Заметим, что последняя

из четырех основных функций, определяющих ОС – обеспечение параллельного использования оборудования и ресурсов – относится только к многопрограммных ОС.)

Файловая система – это совокупность всех информационных средств, обеспечивающих использование файлов, причем сюда традиционно не относят драйверы устройств и их менеджеры. В значительной степени файловая система определяется именно структурами длительного хранения информации в операционной системе, причем структурами, независимыми от конкретных устройств.

Напомним, что файлом называют именованную совокупность информации, хранимую в вычислительной системе (компьютере или вычислительной сети). Причем согласно установившейся традиции, восходящей к первым версиям Unix, эта совокупность информации рассматривается как бесструктурная последовательность байтов. Несколько более абстрактно файл можно представлять как массив данных переменной длины, к которому осуществляется автоматический последовательный доступ.

Хранение именованных совокупностей информации имеет для человечества давнюю историю. В докомпьютерную эпоху основными из таких совокупностей были книги, статьи, рассказы, свитки и папки документов. Уже в этих технических решениях, ориентированных на более совершенную, чем компьютер, систему – человека – были созданы и отработаны принципиальные средства, обеспечивающие доступ при большом числе таких совокупностей.

Эти решения заключались в использовании каталогов и оглавлений, где название именованной совокупности соотносилось с информацией о месте хранения ее. Каталоги библиотечных фондов содержат кроме подробной информации о наименовании объекта и времени его создания еще и чрезвычайно важную информацию, кодирующую место хранения объекта во внутренних хранилищах. Оглавление книги, кроме названия объекта (например, рассказа или стихотворения), содержит номер страницы, с которой начинается размещение этого объекта. Практически именно эти базовые решения и были использованы при внутреннем построении файловых систем. (Заметим, что последовательный просмотр всех единиц хранения также неэффективен в компьютерных системах, как попытка найти нужный рассказ известного автора в хранилище, где все книги без какого-то порядка расставлены на множестве полок.)

Практически любая файловая система имеет две существенно различных стороны построения (два "лица"): для внешнего пользователя и для программ-исполнителей. (Аналогично тому, как работа в библиотеке имеет две стороны: правила запросов на нужный материал (как заполнять карточку заказа) и как библиотечным работникам находить в хранилище нужную книгу или иную единицу хранения.)

Внешняя сторона файловой системы в основном сводится к форме именования файлов в программах. Каждый файл имеет собственное имя, подобно тому, как собственное имя имеет человек, но как и для человека оно может не быть уникальным. Поэтому потребовалось использовать так называемое *полное имя файла*, которое однозначно обозначает файл в вычислительной системе. Ограничимся пока полным именем в границах отдельного компьютера (об обозначении файлов уникальными обозначениями в пределах сети речь будет идти при рассмотрении каналов передачи данных). Полное имя по своему смыслу аналогично именованию человека, в котором для однозначного толкования приводится информация о местонахождении. Этот прием использовался человечеством с глубокой древности. Так часть имени известного средневекового математика было использовано для обозначения алгоритма (Abu Ja'far Mahammed ibn Musa al-Khowarizmi, т.е. из Хорезма).

В простейших файловых системах допускался только один уровень каталогов, но в современных системах, начиная с Unix, стали использовать многоуровневые, образующие древовидные структуры взаимных связей.

В первом приближении файловые системы с внешней стороны подразделяются на использующие и на не использующие обозначения логических дисков (в более ранних ОС логическим дискам соответствовало, хотя и не совсем точно, понятие *имени тома*). Файловые системы типа FAT, NTFS и HPFS используют логические диски, а файловые системы, непосредственно предназначенные для Unix, никогда логических дисков не используют. Применение логических дисков (или ранее имен томов дисков) заметно упрощает для непрофессионального пользователя обозначения файлов: можно указать нахождение файла на логическом диске, где, как он знает, этот файл "лежит". Такой прием показался очень удобным для предшественника операционной системы MS DOS и ее первых версий, предназначавшихся для компьютеров, в которых из дисковой памяти присутствовали только накопители на гибких дисках в количестве нескольких штук. (При этом указывалось, на каком накопителе находится желаемый программистом файл.) До сих пор отзвук этого положения сохранился в ситуации с флоппи-дисками, которые почти всегда теперь именуют буквой A.

Более общий подход использует ОС Unix, где все возможные файлы собраны в одно единственное дерево связями типа "описано в каталоге". Здесь файловое дерево включает три разновидности объектов: обычные файлы, каталоги и специальные файлы. Специальные файлы соответствуют устройствам компьютера: реальным или виртуальным. Дерево внешней структуры файловой системы Unix имеет единственный *корень*, обозначаемый символом / (прямая разделительная черта – slash). Этот корень является корневым каталогом, перечисляющим каталоги следующего уровня. Любой каталог, отличный от корневого, может содержать любое число обычных файлов и каталогов следующего уровня. С учетом свойств

древовидной структуры математического графа (парных связей между объектами) любой узел этой структуры можно однозначно задать последовательностью каталогов, через которые идет путь от корневого узла к данному узлу, и собственным обозначением последнего узла в последнем каталоге. При этом допустимыми являются одинаковые обозначения узлов дерева, если они не связаны непосредственно с одним и тем же каталогом. Таким образом, требование именования узлов в дереве, а следовательно, и в древовидной структуре файловой системы заключается в уникальности имени в пределах одного каталога. (Формальное описание внешней структуры файловой системы в виде дерева заключается в том, что любой обычный или специальный файл должен быть концевой вершиной - листом, а промежуточные вершины дерева, не являющиеся концевыми, – каталогами.)

В файловой системе с логическими дисками внешняя структура файловой системы описывается несколькими деревьями (называемыми в теории также *кустами*) – по одному на каждый логический диск. Различие таких составляющих деревьев и, как следствие, отдельного файла обеспечивается внесением в систему обозначений еще и наименования логического диска. В современных ОС, использующих логические диски, принято обозначать последние одной латинской буквой, за которой в контексте обозначений следует служебный символ двоеточия. Последний прием дает удобное и наглядное средство обозначать логические диски в составе полного наименования файла.

Последовательность каталогов, задающих путь от корневого узла к данному файлу, так и называется *путь* (path). В файловых системах FAT, NTFS и HPFS разделителем в такой последовательности каталогов служит символ \ (обратная разделительная черта – backslash). В файловых системах Unix символом разделения в обозначении последовательности каталогов является прямая разделительная черта (символ /). Заметим, что последний вариант на самом деле более практичен, так как обратная разделительная черта является специальным символом в языке Си, и задание этого символа в своем родном значении требует его двойного указания. Поэтому полное имя файла file, находящегося в каталоге с именем user корневого каталога диска C, будет на языке Си обозначаться текстом "C:\\user\\file", а аналогичное полное имя файла file, находящегося в каталоге с именем usr корневого каталога файловой системы Unix будет на языке Си обозначаться текстом "/usr/file". (Как говорится, мелочь, а приятно.)

Кроме внутренней информации, хранимой собственно в файле, любой файл может иметь атрибуты доступа. В простейших системах типа FAT и преемственных к ним HPFS и NTFS для обслуживания таких атрибутов служит команда ATTRIB.

Аргументом команды может быть имя отдельного файла или задание совокупности файлов через метасимволы. В простейшей форме – без опций – эта ко-

манда выводит атрибуты, обозначаемые символами S, H, R, A (System, Hidden, ReadOnly, Archive). Кроме того, в формах

ATTRIB -*атрибут файл*

ATTRIB +*атрибут файл*

эту команду можно использовать для сброса соответствующего атрибута (операция -) или установки его (операция +). При этом атрибут задается одним символом, и в одной команде может быть задано несколько операций сброса или установки атрибутов. Аналогичные действия можно выполнять с помощью утилиты *Проводник* (EXPLORER) графической оболочки Windows или других визуальных средств просмотра в этой оболочке, которые допускают вызов диалоговой панели *Свойства* из контекстного меню. Действия над атрибутами в этой панели очевидны по их оформлению. В инструментальной системе FAR быстрым средством вызова диалогового окна получения и изменения атрибутов файла служит комбинация клавиш Ctrl-A (с участием клавиши латинской буквы A).

В Unix системах вместо атрибутов используются характеристики доступа для трех категорий пользователей: самого владельца (или создателя), обозначаемого в командах символом *u*, для членов его группы (обозначаемых символом *g*) и всех остальных, обозначаемых символом *o* (others). Для каждой категории может быть задано независимо от других три типа доступа: по чтению (символ r), по записи (символ w) и по выполнению (символ x). Выдача информации о характеристиках доступа запрашивается командой *ls* с ключом *-l* в виде

ls -l имяфайла

Изменение характеристик доступа файла разрешается только его владельцу или администратору и выполняется командой

chmod *операция имяфайла*

В качестве операции используется запись вида *пользователь+типдоступа*, где оба операнда задаются в простейшем случае одним символом из указанных выше. Например, установка доступа "по выполнению для всех" задается командой

chmod o+x *имяфайла*

Операционная система Windows NT обладает мощными средствами аналогичной установки ограничений доступа. Информация об этих средствах может быть получена из ее интерактивной справочной системы.

6.2. Методы распределения внешней памяти

Основная проблема при построении файловой системы "изнутри" заключается в соотнесении имени файла физическим единицам хранения на носителе информации. В настоящее время данные на магнитных носителях хранятся в виде совокупностей по 512 байтов. (Иногда используются единицы хранения в 1024 или даже 2048 байтов, но по ряду причин они не стали пока употребительными.) Такая

совокупность байтов непосредственно на внешнем носителе: дискете или жестком диске называется *сектором* (сектором магнитного диска). Сектор магнитного диска по смыслу использования аналогичен странице бумаги для печатного текста. Почти также, как в делопроизводстве оказывается неудобным пользоваться листами произвольного размера, в технических системах информации, несмотря на ряд теперь уже забытых попыток, оказалось удобней всего пользоваться подобными стандартными блоками для хранения данных. (Вспомните, даже маленькие стихотворения для постоянного хранения размещают по одному на целой странице бумаги, хотя большая часть такой страницы не используется при этом.)

Сектора магнитного диска в пределах логического диска (или даже всего магнитного носителя) могут быть пронумерованы последовательностью целых чисел. Эту нумерацию могут брать на себя программные компоненты системы ввода-вывода или даже аппаратура современных электронных устройств. Для наших целей пока достаточно, что для компонентов файловой системы вся совокупность секторов диска может однозначно рассматриваться как массив со сквозной нумерацией.

Основной внутренней функцией каталогов в файловых системах оказывается связывание имени файла с информацией о местонахождении файла на магнитном носителе (совершенно также, как для каталога в библиотеке на бумажных носителях). Таким образом внутреннюю структуру элемента каталога файловых сисем (ФС) можно образно изобразить в виде

Имя файла -> Информация о месте хранения файла.

В этом же элементе может находиться служебная информации о файле (дата его создания, размер и т.п.), хотя такое решение совсем не обязательно.

Основной технической проблемой хранения файла в техническом носителе является характерное свойство большинства файлов менять свой размер в процессе использования. (Чаще всего файлы пополняются, но могут и уменьшаться в размерах.) Такой эффект не характерен для бумажных носителей информации. Простейшее решение, используемое человеком при хранении увеличивающейся последовательности томов (многотомного издания, печатаемого отдельными томами), заключается в образовании необходимого свободного места для хранения путем раздвижения соседних единиц хранения. Практическая реализация такого подхода для хранения файла породила в свое время метод *последовательного непрерывного размещения*.

В этом методе для файла выдвигается требование его размещения в непрерывной последовательности пронумерованных секторов. Этим облегчается составление информации о местонахождении файла на магнитном диске: такая информация будет всегда включать номер начального сектора в последовательности секторов хранения и число таких секторов для данного файла (теоретически можно было бы хранить и номер последнего сектора в такой последовательности). Подобное решение предельно просто и применялось в ОС "реального времени" в

80-х годах XX в., называвшейся RT-11. К очевидным недостаткам этого метода относится невозможность увеличить файл на старом месте его хранения, если за ним размещен другой файл. В таком случае содержимое файла с увеличенным размером приходилось размещать на новом месте (если оно имелось). При периодическом пополнении файла его содержимое перемещалось с места на место. Практическое использование указанного метода требовало периодической реорганизации размещения файлов, которое приводило к "выдавливанию" неиспользуемых участков размещения файлов и размещению используемых строго один за другим. Современным аналогом такого действия, решающего хотя и несколько иные задачи, является запуск служебной программы дефрагментации в файловых системах Windows 9x.

При достаточно длительной эксплуатации файловой системы в области хранения данных оказывается много незаполненных промежутков, возникающих как результат удаления файлов (аналог ситуации выбрасывания части книг из книжного шкафа). Поэтому оказывается практически целесообразным размещать данные новых файлов в таких незаполненных промежутках (также как в освободившихся местах книжного шкафа). При этом достаточно большие новые файлы могли не поместиться ни в один из незаполненных промежутков, и у разработчиков возникало естественное желание размещать новые файлы по частям в свободных промежутках. Реализация такого желания влечет отказ от непременного размещения файла в виде непрерывной последовательности секторов.

Наиболее радикальным решением при этом стало встраивание в ФС таких информационных средств, которые позволяют хранить содержимое файла в множестве секторов, произвольно разбросанных по области хранения. Встает вопрос, где же файловой системе хранить информацию о практически произвольном размещении содержимого файла по секторам области хранения? Теоретически возможны два принципиальных решения: хранить эту информацию подобно структуре связного списка в самих секторах области хранения или использовать специальные структуры данных, предназначенные исключительно для информации о размещении файлов. Первое решение называется методом *последовательно-связного размещения* файла. Оно имеет максимальную гибкость, восходящую к достоинствам абстрактной структуры связного списка, но, к сожалению, и обладает основным недостатком последнего: практической невозможностью произвольного доступа к промежуточным блокам (доступ к элементу связного списка возможен только по цепочке предшествующих элементов). Последнее лишает файловую систему возможности использовать прямой доступ к произвольному элементу файла и, как следствие, настолько снижает ее производительность, что делает такой метод практически нецелесообразным.

Таким образом, возникла практическая неизбежность конструктивного включения в файловую систему специализированных структур хранения информации о

размещении содержимого файлов в блоках хранения (в частности, в секторах области хранения данных).

Возможны два варианта построения структур размещения файлов:

- произвольное размещение файлов, допускающее какой угодно разброс блоков файла по исходной совокупности пронумерованных блоков дисковой памяти;
- размещение файлов, использующее участки возможно максимальной длины из последовательно пронумерованных блоков.

Такие участки последовательно пронумерованных блоков дисковой памяти принято называть *экстентами* (от слова extent).

Произвольное размещение файлов требует наличия таблиц такого размещения. При этом, в свою очередь, возможны следующие варианты:

- единственная таблица размещения на файл;
- таблицы размещения с несколькими уровнями;
- сцепленные таблицы размещения.

Все эти варианты в непосредственном применении требуют построения таблиц, число которых зависит от текущего числа файлов в системе, и поэтому порождают существенно нерегулярные структуры.

Сцепленность таблиц размещения предполагает, что все такие таблицы помещаются одна за другой, в конце каждой таблицы специальным кодом указывается ее конец, а в записи оглавления для файла помещается указатель на начало такой таблицы, соответствующей этому файлу. Сами таблицы размещаются как связный список. Существенной вариацией идеи сцепленных таблиц являются таблицы размещения дисковой памяти, рассматриваемые далее и свободные от отмеченной выше нерегулярности.

Заметим, что все таблицы размещения файлов при работе с файловой системой должны находиться в оперативной памяти для обеспечения достаточно быстрой работы с файлами.

Прежде чем идти дальше отметим, что для нормальной работы файловой системы необходимо иметь не только структуры размещения файлов, но и информацию о свободных на текущий момент блоках возможного размещения. Действительно, информация о размещении блоков файла необходима для доступа к его содержимому, но создание и увеличение уже имеющегося файла невозможно без информации, где размещать его новые данные.

В связи с этим выделяют *методы распределения файлов*, под которыми понимают распределение новых блоков файлу из свободных на текущий момент. Такие распределения требуют ведения детальной информации о свободных блоках области данных файловой системы. Известны четыре метода распределения файлов, называемые по структуре детальной информации о свободных блоках:

- *списком свободных блоков*;

- *картой свободных участков*;
- *битовой картой*;
- *таблицей распределения дисковой памяти*.

Список свободных блоков использует структуру связного списка, который соединяет в одну цепочку все свободные блоки (на томе дисковой памяти или логическом диске). Недостатком этого метода являются большие затраты времени на создание и расширение файла, когда ему дополнительно выделяется не один, а достаточно много новых блоков. Удаление файлов при использовании этого метода также требует значительных затрат времени от ОС, потому что файловая система в этом случае должна все блоки удаляемого файла вставить в цепочку свободных блоков (выполняя столько операций над дисками, сколько блоков в удаляемом файле).

Наиболее изящным методом из перечисленных является метод битовой карты. Битовая карта представляет собой вспомогательную таблицу, хранимую на магнитном диске, в которой k-му биту таблицы соответствует k-й блок области данных файловой системы. Одно значение этого бита (обычно единичное) отражает занятость соответствующего блока каким-то файлом, другое значение (обычно нулевое) отражает, что соответствующий блок свободен. Если в конкретной файловой системе имеется к примеру, 4 млн. распределяемых блоков хранения по 512 байтов (секторов) с общей возможностью хранения около 2 Гбайт данных, то битовая таблица будет состоять из 4 млн. битов или, иначе говоря, из 512-килобайтной битовой карты. Такой относительно небольшой объем этой карты позволяет при работе компьютера держать ее постоянно в оперативной памяти и достаточно быстро определять занятость блока дисковой памяти с любым номером. Заметим, что битовая карта дает информацию только о свободных блоках (и как несущественное следствие, о занятых), но не несет никакой информации о размещении файлов по конкретным блокам.

Метод свободных участков предполагает для своего использования, что файлы размещаются не произвольными наборами блоков, а небольшим числом участков с непрерывной нумерацией в каждом из них. Отказ от этого требования приводит к тому, что в определенных неблагоприятных сочетаниях условий (которые могут сложиться в реальной работе операционной системы), свободные участки выродятся в отдельные блоки. Например, может получиться, что каждый нечетный блок распределен некоторому файлу, а каждый четный свободен. В таком неблагоприятном случае для хранения данных о свободных участках потребуется больше места, чем требуется для битовой карты.

Таблица распределения дисковой памяти является структурой, объединяющей в себе свойства и битовой карты и таблицы размещения отдельных файлов. Ее английское наименование File Allocation Table дало обозначение ряду файловых си-

стем (FAT), получивших широкое распространение в разработках фирм IBM и Microsoft, которые рассмотрим далее.

6.3. Принципы построения файловых систем типа FAT

Основной внутренней структурой файловых систем типа FAT является таблица размещения файлов. Ведущей идеей ее создания послужило замена отдельного бита в битовой карте на специальный более сложный элемент (элемент таблицы FAT), который кроме кодирования состояния свободного блока, позволяет задавать очередной элемент таблицы, описывающей дальнейшее размещение блоков для конкретного файла. Таким образом, таблица FAT объединяет в своих элементах как функции битовой карты, так и таблицы размещения отдельных файлов.

Единицей выделения дисковой памяти в рассматриваемых системах является не отдельный сектор, а так называемый *кластер*. Кластером в этом контексте называют последовательную группу из фиксированного для конкретной FAT числа секторов. По техническим причинам удобства внутреннего кодирования эта группа должна состоять из 2^n секторов. Поэтому размер кластера может иметь величину в 1, 2, 4, 8, 16, 32 или 64 секторов. При этом реальный объем кластера составляет от 512 байтов до 32 Кбайтов. В системе FAT файлу можно выделить только целое число кластеров, поэтому минимальный участок дисковой памяти, расходуемый файлом есть один кластер. Если для конкретной FAT кластера имеет размер в 4 сектора, то второй кластер образуют секторы с номерами 8, 9, 10 и 11, а 315-й кластер, например, образуют секторы с номерами 1260, 1261, 1262 и 1263. Пересчет номеров кластеров в номера соответствующих секторов выполняют компоненты файловой системы. Число элементов в таблице FAT равно числу кластеров, которые имеются в области данных данной файловой системы. Каждому элементу FAT однозначно соответствует кластер в области хранения данных.

Оглавление файловой системы FAT строится таким образом, что в его элементе информация для связи с местом хранения файла состоит всего лишь из номера первого кластера для того файла, который именует данный элемент оглавления. Собственно таблица FAT в своих элементах может хранить:

- код свободного кластера (обычно это значение 0);
- условное значение для сбойного кластера;
- номер следующего кластера размещения;
- условное значение, обозначающее конец размещения файла (обычно все единицы двоичного кода).

В качестве примера рассмотрим два файла с именами *file1.txt* и *file2.txt*, которым соответствуют следующие элементы каталога:

| file1.txt | 7 |
| file2.txt | 9 |

и участок таблицы FAT, начинающийся с 7-й его ячейки:

Номер элемента таблицы: . . . | 7 | 8 | 9 | 10 | 11 | 12 | . . .

Элементы таблицы FAT: . . . | 8 | 10 | 2 |EOF| 0 | EOF | . . .

Данная таблица FAT описывает 7-м своим элементом, что продолжение файла находится в 8-м кластере; 8-м элементом показывает, что продолжение файла находится в кластере 10; 9-м элементом задает, что файл продолжается в кластере 12; а элемент 10 указывает, что в кластере 10 файл заканчивается. (Предполагается, что буквосочетание EOF обозначает код конца файла в таблице FAT.) Таким образом, приведенная информация задает размещение файла *file1.txt* в кластерах 7, 8 и 10, а файла *file2.txt* – в кластерах 9 и 12. Тот же участок таблицы FAT показывает, что кластер 11 свободен. Заметим, что сами номера элементов в таблице не хранятся, а определяются по порядку размещения элементов в этой таблице.

Таблицы FAT позволяют размещать файл в произвольной совокупности кластеров. Так, в приведенном примере последовательность кластеров размещения файла *file1.txt* временно прерывается для начала размещения файла *file2.txt*. В результате отдельный файл может размещаться в любой прерывистой последовательности кластеров, разбрасываясь своим содержимым по значительной части дискового пространства. Это хорошо тем, что позволяет просто использовать все дисковое пространство данных и всегда легко находить место для расширения файла. С точки же быстродействия, такая особенность системы FAT имеет отрицательный эффект: файл размещенный в несмежных кластерах будет читаться и перезаписываться значительно дольше, чем размещенный в смежных. Поэтому файловые системы FAT предназначены для операционных систем невысокой производительности: для однопрограммных и однопользовательских ОС.

Тем не менее из-за своей простоты файловые системы FAT получили широчайшее распространение в персональных компьютерах конца XX века. При этом использовались две модификации этой системы, называемые FAT12 и FAT16, различаемые по числу двоичных разрядов в элементе таблицы FAT. Вариант FAT12 использовался и широко используется до сих пор для размещения файлов на дискетах. Сложившийся объем памяти таких дискет составляет порядка 1,44 Мбайт (появлялись модификации дискет с объемом около 2 Мбайт, но они не прижились). При размере сектора 512 байтов на этих дискетах можно разместить до 2880 секторов, это число перекрывается возможными значениями 12-битного двоичного кода для нумерации секторов (до 4085 значений, некоторые из значений, оставшихся до $4096=2^{12}$, зарезервированы для кодов конца файла и отметки сбойных секторов). С учетом этих расчетов для дискеты достаточно использовать кластеры размеров в один сектор. Заметим, что сами таблицы FAT на дискете занимают чуть более 4 Кбайт, которые не входят в состав упомянутых 1,44 Мбайт, которые относятся только к областям данных, записываемых на дискету.

В операционной системе MS DOS и еще в немногих Windows95, до сих используемых с жесткими дисками небольшой емкости (до 512 Мбайт), применяется файловая система FAT16. В ней элемент таблицы FAT имеет размер 16 битов. Это позволяет кодировать номерами около 65 тыс. кластеров (соответственно и файлов на таких логических дисках может быть не более указанного числа). Заметим, что для логических дисков с емкостью в интервале от 512 Мбайт до 1 Гбайта эта система использует кластеры размером в 16 Кбайт, а для логических дисков с емкостью в интервале от 1 Гбайта до 2 Гбайт – даже размером в 32 Кбайт. Неэффективность таких вариантов следует из того, что средняя потеря от последнего не полностью заполненного кластера файла теоретически имеет величину в половину размера такого кластера. Поэтому считается, что средняя потеря дисковой памяти при использовании FAT в последнем варианте составляет 16 Кбайт на файл. В действительности потери еще значительно больше, если используется множество маленьких файлов, не занимающих и половины одного кластера.

Современные файловые системы типа FAT, начиная с самых первых в 80-х годах, используют записи оглавления размером 32 байта. Это решение сохранилось и для самых поздних модификаций (VFAT è FAT32). В классическом варианте запись оглавления как структура данных разбита на 8 частей, которые могут быть описаны структурой языка Си:

```
struct {
char name[8]; // имя файла или каталога, дополненное справа пробелами до 8
char ext[3];//расширение имени файла, дополненное справа пробелами до 3
char attr;       // атрибуты файла
char reserved[10]; // зарезервированное поле
FTIME time;       // время создания или последней модификации файла
FDATE date;       // дата создания или последней модификации файла
unsigned short cluster_nu;  // номер начального кластера размещения файла
unsigned  long size;  // размер файла
} FITEM;
```

где структуры для задания времени и даты описываются, в свою очередь как

```
struct {
unsigned short sec : 5, min : 6, hour : 5;// секунды, минуты, часы
FTIME;
struct {
unsigned short day : 5, month : 4, year : 7;// день, месяц, год
FDATE;
```

Таким образом, в ФС FAT вся вспомогательная информация к данным собственно файла хранится в двух структурах: каталоге и таблице FAT, причем запись в файл приводит к модификации не только кластеров хранения данных в файле и

таблицы размещения, но изменяется и содержимое элемента каталога, а именно изменяются поля time, date и size этого элемента.

Структура файловой системы FAT на жестком диске имеет вид, представленный в табл. 6. (перечисляются разделы системы в порядке возрастания номеров секторов).

Табл. 6.1. Структура файловой системы FAT

Загрузочный сектор с характеристиками системы.Первая копия таблицы FAT.Вторая копия таблицы FAT.Корневой каталог.Область данных (кластеры, пронумерованные, начиная со значения 2).

На пронумерованные кластеры разбита только область данных раздела жесткого диска, отформатированного как файловая система FAT. Нумерация этих кластеров ведется от нулевого значения, несмотря на то, что на диске соответствующие им секторы нумеруются не с нуля. Внутренняя перенумерация осуществляется программными компонентами файловой системы. (Есть маленькая тонкость, заключающаяся в том, что реальная нумерация в FAT ведется не от 0, а от 2, а два первых элемента FAT используются для служебных целей.)

Две копии таблицы FAT используются для повышения надежности хранения файлов. Специальные программы, в частности выполняемая при загрузке программа scandisk, сравнивают содержимое этих таблиц и могут обнаружить нарушение структуры файловой системы по их различию. Каждая из этих таблиц для FAT16 может иметь размер до 128 Кбайт и должна храниться во время работы компьютера в оперативной памяти. Нетрудно видеть, что даже при наличии нескольких логических дисков со структурой FAT16 это занимает незначительную часть оперативной памяти.

6.4. Современные модификации файловой системы FAT

Модификации файловой системы FAT возникли как не планировавшийся результат продолжения использования 32-битной модификации для операционной оболочки Windows 3.1 над MS DOS, которая при своем создании получила название Windows 95. В начале выпуска Windows 95 была снабжена промежуточной модификацией базовой файловой системы FAT16/FAT12, получившей название VFAT. Основная задача, решаемая VFAT, заключается в предоставлении пользователям длинных имен файлов. Исходная структура каталогов FAT позволяла использовать имена с собственной длиной не более 8 символов, что широким массам неквалифицированных пользователей казалось достаточно неудобным. Поэтому введенные изменения затронули структуры только каталогов этой системы. Причем из-за необходимости преемственности были сохранены и все старые структуры.

Изменения заключаются в том, что для именования файла стали использовать в общем случае не одну запись оглавления, а набор последовательно размещенных в каталоге записей прежнего 32-байтного формата, который был расширен по своим возможностям. Здесь кстати пришлось резервное поле, запасенное в исходной структуре записи FITEM.

Для того чтобы однозначно и невидимо для старых программ отделять новые по строению записи каталога от старых, использовали поле атрибутов *attr*. В этом поле для новых записей стали записывать код 0x0F, недопустимый в старом варианте (он задавал, что это – файл скрытый, системный, доступный только для чтения и одновременно, что это – не файл, а каталог). Программы, работающие со старыми форматами записей каталога, "не замечают" записи с таким кодом в поле атрибутов.

Старые структуры записей каталогов стали хранить "короткое" имя по старым соглашениям. Короткое имя файлов для файлов по старым соглашениям совпадают с этими старыми именами. Короткое имя для файлов с длинными названиями получается "отрезанием" первых 6 (или, если окажется необходимым, менее) символов и присоединением к ним служебного символа ~ (тильда), за которым идет номер, отличающий короткое имя от уже имеющихся коротких имен. Символы, недопустимые по старым соглашениям, заменяются в коротких именах символами подчеркивания.

Полное длинное имя (long name) распределяется по дополнительным новым записям каталога, которые обязательно предшествуют записи со старым форматом и коротким именем. Причем полные имена в новых соглашениях помещаются в записи каталога в коде Unicode, который для каждого символа требуют двух байтов.

Структура дополнительных записей каталога описывается на языке Си в виде

```
struct {
char num; // порядковый номер доп. записи + 64 для последней записи из них
char dop5[10];    //очередные 5 символов имени в Unicode
char attr;        // признак дополнительной записи, всегда равный Ox0F
char type;        // тип, всегда должен быть равен 0
char cntrsum;     // контрольная сумма
char dop6[12];    // следующие 6 символов имени в Unicode
unsigned short cluster_nu;  // всегда должен быть 0
char dop2[4]      // следующие 2 символов имени в Unicode
} NEWFITEM;
```

Байт *cntrsum* контрольной суммы для длинного имени рассчитывается как остаток от деления на 256 суммы значений определенных полей соответствующей записи для старого формата. В операционной системе Windows 95 это поле используется для выявления испорченных записей в каталоге для длинных имен. До-

полнительные записи каталога для длинного имени файла, если этих записей более одной, размещаются в обратном порядке: вначале (в секторах с меньшим номером) последняя дополнительная запись, затем предпоследняя и т.д., доходя до первой дополнительной записи каталога, непосредственно за которой будет следовать стандартная запись короткого имени файла. Заметим, что принятые Windows информационные средства для записи внутри файловой системы длинных имен достаточно "разорительны" (неэффективны). В частности, они нерационально используют область каталога и требуют громоздко построенных служебных программ, размещая информацию имени файла по очень нерегулярным правилам.

Microsoft, создавая Windows 95, планировала использовать последнюю очень недолгое время, пока более основательная ОС этой же фирмы – Windows NT – не станет основной у ее пользователей. Но массовый пользователь в течение ряда лет не очень соблазнялся этой чрезвычайно мошной, но и более медлительной и сложной в управлении ОС. Поэтому с учетом запросов реального рынка была создана 32-битная файловая система для Windows 95, получившая название FAT32.

Основное изменение в этой системе относительно ее прототипа заключалось в переходе от 16-битных элементов таблицы FAT к 32-битным. Таким образом, потенциальное множество нумеруемых кластеров возросло с 65 тыс. до почти 4 млрд., что значительно пока превосходит число секторов на жестких магнитных дисках.

Структура записи каталога для FAT32 описывается на языке Си следующим образом:

```
struct {
char DIR_Name[11];// имя файла или каталога, дополненное справа пробелами
                                                    //  до 11 символов
char DIR_Attr;          // атрибуты файла
char DIR_NTRES;    // зарезервированное для NT поле, должно быть равно 0
char DIR_CrtTimeTenth // уточняет время создания файла,
                        // задает число десятков мс.(от 0 до 199)
FTIME DIR_CrtTime;  // время создания файла
FDATE DIR_CrtDate;  // дата создания файла
FDATE DIR_LstAccDate;  // дата последнего обращения к файлу для записи
                                    // или считывания данных
unsigned short DIR_FstClassHI; // старшее слово первого кластера файла
FTIME DIR_WrtTime;   // время выполнения последней операции записи в файл
FDATE DIR_WrtDate;  // дата выполнения последней операции записи в файл
unsigned short DIR_FstClassLO;          // младшее слово первого кластера файла
unsigned  long DIR_FilesSize;             // размер файла
} FITEM;
```

Как видим, структура каталога заметно изменилась. Расположение информации о коротком имени файла осталось на своем месте, но из резервных большая часть используется. Номер первого кластера файла теперь задается двумя 16-битными словами, составляющими в совокупности 32 бита, как и следует для 32-битной нумерации кластеров. Основные изменения коснулись расширения временной учетной информации для файла. На смену единственным значениям даты и времени, отражавшим ранее в 16-битных системах момент последнего изменения, пришли различные поля как для собственно создания, так и для последнего изменения и доступа. Если в 16-битной файловой системе время задавалось только с точностью до 2 секунд (всего 5 битов было выделено в структуре FTIME для битового поля задания секунд), то здесь появилась возможность уточнения такого времени отдельным полем с точностью до одной сотой секунды.

Структура размещения частей файловой системы FAT32 на жестком диске описывается следующей схемой, представленной табл. 6.2. (перечисляются разделы системы в порядке возрастания номеров секторов).

Табл. 6.2. Структура разамещения частей FAT32

Загрузочный сектор.Структура FSInfo.Копия загрузочного сектора.Первая копия таблицы FAT.Вторая копия таблицы FAT.Область данных.

Самое существенное в этом строении, что местонахождение корневого каталога теперь не фиксировано, а указатель на его местонахождение находится в составе загрузочного сектора. (Именно в поле загрузочного сектора со смещением 0x2C от начала сектора 32-битным значением задается номер первого кластера корневого каталога. Обычно он равен 2, и каталог размещается как и в FAT16 сразу после копий таблицы FAT. Наличие этого поля позволяет разместить корневой каталог в любом месте области данных, аналогично тому, как это происходит с остальными каталогами.)

Наиболее употребительный размер кластера, используемый файловой системой FAT32, составляет 4 Кбайта (он определяется при форматировании логического диска с целью достичь некоторой оптимальности использования дисковой памяти). Если при этом сам логический диск имеет объем N Мбайт, то для его области данных будет выделено около 256N кластеров, поэтому размер каждой из FAT таблиц будет иметь величину порядка 1000N байтов. В частности, для логических дисков размером в 20 Гбайт, каждая FAT будет иметь размер порядка 20 Мбайт. Если на самом жестком диске такой расход под служебную информации достаточно приемлем, то хранение двух экземпляров этих таблиц в оперативной памяти потребует уже 40 Мбайт, что уже достаточно чувствительно для этого более ограниченного ресурса. Заметим, что выбор размера кластера величиной 2 Кбайта для примера влечет еще более заметное расходование оперативной памяти (в данном случае до 80 Мбайт). Использование логических дисков FAT32 раз-

мером во многие десятки и более Гбайт увеличивает эти расходы еще на порядок. В частности при 100-Гбайтном логическом диске потребовалось бы уже 200 Мбайт оперативной памяти только для хранения FAT, что непосредственно соизмеримо с общим объемом памяти, до последнего времени устанавливаемой на персональный компьютер. Именно поэтому в систему FAT32, подобно FAT16, заложена возможность использования кластеров различного размера, что позволяет сократить объем хранимых в памяти таблиц. Тем не менее большой объем таблиц FAT32 был причиной введения в структуру служебной структуры FSInfo, которая для быстрого доступа хранит информацию о числе свободных кластеров на диске и номере первого свободного кластера в таблице FAT.

В начале таблицы FAT имеются два зарезервированных элемента (подобно тому, как это имело место и для FAT16). Нулевой по нумерации элемент содержит только условный код "среды" (BPB_Media), используемый для дополнительного обозначения типа FAT (16 или 32). В следующем элементе FAT – единичном по индексу, отсчитываемому с нуля – старший бит задает "чистый" (или "грязный") логический диск (бит, имеющий значение 0x80000000 или 0x8000, соответственно для FAT32 или FAT16). "Грязный" диск (dirty) обозначает, что процедура разгрузки операционной системы не была выполнена соответствующим образом, и логический диск поэтому может содержать ошибки размещения. Этот бит предназначен для проверки программами загрузки операционной системы.

6.5. Особенности построения файловой система HPFS

Прежде чем перейти к более известной файловой системе NTFS, рассмотрим ее идейную предшественницу HPFS. Файловая система HPFS, призванная устранить недостатки системы FAT16 для персонального компьютера, появилась в 1989 г. в операционной системе OS/2.

Эта файловая 32-битная система самого начала поддерживала длинные имена (до 255 символов). Кроме этого, ее основными отличительными особенностями от FAT16 были:

- каталог в середине раздела логического диска;
- бинарное дерево поиска;
- битовые карты;
- использование сектора как единицы распределения памяти.

Структура каталогов HPFS с самого начала включала время и дату последнего изменения, а также время и дату последнего доступа (в Windows в варианте FAT32 это появилось позже на целых семь лет).

Еще одной характерной особенностью HPFS явилось автоматическое хранение "расширенных атрибутов", в качестве которых предполагалось хранить и индивидуальные пиктограммы для файлов и каталогов. Практически это – самая бы-

страя и эффективная файловая система за пределами Unix систем. Она обладает высокой производительностью (файлы большого размера копируются внутри ее в несколько раз быстрей чем в FAT и NTFS). Ее отличает высокая надежность, быстрое восстановление после аварийного отключения, чрезвычайно эффективное использование дискового пространства.

Внутренняя структура HPFS использует битовые карты и экстенты для хранения файла. (Напомним, что экстент – это фрагмент файла, расположенный в смежных секторах.) При создании файла (его первого и в лучшем случае единственного экстента) или выделении еще одного экстента HPFS предоставляет ему место по возможности "вразброс", с большим интервалом последовательных свободных секторов до и после этого экстента. Такая стратегия, как правило, обеспечивает возможность расширения текущих экстентов файла на их неизменном месте. Эксперименты и исследования показали, что если файловая система HPFS заполнена файлами не более чем на 80%, то в среднем менее 3% файлов занимают два и более экстентов, подавляющее большинство файлов размещаются всего в одном экстенте.

Основная информация о размещении файла или каталога находится в так называемом файловом узле (Fnode), там же содержатся расширенные атрибуты файла. Все основные блоки системы HPFS размещаются непосредственно в отдельных секторах дисковой памяти размером 512 байтов. Этим обеспечивается очень эффективное размещение с минимальными потерями из-за дискретности дисковых представлений информации.

В частности, Fnode занимает отдельный сектор. При возможности Fnode размещается перед первым блоком хранения (все блоки хранения – также 512 байтов) своего файла. (Это дает возможность считывать и Fnode и начальные блоки файла одной операции чтения с магнитного диска.) Размещение файла описывается в Fnode парами 32-битных чисел: указателем на первый блок экстента и его длиной, выраженной числом секторов в нем. В одном Fnode можно поместить информацию только о 8 экстентах. Если требуется больше (что бывает чрезвычайно редко), то в Fnode помещается указатель на дополнительный блок размещения (allocation block), который содержит до 40 указателей на экстенты или другие блоки размещения.

Каждая запись в каталоге HPFS содержит:

- длину записи;
- байт флагов (атрибутов) со значениями Archive, Hidden, System, ReadOnly;
- указатель на Fnode для данного файла;
- время и дату создания;
- время и дату последнего обращения к файлу;
- время и дату последней модификации файла;
- длину расширенных атрибутов;

- счетчик обращений к файлу;
- длину имени файла;
- имя файла длиной до 255;
- указатель связывающий файлы в бинарное дерево каталога.

Битовая карта в структуре HPFS не одна, а несколько и каждая предназначена для обслуживания отдельного участка области данных. Более детально структуру HPFS можно изобразить в виде последовательности

Служ. область| *Группа1*|BM1|BM2| *Группа2*| *Группа3*|BM3|BM4| *Группа4* ...

Часть этой структуры, состоящая из группы блоков (на схеме *Группа k*) и находящейся рядом с ней битовой карты BMk, названа *полосой* (band). Входящая в такую полосу битовая карта описывает занятость блоков данных в группе данных этой полосы. Размер полосы выбран разработчиками фиксированной величины (8 Мбайт).

Одна из полос, размещенная в середине логического диска названа полосой каталога (directory band) и хранит каталоги диска. В ней наряду с другими каталогами находится корневой. Логическая структура каталога – сбалансированное двоичное дерево с ключевыми записями имен файлов в алфавитном порядке.

Эта структура состоит из корневого блока (root block) и оконечных блоков (leaf blocks). Каждый из этих блоков содержит по 40 записей. Каждая запись корневого каталога указывает на один из оконечных блоков, а каждая запись в оконечном блоке – либо на Fnode, либо на оконечный блок следующего уровня.

Служебная область разделяется на три части: загрузочный блок (BootBlock), дополнительный блок (SuperBlock) и резервный блок (SpareBlock). В SuperBlock содержится указатель на список блоков битовых карт (bitmap block list), указатель на список дефектных блоков (bad block list), указатель на группу каталогов (directory band), указатель на файловый узел (Fnode) корневого каталога. В этом же дополнительном блоке содержится дата последней проверки диска.

Резервный блок служит для резервирования сбойных блоков и хранит карту замещения (hotfix map) их в виде пар слов (номер сбойного, номер запасного блока для замены). Кроме того, в резервном блоке есть флажок Dirty File System, который устанавливается после открытия любого файла на диске и сбрасывается при закрытии всех файлов и сбросе кэша всех отложенных буферов на диск. Практически этот флаг сбрасывается только после нормального завершения разгрузки системы (ShutDown). В процессе загрузки этот флажок проверяется на всех томах HPFS.

Все файловые объекты HPFS имеют указатели на родительские и дочерние блоки. Избыточность этих взаимосвязей позволяет программе CHKDSK полностью восстанавливать строение логического диска, поврежденного при аварийном выключении или сбое.

6.6. Принципы построения файловой системы NTFS

Файловая система NTFS, во-первых, была создана с оглядкой на HPFS, во-вторых, явилась последовательной реализацией ряда категорических принципов. Связь с HPFS видна даже в том, что коды этих систем, рассматриваемые менеджерами загрузки и заносимые в ключевую структуру магнитного диска Master Boot Record, совпадают. (Все остальные файловые системы отличаются друг от друга по этому коду.)

Файловая система NTFS является уникальным образцом безусловного следования некоторым базовым принципам. Хотя большинство технических систем, используя ряд принципов, как правило, сочетают их в некоторой практической взаимосвязи, NTFS строго следует выбранным принципам. (Существует авторитетное мнение, что бескомпромиссная реализация технических принципов в практическом плане может служить только для экспериментов и демонстрации границ их технической применимости.)

В качестве таких принципов NTFS использует три: атрибуты защиты для всевозможных частей операционной системы, реляционную модель данных и рассмотрение любой частично самостоятельной части дисковой информации как файл. (Все есть файл, вся информация – в базе данных и все нужно непременно с одинаковыми усилиями защищать.)

Преемственность к файловым системам FAT внутри NTFS заключается в использовании кластеров вместо секторов. Такое решение не обеспечивает очень эффективного использования дискового пространства, но сокращает объем служебной информации внутри файловой системы.

Согласно реляционному принципу, вся информация в NTFS организована в виде большой таблицы базы данных. Эта таблица называется Master File Table (MFT). Каждому файлу в этой таблице отвечает некоторая строка (запись файла в MFT). (Как собственно файлу с именем, видным пользователю, так и внутренним данным, искусственно оформленным как файл.) Номер строки (записи) в этой таблице служит для внутренней нумерации файлов в NTFS.

Обобщенный состав таблицы MFT, учитывая, что она служит "для всего", изображается следующей схемой:

Номер файла	Назначение
0	Описание самой MFT
1	Копия MFT (неполная)
2	Файл журнала транзакций
3	Файл тома
4	Таблица определения атрибутов
5	Корневой каталог

6	Файл битовой карты
7	Загрузочный файл
8	Файл плохих кластеров
. . .	. . .
16	Далее - пользовательские файлы и каталоги
. . .	. . .

Первые 16 строк этой таблицы описывают так называемые *метаданные* – служебные структуры данных файловой системы.

Загрузочные секторы логического диска рассматриваются здесь как файл (седьмой в таблице – boot file) и могут при работе NT модифицированы путем обычных файловых операций. Но этот "файл" имеет определенные атрибуты защиты, которые не позволяют такую модификацию выполнять никому, кроме специальных системных программ NT. Заметим, что сами загрузочные секторы по техническим причинам (обеспечения автоматической загрузки еще в отсутствии операционной системы) размещаются в фиксированных первых позициях логического диска.

Чтобы перейти затем к более сложным вещам, коротко рассмотрим назначение вспомогательных метаданных. Файл плохих кластеров (bad cluster file) аналогичен по своему назначению карте сбойных блоков в HPFS. Файл тома (volume file) содержит учетную информацию файла: имя тома, версию NTFS, для которой отформатирован том, и бит Dirty File System, сигнализирующий, что содержимое тома может быть повреждено и при загрузке необходимо выполнить утилиту Chkdsk. Назначение файла атрибутов более специально, он задает, какие атрибуты поддерживает том и указывает, можно ли их индексировать (можно ли индексировать внутреннюю систему упорядочивания файлов не только по имени, но и по каким-то другим атрибутам), восстанавливать операцией восстановления системой и т.д.

Битовая карта в NTFS одна на весь логический диск (том) и описывается файлом битовой карты. В процессе работы NTFS записывает промежуточные операции в журнал транзакций, включая создание файла и любые команды (copy, mkdir), изменяющие структуру каталогов. Этот файл применяется в процессе восстановления тома NTFS после сбоя системы.

Неполная копия MFT соответствует служебному файлу, находящемуся в середине диска и содержащему значения первых 16-ти строк таблицы MFT. Предполагается, что эта копия будет использоваться, если по каким-то причинам нельзя прочитать первые строки оригинальной таблицы MFT.

При описании внутреннего содержания структур, описывающих размещение файлов в NTFS, используются термины – *логический номер кластера* LCN (logical clusters numbers) и *виртуальный номер кластера* VCN (virtual clusters numbers).

Логические номера кластеров обозначают последовательные номера кластеров тома, начиная с номер 0 – самого первого кластера в томе. Таким образом, логические номера кластеров непосредственно определяют местонахождение кластера на диске. Виртуальные номера кластеров нумеруют кластеры, принадлежащие конкретному файлу, причем кластер, где содержится начало файла получает виртуальный номер 0, следующий кластер хранения файла – 1 и т.д. Поэтому внутри NTFS возникает задача, как перейти от виртуальных номеров кластеров данного файла к логическим номерам этих же кластеров. Таблицы размещения, сложным образом встроенные в NTFS, и задают такие отображения от VCN к LCN.

Всю таблицу MFT с внешних позиций можно рассматривать как полный аналог реляционных таблиц данных, где некоторые поля имеют значения переменного размера. Каждая строка этой таблицы толкуется как набор *атрибутов таблицы* (не путать с собственными атрибутами файла!). Таким образом, отдельный столбец этой таблицы – это некоторый атрибут. Для удобства программистов и описания эти столбцы-атрибуты имеют собственное наименование, хотя это и необязательно. В частности, используются обозначения атрибутов $FILENAME, $DATA. (На самом деле эти имена соответствуют числовым значениям типа, которые NTFS использует для упорядочивания атрибутов внутри файловой записи – строки MFT.)

Размер каждой строки таблицы MFT задается при форматировании и может иметь значение 1, 2 или 4 Кбайт. Обычно он выбирается совпадающим по размеру с размером кластера диска.

В начале каждого атрибута (т.е. содержательного поля строки MFT) расположен *стандартный заголовок*, содержащий информацию об этом атрибуте. Стандартный заголовок атрибута используется программами для единообразной обработки полей таблицы. (Тем самым MFT заметно отличается от обычных реляционных таблиц, которые используют постоянное и фиксированное по порядку число содержательных столбцов.) Еще одной особенностью является возможность многократного использования в одной записи одинаковых атрибутов. Например, у файла может быть несколько полей (атрибутов) имен или полей данных. (Поля атрибутов упорядочиваются в строке по возрастанию внутренних числовых кодов атрибутов.)

Значение атрибута (значение поля) может храниться непосредственно в таблице MFT, в таком случае оно называется *резидентским* (resident attribute). Либо же (из-за немалого размера) это значение хранится вне таблицы (вне секторов, отведенных таблице MFT). В последнем случае собственно поле таблицы содержит информацию, задающую местонахождение этого значения. В частности содержимое большого файла неизбежно хранится вне MFT, но формально считается, что оно является значением поля строки таблицы, описывающего этот файл и имею-

щего атрибут с именем $DATA. Заметим, что стандартный заголовок поля (атрибута) всегда является резидентным.

Четыре поля (атрибута) строки MFT являются обязательными: это *стандартная информация записи*, *имя файла*, *дескриптор защиты* и *данные*. Кроме обязательных, которые используются, когда файл полностью размещается непосредственно в записи таблицы MFT, могут применяться еще и следующие атрибуты: *корень индекса*, *размещение индекса*, *битовая карта* (только для каталогов), *список атрибутов* (редко используемый атрибут, применяемый, когда описание файла требует более одной строки таблицы MFT). Резидентными всегда являются стандартная информация и имя файла.

Когда файл описывается более чем одной строкой таблицы, его первая запись, в которой хранится местоположение остальных, называется *базовой файловой записью* (base file record).

Когда в области поля (атрибута) внутри записи MFT недостаточно места для размещения его значения (чаще всего такая ситуация имеет место для данных файла), то в этом поле с помощью заголовка описывается структура, задающая эти данные вне строки таблицы. Эта структура имеет внешний вид таблицы с колонками

Стартовый VCN | *Стартовый* LCN | *Число кластеров*.

В частности, если данные файла реально размещаются в кластерах с номерами 1234, 1235, 1236, 1237, 1348, 1349, 1350, то в такой таблице содержится две строки

0 1234 4
4 1348 3

которые описывают два экстента, длиной в 4 и 3 кластера.

Если в одной записи MFT недостаточно места для размещения многих экстентов хранения файла, то используются дополнительные записи, а в базовой строится поле списка атрибутов, которое содержит имя и код типа для каждого из атрибутов файла, а также файловую ссылку на запись MFT, где находится этот атрибут.

Файл на томе NTFS идентифицируется 64-битным значением, называемым *файловой ссылкой* (file reference). Она состоит из *номера файла* (48-битного номера единственной или базовой записи файла в MFT) и *номера последовательности*. (Номер последовательности носит вспомогательный характер, имеет всего 16 битов и увеличивается всякий раз, когда текущая позиция строки в MFT используется повторно; он служит только для внутренней проверки целостности файловой системы.)

Нерезидентские атрибуты хранятся в отдельных экстентах или группе экстентов. В этом случае заголовок атрибута содержит информацию, необходимую для нахождения значения атрибута на диске.

Каталог в NTFS – это так называемый в терминологии баз данных *индекс*, т.е. набор имен файлов или каталогов с соответствующими файловыми ссылками. Причем этот набор организован как двоичное дерево для ускорения доступа. Для больших каталогов (которые не помещаются в запись MFT) имена файлов хранятся в *индексных буферах* – экстентах размером в кластер. Именно индексные буферы совместно с областью в поле *корня индекса* реализуют структуру двоичного дерева. В каждом индексном буфере может помещаться от 15 до 30 записей для имен файлов (отдаленных аналогов полей FITEM в FAT каталогов). В общем случае, не все места для таких записей в индексном буфере оказываются занятыми, и при добавлении новых файлов для структуры двоичного дерева нужно найти место в одном из индексных буферов (или создать новый индексный буфер, если свободных мест нет). Для решения этой проблемы – добавления файла в каталог – используются битовые карты для каталогов. Они отражают текущую заполненность имеющихся для каталога индексных буферов.

Каждая запись для имени файла в каталоге хранит как имя файла, так и информацию о временных отметках файла и размер файла. Тем самым дублируется часть стандартной информации из строки MFT, описывающей файл. Как следствие, обновление информации о файле приходится делать в двух местах: в записи о самом файле в MFT и в структуре каталога, содержащего этот файл. Это замедляет работу файловой системы, но зато соответствует старым традициям FAT. (В оправдание обычно замечают, что такое решение ускоряет выдачу справочной информации по заданному каталогу.)

Корень индекса содержит отсортированный перечень записей файла в каталоге. В простейшем случае этим и ограничивается построение структуры двоичного дерева. Если же часть его размещается в индексных буферах, то от каждой записи отсортированного перечня в корне индекса задается указатель на индексный буфер в виде VCN, содержащий имена файлов, которые в лексикографическом порядке предшествуют имени в записи отсортированного перечня. Поле размещения индекса задается в обычной форме таблицы отображения VCN ->LCN. В частном случае указатель может отсутствовать для некоторых файлов перечня. Записи в индексных буферах организованы аналогичным образом и могут ссылаться на индексные буферы более низкого уровня.

Например, отсортированный перечень файлов <file4, file10, file15> через отображение виртуальных кластеров ссылается на индексные буферы с записями (также отсортированными) для файлов <file0, file1, file3>, <file6, file8, ,file9> и <file11, file12, file13,file14>. Причем второй из перечисленных индексных буферов имеет незаполненную запись на третьем месте.

Подробней о строении файловой системы NTFS можно прочитать в книге [5].

6.7. Особенности строения файловых систем для Unix

Принципиальной особенностью файловых систем для Unix является хранение минимальной информации в записях оглавления. Так, в современной расширенной файловой системе для Linux, называемой в сокращении ext2, структура записи каталога описывается структурой

```
struct dir
{unsigned long  inode_num;
 unsigned short  rec_len;
 unsigned short name_len;
 char name[256];     /* between 0 and 256 chars */
};
```

В самой старой файловой системе для Unix, созданной еще в 70-х годах XX века, запись каталога занимала всего 16 байтов, из которых 14 байтов предназначались под имя файла (или каталога), а два оставшихся хранили значение индексного узла (*inode*).

В современном решении запись каталога имеет переменную длину, определяемую для каждой записи полем *rec_len* и содержит имя длинной до 255 символов, причем поле *name_len* задает действительное значение этого имени. Такое решение связано со стремлением, как допускать очень длинные имена и в то же время не делать очень большими записи каталога. По существу, поля *rec_len*, *name_len* – вспомогательные, и запись каталога несет все ту же содержательную информацию.

Вся информация о файле (каталоге), за исключением его имени в текущем каталоге, хранится в специализированной структуре файловой системы, называемой *индексным узлом* (inode). Именно в нем хранится время создания файла (каталога), время последней модификации, время последнего доступа, число ссылок на файл, размер файла. Более подробная информация представлена в описании структуры *inode*:

```
typedef struct
{  unsigneg  short i_mode;        /* File mode */
   unsigneg  short i_uid;         /* Owner Uid */
   unsigneg  long i_size;         /* Size in bytes */
   unsigneg  long i_atime;        /* Access time */
   unsigneg  long i_ctime;        /* Creation time */
   unsigneg  long i_mtime;        /* Modification time */
   unsigneg  long i_dtime;        /* Deletion Time */
   unsigneg  short i_gid;         /* Group Id */
   unsigneg  short i_links_count; /* Links count */
   unsigneg  long i_blocks;             /* Blocks count */
   unsigneg  long i_flags;        /* File flags */
   unsigneg  long i_reserved1;    /* Reserved 1 */
   unsigneg  long i_block[15];    /* Pointers to blocks */
```

```
    unsigneg  long i_version;       /* File version (for NFS) */
    unsigneg  long i_file_acl;      /* File ACL */
    unsigneg  long i_dir_acl;             /* Directory ACL */
    unsigneg  long i_faddr;         /* Fragment address */
    unsigneg  char i_frag;          /* Fragment number */
    unsigneg  char i_fsize;         /* Fragment size */
    unsigneg  long i_reserved2[2];/* Reserved 2 */
} inode;
```

6.8. Программный опрос файловой системы

Операции по получению данных из файлов и запись в файлы составляют одну из основ программирования и в применении к отдельным ОС уже рассматривались в гл. 2. Кроме этих общеупотребительных функций нередко возникает проблема нахождения в некотором каталоге файлов, удовлетворяющих определенным условиям. В частности, иногда требуется получение информации, какие файлы находятся в некотором каталоге. Для решения подобных задач в операционные системы включены системные функции просмотра содержимого каталогов.

В ОС Unix операции с каталогом строятся подобно операциям с типизированными файлами, используемыми в Паскале, а именно, вводится указатель на структуру данных, описывающую каталог. Эта структура описана в заголовочном файле dirent.h и имеет имя DIR. Указатель на эту структуру данных используется для получения значения от функции opendir, имеющей прототип

DIR* opendir(char *dirname),

где единственный аргумент задает имя того каталога, из которого требуется получить информацию. При невозможности открыть указанный аргументом каталог функция возвращает значение NULL.

Дальнейшие действия выполняются системной функцией с прототипом

struct dirent *readdir(DIR* dirptr),

с аргументом, полученном от предыдущей функции. Каждое выполнение вызова readdir() возвращает указатель на содержимое структуры типа *dirent*, содержащей информацию об очередном элементе каталога. Эта структура данных описана также в заголовочном файле dirent.h. В последней структуре два основных поля, которые заданы в ней как

```
ino_t  d_ino;                 /* Номер индексного дескриптора */
char   d_name[ ];   /* Имя файла, заканчивающегося нулевым байтом*/
```

При использовании этих полей каталога следует иметь в виду, что нулевое значение поля *d_ino* вполне возможно у используемого каталога и обозначает неиспользуемую запись в каталоге (обычно по причине удаления информации о файле из данного каталога).

После окончания использования указателя на каталог, полученный от функции opendir(), следует выполнить закрытие доступа к каталогу и освобождение ресурсов вызовом функции с прототипом

int closedir(DIR* dirptr),

Вспомогательной функцией работы с каталогами служит описываемая прототипом

void rewinddir(DIR* dirptr),

которая позволяет вернуться к началу каталога с целью чтения его опять с самого начала.

Применение описанных функций демонстрирует программа примера, приведенного в листинге 6.8.1.

```
#include <unistd.h>
#include <dirent.h>
#include <string.h>

int main()
{DIR *dp;
 struct dirent *de;
 int len, rc;

   dp=opendir(".");
   if (dp= =NULL)
         {printf("No those files\n"); exit(1);}
   while (de=readdir(dp)) {
    if (de->d_ino != 0)
      {len=strlen(de->d_name);
       if (!strcmp(".c",(de->d_name)+len-2))
         printf("%s\n",de->d_name);
      }
   }
   closedir(dp);
  return 0;
}
```

Листинг 6.8.1. Программный доступ к информации каталога в Unix

Эта программа последовательно читает все записи текущего каталога (обозначаемого символом '.'), и если запись не пуста, то проверяет, не оканчивается ли имя файла, заданного в этой записи на цепочку символов ".c". При совпадении выполняется вывод имени файла на экран.

В операционных системах типа Windows и OS/2 разработчики включили проверку условия для имени файла в действия соответствующей системной функции. В этих ОС основных функций, работающих с содержимым каталогов, – две: функция поиска первого файла по задаваемому условию и функция поиска следующего файла по тому же условию. Само условие задается как метанотация, т.е. записью совокупности файлов с помощью метасимволов * и ? в соответствующем аргументе имени файла. Использование этих метасимволов полностью совпадает с традиционным их применением в командах операционной системы, восходящим к правилам командного интерпретатора Unix.

В Windows для поиска первого файла в каталоге служит функция с прототипом

HANDLE FindFirstFile(char *metaname, WIN32_FIND_DATA *FindFileData),

где аргумент *metaname* задает метанотацию файла в текущем каталоге или в явно заданном в аргументе каталоге, а второй аргумент задается адресом (указателем на) экземпляра структуры данных, куда должна быть помещена служебная информация о файле. Поля этой структуры данных, описывающей кроме имени файла еще и вспомогательную информацию, дают среди прочего время последней коррекции и последнего доступа (а также ряд других параметров). Наиболее значимым является поле с именем *cFileName*, описывающее имя файла массивом символов. При неудаче функция возвращает значение INVALID_HANDLE_VALUE (равное -1), в противном случае она возвращает специальный хэндл, предназначенный для использования только в функции продолжения поиска и закрытия этого хэндла. По существу, упомянутый хэндл соответствует хэндлу каталога, который получается от функции открытия каталога в Unix.

Для поиска следующих файлов, удовлетворяющих той же метанотации, что была задана при выполнении функции FindFirstFile, в Windows служит функция с прототипом

BOOL FindNextFile(HANDLE hFindFile, WIN32_FIND_DATA *FindFileData),

где аргумент *hFindFile* должен быть получен от функции FindFirstFile, а второй аргумент задает экземпляр структуры для размещения служебной информации о файле и уже рассматривался для предыдущей функции. Последняя функция возвращает значение TRUE, если находит очередной файл в текущем каталоге, удовлетворяющий метанотации, в противном случае она возвращает значение FALSE.

В завершение работы с каталогом должна вызываться функция с прототипом

BOOL FindClose(HANDLE hFindFile),

которая закрывает хэндл, ранее полученный от функции FindFirstFile.

Следующая программа для Windows, приведеная в листинге 6.8.2, демонстрирует поиск в текущем каталоге файлов, которые имеют расширение ".c", с выво-

дом их имен на экран. По существу эта программа полностью соответствует программе в листинге 6.8.1.

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

int main()
{HANDLE fdirsearch;
 WIN32_FIND_DATA dan;
 BOOL rc;

   fdirsearch=FindFirstFile("*.c", &dan);
   if (fdirsearch= =INVALID_HANDLE_VALUE)
         {printf("No those files\n"); exit(1);}
   do {
   printf("%s\n",dan.cFileName);
   rc=FindNextFile(fdirsearch, &dan);
   } while(rc!=FALSE);
   FindClose(fdirsearch);
  return 0;
}
```

Листинг 6.8.2. Программный доступ к информации каталога в Windows

Для вспомогательных действий по переустановке текущего каталога предназначена в Windows функция с прототипом

BOOL SetCurrentDirectory(char *PathName)),

а также функция с прототипом

DWORD GetCurrentDirectory(DWORD BufferSize, char *Buffer),

которая позволяет запомнить в символьном массиве полное имя текущего каталога.

7. ОБЕСПЕЧЕНИЕ МНОЖЕСТВЕННОСТИ ПРОЦЕССОВ

7.1. Основные понятия теории вычислительных процессов

В профессиональном программировании *процессом* называют действия, выполняемые над конкретными данными под управлением программы. Следует отметить существенное отличие процесса от программы и даже от выполнения программы. Процесс разворачивается во времени, это динамический объект, программа – это статический, неизменяемый объект. (Заметим, что программы, модифицирующие сами себя, практически никогда и нигде не применяются, а теоретиче-

ски – не способны быть надежным средством обработки данных.) Замечательной особенностью современных операционных систем является их способность обеспечивать одновременное выполнение нескольких программ, даже если в компьютере используется лишь один процессор (последнее имеет место практически во всех персональных компьютерах). Для начального представления можно сказать, что одновременное выполнение нескольких программ в одном компьютере и составляет несколько процессов. Для большой ясности следует уточнить, что на основе одной программы в компьютере могут одновременно существовать (выполняться) несколько процессов! Например, хотя такая ситуация скорее характерна для мощных серверов обработки данных, в одном компьютере может одновременно компилироваться несколько текстов программ с одного и того же языка и одним компилятором. Компилятор – программа, она присутствует в оперативной памяти в одном экземпляре, но выполняет обработку нескольких исходных текстов. Каждая такая обработка определяет процесс, но все они связаны с одной программой. Зато обрабатывают различные данные – исходные тексты. При подобной организации процессов, когда несколько таких процессов одновременно обрабатывают различные данные, пользуясь при этом единственным общим экземпляром программы, приходится принимать особые меры. Они заключаются в построении такой программы как *реентерабельной* (reenterable – повторно входимой).

Реентерабельной называют программу, которая обеспечивает правильное функционирование множества одновременных процессов, функционирующих на ее основе. Нижний уровень программирования современных архитектур процессоров и современные системы разработки программ ориентированы на построение именно таких программ. Современные операционные системы также строятся таким образом, чтобы по возможности поддерживать реентерабельность. К сожалению, технические особенности, которые мешают или способствуют реентерабельности видны только на уровне машинных кодов и машинных представлений программ, поэтому подробное рассмотрение вопросов реентерабельности возможно только на уровне элементарного программирования с помощью ассемблера и отладчиков, обеспечивающих выдачу информации во внутренних форматах: численных адресах и машинных кодах.

Процессы называются одновременно выполняющимися или *параллельными*, если каждый из них начат, но не завершен. Процессор компьютера может быстро переключаться (с помощью операционной системы и средств системной архитектуры) с одного процесса на другой, запоминая при этом переключении всю необходимую информацию, чтобы в дальнейшем продолжить выполнение процесса с места его приостановки. Системные средства быстрого переключения процессов рассмотрены в ряде технических и учебных источников [17, 19].

Переключение между процессами происходит как независимо от их собственных действий, так и в результате ряда запросов к ОС. Первый вариант имеет ме-

сто, когда ОС решает, что выполняемый процесс уже достаточно долго использует аппаратуру процессора и, руководствуясь управляющей программой, передает использование процессора другому процессу. Простейший запрос к ОС, приводящий к переключению процессов, – это запрос на ввод информации. Запросив, например ввод символа с клавиатуры, процесс вынужден ждать, пока пользователь соберется нажать на клавиши. Для клавиатуры такое ожидание может продолжаться очень долго не только для временного масштаба действий внутри компьютера, но и с точки зрения человека. Было бы нелепо позволить компьютеру в этот период ничего не делать, если его можно загрузить какой-то работой. Более сложные ситуации и потребности ожидания, явно или неявно запрашиваемые у ОС путем выполнения ряда системных функций, будут рассматриваться далее вместе с соответствующими программными средствами.

В современных ОС, кроме собственно процессов, выделяют их более элементарные составляющие, называемые *нитями* (threads). Иное название этих составляющих на русском языке – потоки, кроме того, иногда используется более старое название – *задачи* (или *мини-задачи*). В данном изложении будет использоваться наименование нить, как более близкое к оригинальным техническим названиям разработчиков.

Заметим, что в ряде основополагающих ОС 60-70-х годов многопрограммное функционирование ОС строилось исключительно на понятии и использовании процессов, а нити как составляющие части процессов стали широко применяться относительно недавно. Поэтому теория вычислительных процессов строилась исключительно на понятии абстрактного процесса.

В настоящее время в рамках процесса могут функционировать несколько (и даже много) нитей. В простейших случаях в составе процесса обязательно присутствует одна нить. Собственно процесс в современных разработках и применениях специализирован исключительно на владении ресурсами вычислительной системы. К ресурсам процесса относятся области данных, используемые и обрабатываемые процессом (программой). Более того, оперативная память, которую процесс использует по своему усмотрению, распределяется процессу и временно является как бы его собственностью. В современных ОС такая память распределяется и закрепляется с помощью архитектурных средств виртуальной памяти, которыми управляет соответствующая подсистема ОС. Эта подсистема - одна из наиболее важных и ответственных частей ОС. К другим ресурсам процесса относят открытые файлы, которые конкретизируются для процесса благодаря дескрипторам (учетно-информационным описателям работы с файлом) и обозначаются их логическими номерами – хэндлами (handle). В многозадачных ОС в качестве ресурсов процесса появляются дескрипторы и хэндлы еще ряда более сложных объектов (очередей, конвейеров, окон и т.д.).

Отдельный процесс, выполняемый в компьютере, это не отвлеченное понятие, а конкретный объект для операционной системы. Эта конкретизация обеспечивается тем, что ОС поддерживает процесс, используя внутренний учетный "документ", аналогичный паспорту для гражданского лица или материального технического объекта. (Есть документ – есть процесс, нет документа – нет процесса.) Такой учетный документ представляет собой специализированную структуру данных, которая содержит в своих полях всю информацию, необходимую для учета и оперативного технического сопровождения. Подобная структура данных, предназначенная в первую очередь для использования собственно ОС по традиции называется управляющим блоком процесса (Process Control Block) – сокращенно PCB. Иногда эти управляющие блоки называют также *дескрипторами* процессов. Некоторые ОС предоставляют профессиональному программисту с правами администратора ОС доступ по чтению к отдельным полям PCB; другие, ориентированные преимущественно на непрофессионалов, полностью скрывают структуру и содержание таких блоков. В любом случае такие блоки содержат, как минимум, уникальный идентификатор (некоторое специальное число) для процесса и указатель или иную форму доступа к информации о памяти, доступной процессу В современных ОС последняя информация тесно связана с аппаратурной реализацией доступа к памяти и представляет аппаратно-программные структуры доступа к *таблицам дескрипторов памяти*. Уникальность идентификатора процесса относится ко всей ОС, но для более частных применений процесс может обозначаться еще и хэндлом. Этот хэндл служит для обозначения процесса в некоторой группе процессов. Обычно такой группой является совокупность некоторого процесса и явно, по его приказу, дополнительно созданных процессов (*дочерних* в сложившейся терминологии), они будут рассмотрены позже в этой же главе. Если в операционной системе используются нити, то для них ОС также строит и использует управляющие блоки – Thread Control Block (TCB).

При переключении (во временно приостановленное состояние) процессов, которые непосредственно осуществляет процессор (или процессоры, если их более одного в компьютере), оказывается принципиально необходимым сохранять всю информацию, которая потребуется для продолжения приостанавливаемого процесса после того, как ему будет предоставлена такая возможность. Современные компьютеры строят так, чтобы информация в оперативной памяти, относящаяся к различным процессам, автоматически сохранялась вне зависимости от действий этих процессов. (Эти средства включают защиту памяти процессов друг от друга и подкачку страниц виртуальной памяти, которые будут рассмотрены позже.) Но кроме информации в оперативной памяти, вычислительные процессы современных компьютеров используют еще и аппаратные блоки (временного) хранения информации. Их обычно называют *регистрами процессора*. Совокупность значений, одновременно хранимых процессом в таких аппаратных блоках, называют

контекстом процесса или соответственно *контекстом нити*. Для указания места сохранения контекста процесса или нити используют соответствующие поля их управляющих блоков. Заметим, что в современных архитектурах процессоров сохранение и восстановление контекстов нити осуществляется специальными аппаратными процедурами по упрощенному указанию самого действия – переключения между нитями. В этих случаях структура контекста процесса или нити определяется техническим описанием на архитектуру процессора, в частности технической документацией на архитектуру процессоров Pentium и его модификаций.

После порождения некоторым процессом другого процесса, первый из них становится родительским (parent), а порожденный – дочерним (child). При порождении одним процессом многих дочерних возникают связи между процессами, отражающие эти отношения parent-child. Аналогично естественной иерархии родства, между процессами компьютера образуется при этом иерархия связей parent-child. Она используется для организации совместных работ многими процессами. При этом в простейшем случае процесс, становящийся родителем, порождает процесс для выполнения некоторых вычислительных работ, а сам при этом приостанавливается. Его дочерний процесс выполняет запланированную работу и завершает свои действия специальным обращением к операционной системе, информирующей последнюю об успешном или неуспешном завершении. В ответ на это обращение операционная система переводит приостановившийся родительский процесс опять в функционирующее состояние. При таком порядке выполнения родительского и дочернего процессов их называют *последовательными процессами*.

Для технической организации переключения процессов очень важное значение имеет характеристика, которую называют состоянием процесса. Эта характеристика записывается как одно из важнейших полей в управляющий блок процесса или управляющий блок нити, в зависимости от того, используются нити в операционной системе или нет. Для большей общности изложения и для его некоторого сокращения будем называть *абстрактным процессом* нить, если ОС допускает использование нитей, и обычный процесс – в противном случае. Абстрактный процесс обязательно использует как минимум три различных состояния. Эти состояния называют *состоянием готовности* (ready), *состоянием выполнения* (run) и *состоянием блокировки* или *заблокированности* (blocked). В *состоянием выполнения* процессор выполняет команды (операторы) именно этого абстактного процесса. Иначе говоря, в состоянии выполнения абстрактный процесс владеет аппаратурой процессора. В состоянии готовности абстрактный процесс обладает всем необходимым для продолжения обработки данных, кроме процессора, который в этот момент обслуживает другой абстрактный процесс. Наконец, состояние блокировки отвечает необходимости дожидаться появления некоторого события или явления, без которого продолжить обработку данных абстрактным процессом невозможно. Характерным примером блокировки абстрактного процесса является ожи-

дание выполнения аппаратурой внешнего устройства запроса на ввод, в частности ожидание нажатия на клавишу клавиатуры.

Распределение одного или нескольких наличных процессоров среди готовых абстрактных процессов выполняется на основе *квантов времени* (slice time) и приоритетов. Квант времени представляет собой небольшой (обычно временно фиксированный) промежуток времени, в течение которого абстрактному процессу разрешается занимать процессор. Величина кванта времени имеет порядок долей секунды, чаще всего малых долей секунды. С точки зрения имитации владения процессором со стороны многих процессов целесообразно уменьшать эту величину, но тогда увеличиваются "накладные расходы". Последние заключаются в том, что переключение между отдельными абстрактными процессами (особенно обычными процессами) занимает немало времени и является совершенно непроизводительной затратой этого времени. Все же только использование квантов времени обеспечивает эффективную для интерактивного пользователя работу параллельных программ.

Диаграмма, представленная на рис. 7.1, называемая диаграммой состояний абстрактного процесса, описывает переключения между состояниями этого процесса в ходе его существования в операционной системе.

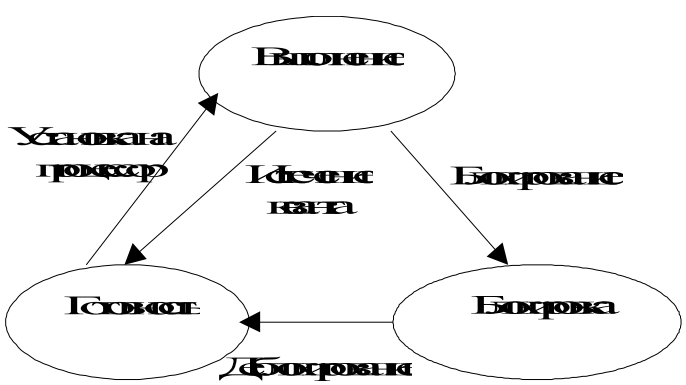


Рис. 7.1. Диаграмма состояний абстрактного процесса

Функционирование операционной системы, когда она обеспечивает выполнение параллельных процессов на основе использования квантов времени, называют *режимом разделения времени*.

Заметим, что так как абстрактных процессов, находящихся в состоянии готовности, как правило, более одного, то все они организуются в одну или более очередей, описываемых как связные списки. Поле связи для такого списка также входит в состав управляющего блока абстрактного процесса. Действия операционной системы по установке на процессор и снятии с процессора в связи с истечением кванта выполняет компонент ОС, называемый *диспетчером* или *планировщиком*

задач. Сами действия этого диспетчера называют *диспетчеризацией* абстрактных процессов.

Когда процессы, более одного, выполняют некоторую вычислительную задачу, возникает проблема согласования их действий. Это согласование, в котором основным моментом является указание и определение готовности данных или освобождение общих ресурсов, называют *синхронизацией* абстрактных процессов. Минимальные средства синхронизации обычных процессов, связанные с завершением работы отдельных процессов будут рассматриваться в этой же главе. Более общие средства синхронизации изложены в гл. 9.

Далее в этой главе рассмотрим обычные процессы, а изучение нитей – в 8 гл.

7.2. Программное порождение процессов

Процесс – не программа, но без программы, которая определяет действия процесса, он функционировать не может. Поэтому при создании процесса, в частности путем заявки другого процесса на создание процесса, как минимум необходимо дать ОС информацию, какой программой воспользоваться при создании и запуске процесса. Заметим, что в дальнейшем процесс может сменить используемую программу соответствующим системным вызовом.

При вызове программы из командной строки иногда используются текстовые аргументы, задаваемые в этой строке после имени программы. Так, например, из командной строки часто нередко архиваторы. В ряде случаев такая возможность незаменима. Для ее использования в качестве аргументов вызова функции создания процесса может быть использована строка аргументов.

Оригинальное и очень изящное решение для создания процессов принято в Unix. Для создания процесса здесь служит системная функция fork(), которая даже не имеет аргументов! Эта функция по существу раздваивает процесс (fork - вилка), в результате в ОС оказываются два почти идентичных процесса. Единственно различие между ними для ОС - их различные идентификаторы, под которыми они зарегистрированы в системе. Различие этих процессов для выполняемой программы в том, что процесс-родитель получает как результат функции *fork()* значение идентификатора нового - дочернего процесса, а дочерний процесс в качестве числа, возвращаемого функцией, получает нуль. Поэтому в программе, общей хотя бы на участке порождения дочернего процесса, анализом кода возврата функции можно решить, какой процесс в анализируемый момент обслуживает программа - родительский или дочерний. Дочерний процесс, как правило, распознав ситуацию и обнаружив себя, использует вызов запуска другой исполняемой программы. Имеется несколько вариантов системных функций такого запуска. Особенностью их в Unix является то, что они не создают новый процесс, но просто позволяют процессу сменить управляющую им программу.

В листингах 7.2.1a и 7.2.1b приведены программы примера создания дочернего процесса, который работает одновременно с родительским и параллельно с ним выводит результаты на экран. Программы этого примера решают уже использовавшуюся демонстрационную задачу: выдают на экран сообщения о шаге цикла вывода для родительского и дочернего процессов.

```
#include <stdio.h>
int main()
{int rc, k;
  printf("Parent Proccess\n");
  rc=fork();
  if (!rc)     {execl("child1.exe",0);   }
 printf("For Child Process:\n");    printf("PID=%d \n", rc);
  for (k=0; k<10; k++)
     {printf("I am Parent (My K=%d)\n", k);    usleep(2000000);       }
  printf("Parent ended\n");   return 0;
}
```

Листинг 7.2.1a. Программа proces.c для Unix

```
#include <stdio.h>
int main()
{int k;

  printf("Demonstration processes, Child Proccess\n");
  for (k=0; k<30; k++)
     {printf(" ------- I am Child ... (child=%d)\n", k);  sleep(1);   }
  printf("Child ended\n");
  return 0;
}
```

Листинг 7.2.1b. Программа child1.c для Unix

Программа proces.c для Unix после сообщения о начале работы процесса-родителя вызывает функцию разветвления *fork*(), возвращающую значение в переменную *rc*. Далее анализируется эта переменная *rc*, при нулевом ее значении вызывается функция замены программы *execl()*, которая использована в виде

execl("child1.exe",0).

Эта функция имеет переменное число аргументов, и последний из них должен иметь значение 0. В более общей форме эта функция позволяет задать аргументы для командной строки программы процесса. После выполнения вызова

execl("child1.exe",0) дочерний процесс управляется программой *child1.exe*, а исходная программа не имеет к нему уже никакого отношения.

Родительский же процесс использует свою программу *proces.exe* дальше участка анализа: выдает на экран сообщение о идентификаторе дочернего процесса

For Child Process PID=*номер*

и переходит к циклу выдачи на экран сообщений о своем очередном шаге. После каждого из этих циклов производится приостановка выполнения функцией *usleep()*. Последняя функция требует задания времени приостановки в микросекундах (а функции приостановки с параметром в миллисекундах в Unix нет!).

Для ОС семейства Windows характерны наиболее сложные функции из рассматриваемых функций создания процессов. Порождение нового процесса достигается в этих ОС функцией *CreateProcess* с десятью параметрами, имеющей прототип:

```
BOOL CreateProcess(LPCTSTR pNameModule, LPCTSTR pCommandLine,
                   SECURITY_ATTRIBUTES *pProcessAttr,
                   SECURITY_ATTRIBUTES *pThreadAttr, BOOL InheritFlag,
                   DWORD CreateFlag, LPVOID penv, LPCTSTR pCurrDir,
                   STARTUPINFO *pstartinfo,
                   PROCESS_INFORMATION *pProcInfo);
```

Здесь тип LPCTSTR задает дальний указатель (адрес в модели FLAT) для текстовой строки, а DWORD аналогичен типу unsigned LONG. Заметим, что параметры *pProcessAttr* и *pThreadAttr* относятся к средствам расширенной защиты в Windows NT и в большинстве приложений не используются, что указывается заданием соответствующего аргумента равным NULL.

Параметр *pNameModule* является указателем на строку текста – на имя файла запускаемой программы (файла с расширением exe), а параметр *pCommandLine* – указателем на текст командной строки запуска программы. С формальной стороны эти два аргумента частично дублируют друг друга, практически чаще всего используется лишь один из них.

Если параметр *pNameModule* задан равным NULL, а используется параметр *pCommandLine* (т.е. он задан не равным NULL), то из текста параметра *pCommandLine* извлекается первое слово – последовательность символов до первого пробельного символа. Это слово используется как имя файла запускаемой программы, причем при отсутствии в нем расширения к нему автоматически приписывается расширение EXE. Это имя вначале используется для поиска исполняемого файла в том же каталоге, где находится EXE-файл процесса, вызывающего функцию CreateProcess(). При не обнаружении требуемого файла, его поиск продолжается в текущем каталоге вызывающего процесса, потом (при еще одной неудаче) – в си-

стемном каталоге Windows, затем в основном каталоге Windows и, наконец, в последовательности каталогов, перечисленных в переменной окружения PATH.

Параметр *pNameModule* позволяет задавать полное имя запускаемой программы или ее имя в текущем каталоге, причем обязательно с явно присутствующим в тексте расширением EXE. Если при этом дополнительно присутствует и ненулевой параметр *pNameModule*, то текст последнего используется как последовательность аргументов для командной строки. При нулевом *pNameModule* параметр *CommandLine*, задаваемый указателем *pCommandLine*, кроме имени программы задает после него и аргументы командной строки. Таким образом, параметр *pCommandLine* дает больше возможностей, но использование одного параметра *pNameModule* – проще. Параметр *InheritFlag* используется для задания наследования объектов дочерним процессом и в нашем изложении использоваться не будет (полагается равным FALSE, т.е. равным 0). Параметр *pCurrDir* относится к дополнительным возможностям функции создания и задает новый текущий каталог для запускаемого процесса. В ближайших примерах он всегда будет задаваться нулевым указателем, т.е. его возможности не будут использоваться. Параметр *penv* задает указатель на блок памяти, хранящий строки *окружения*, в простейших и наиболее частых ситуациях он равен NULL.

Заметим, что кроме одной текстовой строки аргументов в ОС, начиная с Unix – родоначальницы всех современных ОС, представляется возможность применения так называемого *окружения* (environment) процесса. Окружение процесса (называемое также *средой* процесса) – это специальная текстовая область, служащая для передачи данных создаваемому процессу. В Unix она предназначена в первую очередь для передачи настроек программы от операционной системы или командной оболочки. В иных ОС окружение используется достаточно редко, но сама такая возможность оставлена. Чтобы не использовать передаваемую информацию для окружения, следует просто задать в качестве аргумента *env* нулевой указатель.

Параметр *CreateFlag* задает флаги создания и, кроме достаточно тонких нюансов, определяет класс *приоритета* создаваемого процесса. Обычное значение этого флага задается константой NORMAL_PRIORITY_CLASS.

Два последних параметра задают адреса двух специальных структур данных при создании процесса. Структура STARTINFO соответствует структуре описания сессии (сеанса) в OS/2 и содержит 18 полей, определяющих в первую очередь характеристики окна для создаваемого процесса. В простейших случаях все поля могут быть заполнены нулями, кроме поля с именем *cb*, в котором должна быть записана длина экземпляра этой структуры в байтах.

Структура PROCESS_INFORMATION, задаваемая адресом в последнем параметре, содержит четыре поля для возврата учетной информации из ОС после создания нового процесса. Описывается она в заголовочном файле как

typedef struct

```
{HANDLE hProcess;
 HANDLE hThread;
 DWORD dwProcessId;
 DWORD dwThreadId;
} PROCESS_INFORMATION;
```

Наиболее значимыми в ней является поле *hProcess*, возвращающее хэндл созданного процесса, и поле *dwProcessId*, возвращающее идентификатор (уникальный номер) созданного процесса.

Возвращаемое функцией CreateProcess значение типа BOOL позволяет вызывающей программе решить, удачно ли прошел запуск – удалось ли создать процесс. При успешном создании возвращается значение TRUE, иначе FALSE, численно равное нулю.

Следующий пример, представляемый программами в листингах 7.2.2a и 7.2.2b, демонстрирует простейшее использование функции создания процесса в Windows. Получаемые из этих исходных текстов программы должны обязательно строиться как консольные приложения, в частности, в интегрированной системе разработки Borland/Inprise языка C выбором опции Console.

```
#include <windows.h>
#include <stdio.h>
int main()
{int k;
 DWORD rc;
 STARTUPINFO si;
 PROCESS_INFORMATION pi;

        printf("Demonstration processes, Main Proccess\n");
        memset(&si, 0, sizeof(STARTUPINFO));   si.cb=sizeof(si);
        rc=CreateProcess(NULL, "child1.exe", NULL, NULL, FALSE,
                         NORMAL_PRIORITY_CLASS, NULL, NULL, &si, &pi);
        if (!rc)
         {printf("Error create Process, codeError = %ld\n", GetLastError());
           getchar(); return 0;  }
        printf("For Child Process:\n");
        printf("hProcess=%d  hThread=%d  ProcessId=%ld  ThreadId=%ld\n",
               pi.hProcess, pi.hThread, pi.dwProcessId, pi.dwThreadId);
        for (k=0; k<10; k++)
           {printf("I am Parent... (my K=%d)\n", k); Sleep(2000);   }
        CloseHandle(pi.hProcess);    CloseHandle(pi.hThread);
        return 0;
```

```
}
```

Листинг 7.2.2a. Программа Parent.c для Windows

```
// Демонстрация порождения процессов. Child Process.
#include <stdio.h>
#include <windows.h>
int main()
{int k;
        printf("Demonstration processes, Child Proccess\n");
        for (k=0; k<30; k++)
           {printf("I am Child ... (k=%d)\n\r", k);    Sleep(1000);   }
        return 0;
}
```

Листинг 7.2.2b. Программа Child1.c для Windows

В программе описаны переменные *si, pi* типов STARTUPINFO и PROCESS_INFORMATION, причем в поле *cb* структуры *si* записана длина переменной *si* (делать это надо обязательно перед использованием такого аргумента в вызове функции CreateProcess), а остальные поля этой структуры обнулены обращением к функции

memset(&si, 0, sizeof(STARTUPINFO))

Последняя записывает в память, начало которой задано первым аргументом, столько значений, заданных вторым аргументом, сколько указано числом в третьем аргументе. Значение параметра *CreationFlags* задает нормальный класс приоритета. Вызов функции создания процесса в качестве ненулевых содержит только параметры *pCommandLine*, *CreateFlag*, *pstartinfo*, *pProcInfo*, в качестве которых использованы "child1.exe", NORMAL_PRIORITY_CLASS, &si, &pi. После вызова функции ее код возврата в переменной *rc* используется для анализа ситуации. При *rc,* равной нулю, выдается сообщение об ошибке запуска, код которой извлекается системной функцией GetLastError(), и процесс завершается оператором return 0.

При отсутствии ошибки на экран выдается информация о запущенном процессе путем вывода значений полей *hProcess, hThread, dwProcessId, dwThreadId* возвращаемой переменной *pi.* Далее 10 раз с приостановкой на 2 с. выдается сообщение о шаге работы процесса-родителя. Приостановка выполняется функцией Sleep(), аргумент которой задается в миллисекундах. В конце программы полученные при создании процесса хэндлы закрываются вызовом функции CloseHandle(). Для понимания следует обратить внимание, что в данном примере закрываются не только хэндлы, но и связанные с ними управляющие блоки (структуры описания процесса и нити). (Это имеет место, когда на управляющие блоки указывает только один хэндл, в общем случае дублирования хэндлов ситуация может быть иной.)

Функция execl(), использованная в примере порождения дочернего процесса для Unix, является на самом деле лишь одним из представителей довольно широкого семейства функций с общим префиксом *exec*. В это семейство входят функции execl(), execlp(), execle(), execv(), execvp(), execve(). Несмотря на обилие этих функций, их действия идентичны, а отличаются они только наборами аргументов. Все перечисленные функции предназначены для замены текущей выполняемой программы на новую исполняемую программу, имя которой задается первым аргументом вызова всех перечисленных функций.

Наличие и предназначение следующих в списке аргументов условно обозначается в именах рассматриваемых функций с помощью одной из дополнительных букв. Символ *l* в составе имени функции задает использование переменного числа аргументов, символ *v* указывает, что список аргументов командной строки задается массивом указателей на текстовые строки. (Поэтому символы *l* и v в имени рассматриваемых функций являются альтернативными и исключают друг друга.) При наличии символа *p* в составе имени функции, последняя ищет указанный при вызове исполняемый файл во всех каталогах, заданных переменной среды PATH, все другие функции используют только явное указание имени функции. (Функция с символом *p* в имени по списку аргументов не отличается от соответствующей ей функции без этого символа.) При наличии в имени функции символа *e* в качестве дополнительной информации в списке аргументов присутствует аргумент – массив переменных окружения.

В частности, прототипом функции execve() является

int execve(char *filename, char *argv[ ], char *env[ ]),

(которая и служит наиболее общей исходной системной функцией), а прототип функции execlp() описывается как

int execlp(char *filename, char *arg1, char *arg2, . . . , char *argn, NULL),

где последний аргумент нулевым указателем задает конец списка аргументов.

(Заметим, что использование перечисленных функций с параметром argv[ ] позволяет программисту "обмануть" вызываемую программу, задав в качестве нулевого элемента списка аргументов текстовое наименование, отличное от имени запускаемой программы.) При неудачном запуске все рассматриваемые функции возвращают значение -1. (При их успешном запуске некому воспользоваться возвращаемым значением, так как после этого запуска будет выполняться совсем другая программа.)

В операционных системах Windows аналогичный по назначению набор функций имеет названия, начинающиеся с префикса spawn. Набор этот состоит из функций spawnl(), spawnle(), spawnlp(), spawnlpe(), spawnv(), spawnve(), spawnvp(), spawnvpe(), где первая и последняя функция из набора задается прототипами

int spawnl(int mode, char *path, char *arg0, arg1, ..., argn, NULL),

int spawnvpe(int mode, char *path, char *argv[ ], char *envp[ ]).

Остальные функции имеют аналогичное строение и используют вспомогательные символы *l*, *e*, *p* и *v* по тому же соглашению, что и функции Unix. Первый аргумент *mode* всех рассматриваемых функций задает действия родительского процесса после создания дочернего. Если этот параметр задается символической константой P_WAIT, то родительский процесс приостанавливается до тех пор, пока не завершится запущенный дочерний процесс. При задании первого аргумента символической константой P_NOWAIT родительский процесс продолжается одновременно с дочерним, обеспечивая тем самым параллельное выполнение обоих процессов. Заметим, что функции набора spawn не входят в стандарт языка С, а специфичны для указанных операционных систем.

7.3. Уничтожение процессов

Процесс не просто понятие для изучения, а конкретный информационный объект в операционной системе. Над информационными объектами, находящимися в компьютере, естественными оказываются операции, которые осуществляют действия целиком над каждым таким объектом. Для вычислительных процессов оказалось возможным выполнять следующие операции: создание, уничтожение и ожидание завершения. Эти операции далее рассматриваются более детально. Вначале рассмотрим смысловую альтернативу создания процесса – операцию уничтожения имеющегося процесса.

При построении приложений на основе нескольких процессов может возникнуть необходимость принудительно и окончательно прекратить функционирование некоторого другого процесса. Процесс-родитель может уничтожить процесс, который он создал, используя соответствующий системный вызов и идентификатор этого процесса. Отсюда становится понятным простейшее предназначение идентификатора процесса: если родительским процессом создано несколько дочерних процессов, то, используя такой идентификатор, можно задать уничтожение любого из этих дочерних процессов. Для уничтожения процессов-потомков в OS/2 служила системная функция, описываемая прототипом

APIRET DosKillProcess(ULONG action, PID pid).

Ее первый параметр задавал, следует ли уничтожать только одного потомка – дочерний процесс, или же следует уничтожить и всех последующих потомков указанного дочернего. В первом случае использовалось значение 1, во втором 0 или же символические константы DKP_PROCESS и DKP_PROCESSTREE. Второй параметр вызова определял идентификатор дочернего процесса, являющегося объектом уничтожения.

Особенно велика потребность в функции уничтожения процессов внутри Unix, где права пользователя-администратора настолько широки, что ему следует позволить уничтожать любые процессы. В Unix для уничтожения процессов пред-

назначена функция kill(), возможности которой даже несколько шире, чем только уничтожение процесса. Непосредственно для уничтожения процесса ее употребление имеет вид

kill(pid, SIGKILL)

первый аргумент задает идентификатор процесса, полученный в свое время родительским процессом после выполнения функции *fork()*, второй является символической константой, определенной в заголовочном файле для посылки приказа именно уничтожения, а ни чего-то другого. (Эта же функция может использоваться и для временной приостановки процесса по воздействию со стороны и т.п.).

Пример использования описанных системных функций в Unix задается программами листингов 7.3.1a, 7.3.1b и 7.3.1c.

```
#include <stdio.h>
#include <sys/types.h>
#include <signal.h>
int main()
{int rc, k;

  printf("Parent Proccess\n");
  rc=fork();
  if (!rc)     {execl("child1.exe",0);   }
  printf("For Child Process:\n");    printf("PID=%d \n", rc);
  for (k=0; k<10; k++)
     {printf("I am Parent (My K=%d)\n", k);
      if (k= =6)
         {kill(rc, SIGKILL);  printf("I am killing my child with PID=%d\n",rc);}
      usleep(2000000);
     }
  printf("Parent ended\n");   exit(0);
}
```

Листинг 7.3.1a. Программа procest.c для Unix

```
#include <stdio.h>
int main()
{int rc, k;
  printf("First Child Proccess\n");
  rc=fork();
  if (!rc)     {execl("child2.exe",0);  }
  printf("For Second Child Process:\n");    printf("PID=%d \n", rc);
  for (k=0; k<30; k++)
```

```
    {printf("I am First Child (My K=%d)\n", k); usleep(750000);  }
  printf("First Child ended\n");
  exit(0);
}
```

Листинг 7.3.1b. Программа child1.c для Unix

```
#include <stdio.h>
int main()
{int k;
 printf("Demonstration processes, Child 2 Proccess\n");
  for (k=0; k<30; k++)
    {printf(" ------- I am second Child ... (child2=%d)\n", k);  sleep(1);   }
  printf("Child 2 ended\n");
  exit(0);
}
```

Листинг 7.3.1c. Программа child2.c для Unix

Для работающего примера в Unix необходимо создать три исполняемых файла procest.exe, child1.exe и child2.exe, изготовляемых из исходных текстов в листингах 7.3.1a, 7.3.1b, 7.3.1c. Программа procest.c описывает создание еще одного процесса из исполняемого файла child1.exe, который будет дочерним процессом для исходного процесса по программе procest.c, а процесс child2.exe, автоматически запускаемый им на основе программы в листинге 7.3.1c, оказывается процессом-внуком для родительского. Программа в листинге 7.3.1a – после создания процесса и циклической выдачи сообщений на 6-м шаге – приказывает оператором

```
kill(rc, SIGKILL)
```

уничтожить дочерний процесс. В результате этого приказа процесс по программе child1.c прекращается, но все еще продолжается процесс по программе child2.c в листинге 7.3.1c, пока он не завершит все действия, предусмотренные в ней. Заметим, что для использования в программе для Unix системных вызовов kill() необходимо включение в исходную программу на языке Си заголовочных файлов *types.h* и *signal.h*, первый из которых размещается в подкаталоге *sys* основного каталога для хранения заголовочных файлов системы программирования на языке Си.

Для уничтожения процессов в операционных системах Windows предназначена функция TerminateProcess(), которая имеет прототип

```
BOOL  TerminateProcess(HANDLE hProcess, UINT uExitCode)
```

и в котором аргумент *hProcess* задает какой именно процесс следует уничтожить, а аргумент *uExitCode* – код завершения для такого принудительного завершения. Нормальный (не принудительный) код завершения задает функция

VOID ExitProcess(UINT uExitCode),

которую следует ставить последней в действиях программы процесса, или вместо нее использовать мобильную (независимую от операционной системы) функцию exit(*код*), имеющую один аргумент – код завершения.

Программы для Windows, выполняющие те же задачи, что и рассмотренные выше программы из листингов 7.3.1a, 7.3.1b и 7.3.1c для Unix, предлагаются читателю в качестве упр. 1 в конце данной главы.

7.4. Ожидание завершения процессов

Последней из операций, которую "удалось придумать" для информационных объектов-процессов, является *ожидание завершения процесса*. Следует обратить внимание, что эта операция в действительности чрезвычайно ценная, так как обеспечивает гарантированное использование результатов процесса родительским процессом.

Из содержательных соображений непосредственно вытекает, что такая операция должна задаваться информацией вида

ждать_завершения_процесса(какой_процесс_ждать),

кроме того, в вызов этой функции может быть включен возвращаемый параметр, который позволяет получить код завершения процесса, обычно формируемый функцией exit(*код_возврата*).

Наиболее мощные, но и сложные средства ожидания завершения процесса использованы в OS/2. Во-первых, процесс, завершение которого предполагается не только ожидать, но и получить от него значение кода завершения, должен запускаться функцией создания процесса со специальным значением параметра флагов, а именно этот параметр должен задаваться символической константой EXEC_ASYNCRESULT. Если требуется ожидание только самого момента завершения, то достаточно использовать константу EXEC_ASYNC.

Для задания собственно ожидания завершения процесса в этой ОС предназначена функция с прототипом

APIRET DosWaitChild(ULONG action, ULONG option,
RESULTCODES *pres, PID *ppid, PID pid),

где параметр *action* задает чего именно требуется ожидать – завершения лишь одного процесса (задается константой DCWA_PROCESS) или завершения всей совокупности процессов-потомков, начиная с указанного процесса (задается константой DCWA_PROCESSTREE). Параметр *option* определяет режим выполнения данной функции и может задаваться константами DCWW_WAIT и DCWW_NOWAIT. Первая из них задает действительное ожидание, а вторая требует опросить состояние завершения и немедленно вернуть информацию об этом, не блокируя выполнение текущего процесса. Последний параметр *pid* задает идентификатор процес-

са, завершение которого требуется ожидать (или всех его потомков при задании соответствующего параметра *action*). Если значение этого параметра *pid* задается равным нулю, то это является указанием ожидать завершения любого дочернего процесса. Возвращаемый параметр *ppid* используется для получения значения идентификатора того процесса-потомка, который завершился последним при ожидании завершения всех потомков некоторого дочернего процесса или того дочернего процесса, который завершился после вызова функции с параметром pid, равным нулю.

Значение кода возврата, возвращаемое от завершенного процесса после выполнения рассматриваемой функции, находится в поле *codeResult* возвращаемого экземпляра *pres* структуры RESULTCODES. При этом поле *codeTerminate* этого возвращаемого параметра дает информацию о качественной причине завершения процесса. Эти причины описываются символическими константами и отображают как нормальное завершение, так и завершение по причине уничтожения другим процессом.

В операционной системе Unix имеется ранний вариант функции ожидания wait() с прототипом

pid_t wait(int *status).

Эта функция переводит текущий процесс в состояние ожидания до тех пор, пока не завершится какой-то из его дочерних процессов. Код возврата этого дочернего процесса получается функцией посредством параметра *status*. Если было запущено несколько дочерних процессов и необходимо дождаться завершения всех их, следует сделать соответствующее число вызовов функцией wait(), в частности, записав ее несколько раз подряд. Практически этой функцией удобно пользоваться при работе только с одним дочерним процессом. Поэтому в более поздних версиях Unix появилась дополнительная функция ожидания

pid_t waitpid(pid_t pid, int *status, int option),

которая позволяет задать с помощью значения *pid* завершение какого именно процесса будет ждать данный процесс, выполняющий функцию waitpid(), а параметром *option* - блокирующим или нет будет вызов функции. Использование этой функции требует включение заголовочного файла директивой *#include <sys/wait.h>*. Использование константы WNOHANG из этого файла в качестве параметра *option* обеспечивает немедленное возвращение функцией waitpid значения 0, если запращиваемый процесс еще не завершен. При нулевом значении этого параметра обеспечивается ожидание завершения процесса.

Следующая программа, приведенная в листинге 7.4.1, является исходным текстом для исполняемого файла procesw.exe, предназначенного для Unix. Для ее использования необходимо в автоматически доступном каталоге (в частности текущем) иметь исполняемый файл child1.exe, созданной из исходной программы, приведенной в листинге 7.3.1b.

```
#include <stdio.h>
#include <sys/types.h>
#include <signal.h>
int main()
{int rc, k;

  printf("Parent Proccess\n");
  rc=fork();
  if (!rc)     {execl("child1.exe",0);   }
  printf("For Child Process: PID=%d \n", rc);
  wait(&rc);
  printf("I have waited my child with retcode =%d\n",rc);
  for (k=0; k<10; k++)
     {printf("I am Parent (My K=%d)\n", k);
      usleep(2000000);
     }
  printf("Parent ended\n");   exit(0);
}
```

Листинг 7.4.1. Программа procesw.c для Unix

В этой программе – после создания дочернего процесса и выдачи учетной информации о нем (идентификатора процесса PID) – вызывается функция wait() для ожидания завершения дочернего процесса. Наблюдение за поведением процессов, образующихся при запуске этой программы, позволяет явно видеть, что после выполнения wait() и до завершения дочернего процесса родительский приостановлен и продолжает работу только после завершения дочернего.

В операционных системах Windows для ожидания завершения процесса применяется универсальная функция ожидания WaitForSingleObject, имеющая прототип

DWORD WaitForSingleObject(HANDLE hProcess, DWORD Timeout).

Здесь первый параметр задает хэндл того процесса, завершения которого следует ждать, а второй параметр – максимальное время ожидания в миллисекундах. Если величину *Timeout* задать символической константой INFINITE, то ожидание неограниченно. В других случаях после истечения времени, заданного в *Timeout*, функция возвращает управление со значением самой функции, равным символической константе WAIT_TIMEOUT, по которой программа может определить, что ожидание было неудачным.

При удачном завершении ожидания можно получить код завершения дочернего процесса. Для его получения служит функция GetExitCodeProcess() с прототипом

BOOL GetExitCodeProcess(HANDLE hProcess, LPDWORD pExitCode).

Следующая программа, приведенная в листинге 7.4.2, является исходным текстом для исполняемого файла Procesw.exe, предназначенного для Windows. Для ее использования необходимо в автоматически доступном каталоге (в частности текущем) иметь исполняемый файл Child1.exe, действия которого аналогичны программе из листинга 7.3.1b, но предназначеного для Windows.

```
#include <windows.h>
#include <stdio.h>
int main()
{int k;
 DWORD rc;
 DWORD CreationFlags;
 STARTUPINFO si;
 PROCESS_INFORMATION pi;
        printf("Demonstration processes, Main Proccess\n");
        memset(&si, 0, sizeof(STARTUPINFO));   si.cb=sizeof(si);
        CreationFlags = NORMAL_PRIORITY_CLASS;
        rc=CreateProcess(NULL, "child1.exe", NULL, NULL, FALSE,
                    CreationFlags, NULL, NULL, &si, &pi);
        if (!rc)
         {printf("Error create Process, codeError = %ld\n", GetLastError());
          getchar(); return 0;  }
        printf("For Child Process:\n");
        printf("hProcess=%d  hThread=%d  ProcessId=%lu  ThreadId=%lu\n",
              pi.hProcess, pi.hThread, pi.dwProcessId, pi.dwThreadId);
        WaitForSingleObject(pi.hProcess, INFINITE);
        printf("It terminated child with hProcess\n", pi.hProcess);
        GetExitCodeProcess(pi.hProcess, &rc);
        printf("ExitCodeProcess=%ld\n", rc);
        for (k=0; k<10; k++)
           {printf("I am Parent... (my K=%d)\n", k); Sleep(2000);   }
        CloseHandle(pi.hProcess);    CloseHandle(pi.hThread);
        return 0;
}
```

Листинг 7.4.2. Программа Parentw.c для Windows

В этой программе – после создания дочернего процесса и выдачи учетной информации о нем (идентификатора в *pi.hProcess*) – вызывается функцию WaitForSingleObject(pi.hProcess, INFINITE) для ожидания завершения дочернего процесса, сколько бы долго он не длился. Получение информации о завершении процесса выполняется функцией GetExitCodeProcess(pi.hProcess, &rc).

Наблюдение за поведением процессов, образующихся при запуске этой программы, позволяет явно видеть, что после задания ожидания и до завершения дочернего процесса родительский приостановлен, он продолжает работу только после завершения дочернего.

Упражнения

1. Разработать программы для Windows, выполняющие те же задачи, что и рассмотренные выше программы в листингах 7.3.1a, 7.3.1b, 7.3.1c для Unix. Аналогично рассмотренным примерам, при выполнении упражнения необходимо создать три исполняемых файла Procest.exe, Child1.exe и Child2.exe. Программа Child1.c должна описывать создание (из исполняемого файла child2.exe) процесса, который будет процессом-внуком для исходного процесса.

8. МНОГОПОТОЧНОЕ ФУНКЦИОНИРОВАНИЕ ОС

8.1. Понятие нити и связь ее с процессом

В современных операционных системах широко используются *нити* (thread), называемые несколько неточно в русском переводе также *потоками*. Это понятие возникло как результат развития понятия абстрактного процесса. Оказалось, что иногда целесообразно процесс – как некоторую общность программного выполнения, имеющую прикладную цель, – разбить на части, которые выполнялись бы параллельно, конкурируя за главный ресурс – процессор, но в остальном выполняли бы общую работу. Можно подойти к осознанию понятия процесса с другой стороны. Абстрактные процессы теоретически разделяют на конкурирующие и кооперативные. Конкурирующие процессы по существу мешают друг другу, но в совокупности выполняют много работ одновременно. Кооперативные процессы выполняют по частям общую работу совместно. Организационно-техническое объединение (под одной "крышей" обобщенного процесса) аналогов кооперативных процессов и составляет существо объединения нитей в одном процессе.

Процесс в современных ОС – это владелец всех основных ресурсов кооперативно работающих нитей. У этих нитей – общее виртуальное адресное пространство, у них общие дескрипторы (управляющие блоки – описатели внутри служебных областей ОС) и соответствующие им хэндлы, причем дескрипторы и хэндлы всевозможных допустимых в ОС объектов: файлов, созданных дочерних процессов, графических окон, семафоров и т.п.). Практически нити одного процес-

са используют глобальные переменные, в которые каждая из нитей может записывать данные или брать их оттуда. Нити пользуются процессором компьютера по очереди, так что он не принадлежит ни одной из них, но и не принадлежит самому процессу.

Чем же владеет отдельная нить? Нити принадлежит только текущая совокупность значений в аппаратных узлах хранения информации (регистрах) и локальные переменные подпрограмм. Текущую совокупность значений в аппаратных узлах хранения информации называют *контекстом задачи*. Когда происходит переключение между задачами (нитями), запоминается контекст приостанавливаемой задачи и восстанавливается из места сохранения контекст запускаемой на процессор задачи. В современной архитектуре процессоров много внимания уделяется аппаратной реализации сохранения контекста, его восстановлению и переключению между задачами. Зачем необходимо запоминать и восстанавливать контекст задачи? По той же причине, по которой сохраняется и восстанавливается содержимое аппаратных узлов при возникновении прерывания.

Обычно нить своей работой реализует действия одной из процедур программы. Теоретически любой нити процесса доступны все части программы процесса, в частности, все его процедуры, но реально работа организуется так, чтобы нити отвечала отдельная процедура. Учитывая, что процедуре для нормальной работы необходимы локальные переменные, становится понятным закрепление области этих переменных за нитью. Объект хранения локальных переменных (вместе со служебной информацией при вызове подпрограмм) называют *стеком*. (Более точное понятие стека программы формируется только с помощью архитектуры процессора.) Этот стек в действительности является частью оперативной памяти, он используется не только программно, но и аппаратно, в частности, при реализации прерываний. Стек процедуры является неотъемлемой частью ресурсов, принадлежащих процедуре. (Более точным термином является *кадр стека* для процедуры или *фрейм стека*.)

Очень описательно, но не очень точно соотношение между процессом и нитью можно представить аналогией между собственником земли и его наемными работниками. Как минимум, собственник выполняет организаторскую и контролирующую работу (главная нить процесса), но для более эффективного использования ресурсов привлекает еще наемных работников (остальные нити в процессе).

При разработке многопоточных приложений следует иметь в виду необходимость явного указания для системы разработки, что данное приложение будет использовать более одной нити. Для таких приложений в процессе построения исполняемого файла подключаются специальные библиотеки подпрограмм.

Для задания многопоточного приложения как результата разработки в системах программирования Borland Inc. (Inprise Inc.) при использовании командного вызова компилятора следует обязательно использовать опцию -tWM, так что вы-

зов компилятора для построения исполняемой программы из файла prog.c будет записываться в виде

BCC32 -tWM prog.c

Для компиляции исходного файла в системе MS Visual C++ с помощью компилятора CL.EXE этого пакета следует использовать опцию /MT, так что аналогичный вызов компилятора будет иметь вид

CL /MT prog.c

При использовании интегрированных систем разработки нужно учитывать необходимость установки флажка многопоточного приложения (флажка Multithread) в соответствующем диалоговом окне окна цели (target) разработки. В частности, в среде разработки Borland C++ этот флажок необходимо установить в окне TargetExpert с заголовком New Target, возникающим после выбора подпункта New в пункте Project меню, выпадающего из пункта File главного меню.

При разработке многопоточных программ для Linux следует указывать соответствующую библиотеку поддержки. Такое указание может задаваться в одной из двух основных форм. Первая из них явно задает библиотеку и имеет вид

gcc prog.c /usr/lib/libpthread

а вторая задает эту же библиотеку неявно и записывается в виде

gcc prog.c -lpthread

Естественно, что при этом могут быть использованы и другие опции вызова компилятора, в частности, явное именование результирующего исполняемого файла и добавление отладочной информации.

8.2. Создание нитей (thread) в программе

Главная нить процесса создается автоматически при создании процесса. Если процесс нуждается в дополнительных нитях, то его программа вызывает системные функции создания нити.

В операционной системе OS/2 для создания нити служит функция DosCreateThread, которая имеет прототип

APIRET DosCreateThread(PTID ptid, PFNTHREAD pfn,
ULONG param, ULONG flag, ULONG cbStack),

где *ptid* – указатель на возвращаемый номер (идентификатор) нити, *pfn* – имя подпрограммы (описанной выше в программе прототипом), *param* – передаваемый в процедуру параметр, *cbStack* - размер стека для нити. Аргумент *flag* задает, запускается ли в работу нить сразу после создания или создается приостановленной. В первом случае значение *flag* = 0, во втором *flag* = 1. Эти значения могут быть указаны символическими константами CREATE_READY и CREATE_SUSPENDED. Тип PFNTHREAD определяет процедуру нити как имеющую тип VOID с

единственным аргументом типа ULONG. Используя явное преобразование типов, можно задать при создании нити использование любого вида процедуры.

В операционной системе Unix многопоточное программирование появилось достаточно поздно. К настоящему времени эта возможность входит в стандарт POSIX для Unix и поддерживается во всех современных ОС. Использование нитей при этом требует подключения заголовочного файла pthread.h. Системная функция создания нити в Unix по указанному стандарту имеет прототип

int pthread_create(pthread_t* tid, const pthread_attr_t* att,
void*(*fun)(void*), void* argp).

Первый параметр этой функции возвращает идентификатор созданной нити. (В качестве дополнительных возможностей предлагается задавать в этот параметр исходное значение NULL, тогда идентификатор не возвращается). Важнейшим и обязательным параметром является адрес процедуры нити, этот параметр стоит третьим в списке и ему должна отвечать процедура с возвращаемым типом *void** и единственным аргументом типа *void**. При ином построении процедуры (что практически не имеет смысла, если эта процедура используется только для работы нити) требуется указывать явное преобразование типа. Последний, четвертый параметр, предназначен для передачи конкретного аргумента при создании нити. Второй параметр вызова *attr* – для задания атрибутов, отличных от атрибутов по умолчанию, причем параметр является указателем на место объекта, содержащего эти атрибуты. В простейшем и многих иных случаях этот параметр задается как NULL, что информирует ОС о необходимости использовать атрибуты по умолчанию. (О более сложном использовании этого параметра здесь рассказываться не будет.) Если функция создания нити возвращает значение >0, то в ходе выполнения системной функции произошла ошибка, и нить не была создана. Эту ситуацию рекомендуется анализировать в программе. Нулевое возвращаемое значение соответствует безошибочному выполнению, а положительное значение выдает непосредственно числовое значение кода ошибки.

Действия процедуры нити могут завершаться специальной функцией pthread_exit(). Эта функция имеет прототип

int pthread_exit(void *status),

где аргумент функции является указателем на значение любого удобного программисту типа данных. Указанное значение предназначено для выдачи кода завершения из завершающейся нити. Этот код может быть получен и проанализирован специальной функцией ожидания завершения нити, которая будет рассмотрена в этой же главе. Если предполагается не использовать код завершения нити, то аргумент функции pthread_exit может задаваться как нулевой указатель или просто как нулевое значение. Указанная функция требуется только, если эта нить должна передавать код завершения или при задании завершения внутри программы процедуры. На конечном месте процедуры (перед закрывающей фигурной скобкой тела

процедуры нити) вызов функции завершения можно опускать. (Практически она автоматически вызывается после возврата из процедуры нити, когда последняя завершается как обычная процедура.)

Следующая программа, приведенная в листинге 8.2.1, демонстрирует порождение и использование нитей.

```
#include <pthread.h>
#include <stdio.h>
char lbuk[ ]="abcdefghijklmnoprstvuwxy";
pthread_t tid1, tid2, tid3;

void procthread1(void *arg)
{int k, j;
  for (k=0; k<24; k++)
    {printf("\033[%d;20H\033[1;34m",k+1);
     for (j=0; j<(int)arg; j++)      printf("%c",lbuk[k]);
     printf("\n");      usleep(1000000);
    }
}

void procthread2(void *arg)
{int k, j;

  for (k=0; k<24; k++)
    {printf("\033[%d;40H\033[1;32m",k+1);
     for (j=0; j<(int)arg; j++)      printf("%c",lbuk[k]);
     printf("\n");      usleep(1000000);
    }
}

void procthread3(void *arg)
{int k, j;
  for (k=0; k<24; k++)
    {printf("\033[%d;60H\033[31m",k+1);
     for (j=0; j<(int)arg; j++)      printf("%c",lbuk[k]);
     printf("\n");       usleep(1000000);
    }
}

void main()
```

```
{int k;
 int rc;
  printf("\033[2J\n");
   rc=pthread_create(&tid1, NULL, (void*)procthread1, (void*)2);
   rc=pthread_create(&tid2, NULL, (void*)procthread2, (void*)3);
   rc=pthread_create(&tid3, NULL, (void*)procthread3, (void*)4);
   for (k=0; k<24; k++)
     {printf("\033[%d;1H\033[37m",k+1);
      printf("%c++\n",lbuk[k]);
      usleep(1500000);
     }
   getchar(); printf("\033[0m");
}
```

Листинг 8.2.1. Программа с многими нитями для Unix.

В этой программе описаны три процедуры с именами *procthread1*, *procthread2*, *procthread3*. Они имеют тип результата void и единственный аргумент типа *void**. Главная часть программы запускает три нити, используя эти процедуры как основы их. Четвертый аргумент функции pthread_create передает в процедуры параметр, который в примере для всех вызовов является числовым (числом 2, 3, 4), что соответствует описанию аргументов в прототипе.

В каждой процедуре осуществляется многократный вызов (на основе цикла с повторением 24 раза) функции вывода на экран. Причем последовательно выводятся символы, описанные в символьном массиве *lbuk*, общем для всех процедур. Каждая процедура выводит символы своим цветом (первая – ярко-синим, вторая – ярко-зеленым, третья – красным), а начиная со своей колонки экрана (соответственно с 20-й, 40-й и 60-й). Каждый шаг во внешнем цикле процедуры выводит символы в строку с номером, на единицу большим номера цикла (отсчет строк ведется с единицы, номера шага переменной *k* – с нуля). Все процедуры выводят подряд столько одинаковых символов, сколько задает параметр, передаваемый процедуре. Здесь же демонстрируется преобразование общей формы передачи параметра, описанного в прототипе как *void**, в конкретный тип, нужный в месте использования. Для этого аргумент в месте использования записан как *(int)arg*. Для удобства наблюдений используется функция приостановки usleep, использование которой позволяет наглядно воспринимать происходящее в процессе и видеть как по очереди работают отдельные нити.

В главной части программы также выполняется циклический вывод на экран символов из того же массива *lbuk*, но только уже в белом цвете. Причем символы выводятся по одному в первую колонку экрана. Для упрощения программы код возврата функций при создании нитей в этой программе не анализируется (такое

упрощение не следует применять в реальных программах). Особенностью демонстрационной программы для операционной системы Unix является необходимость дополнительных выводов управляющих символов "перевода строки" (в обозначениях Си '\n'). Причиной этого служит встроенная в реализацию Linux внутренняя оптимизация вывода на экран консоли. Этот вывод не отображается на экране, пока вывод строки не завершен. Практически вывод идет во внутренний буфер, но на пользовательском экране появляется после передачи указанного управляющего символа. Следствием такого механизма является включение в программу отдельных операторов printf("\n"). Обучаемому рекомендуется, удалив такие операторы, пронаблюдать поведение полученной программы. (Иным, но не лучшим решением является применение не стандартного вывода, неявно используемого функцией printf(), а стандартного вывода ошибок вместе с использованием функции fprintf() в виде fprintf(stderr, "*текст_формата*", *аргументы*)).

Наиболее сложная ситуация оказывается с порождением нитей в операционных системах Windows. И дело все в том, что для порождения нитей в этих системах имеется не одна функция, а несколько. Самой ранней является системная функция CreateThread, но в ее реализации при использовании на языке программирования Си позже были обнаружены некоторые проблемы некорректного поведения и было предложено заменить ее функцией _beginthread. Еще позже непредусмотренное поведение было обнаружено (в редких, но возможных ситуациях) и у этой функции, в связи с чем разработчики систем программирования на языке Си ввели в использование новый вариант последней функции. Особо отметим, что проблемы использования функции создания возникают при написании программ именно на языке Си, который в свое время создавался в неявном предположении единственной нити в программе (как потом оказалось!).

В системах программирования фирмы Microsoft, называемых традиционно Visual C++, модифицированная функция создания нитей получила название _beginthreadex. В системах программирования фирмы Borland/Inprise Inc. модифицированную функцию назвали _beginthreadNT. Все эти перечисленные функции отличаются по порядку, а частично и по набору своих аргументов. Введение в использование новых функций для порождения нитей имело целью сохранить старые программы с их сложившимся поведением и одновременно исправить ошибки и обеспечить более совершенное поведение для новых программ. Основная нагрузка принятия решений при программировании и ответственности за эти решения переносилась с разработчиков системных компонентов Windows на далеких от них прикладных разработчиков-программистов.

К настоящему времени наиболее безопасными функциями порождения нитей являются _beginthreadNT и _beginthreadex, именно их и рассмотрим в первую очередь, предлагая для ближайшего использования.

Функция _beginthreadex от Microsoft имеет прототип

unsigned long _beginthreadex(void* security_attrib, unsigned stack_size,
unsigned (*proc)(void*), void *arglist, unsigned flag,
unsigned *tid).

Функция _beginthreadNT имеет прототип

unsigned long _beginthreadNT(void (*proc)(void *), unsigned stack_size,
void *arglist, void *security_attrib,
unsigned long flags, unsigned long *tid).

Параметр *proc*, записанный в форме определения подпрограммы, задает адрес процедуры нити, в стандартном случае эта процедура должна быть функцией без возвращаемого значения (формально – с возвращаемым значением типа *void*) и нетипизированным аргументом (формально типа *void**). Аргумент *stack_size* функций создания нити указывает размер стека, практически он округляется до кратного величине 4096. Аргумент *arglist* задает указатель на строку аргументов, в простейшем случае отсутствия аргументов этот параметр может быть задан значением NULL. Аргумент *security_attrib* применяется в более сложных программах, использующих всю мощь средств встроенной защиты в Windows NT. В более простых случаях его можно всегда полагать равным NULL. Аргумент *flag* служит для задания режима приостановленной при создании нити, что обозначается символической константой CREATE_SUSPENDED. В противном случае (запуск тут же функционирующей нити) этот параметр-флаг устанавливается равным нулю. Последний параметр функций создания предназначен для возвращаемого значения идентификатора нити. Возвращает функция в случае неудачи число -1, в остальных случаях возвращаемое число – хэндл созданной нити, который как переменная должно быть описана с типом HANDLE. Заметим, что в системе разработки программ Watcom C++ используется вариант Microsoft функции создания нити.

Процедура нити, используемая функциями _beginthreadex и _beginthreadNT, может для завершения своих действий использовать системный вызов функции _endthreadex() или соответственно _endthread().

Исходной системной функцией для построения всех описанных функций создания нитей в Windows служит функция CreateThread, которой можно пользоваться в тех программах, процедуры задания нитей которых не содержат стандартных функций базовой библиотеки языка Си и неявно не используют их. Такая ситуация имеет место, когда процедура нити строится только на основе собственно Windows API – только этого программного интерфейса для ввода-вывода. Заметим, что именно этот вариант имеет место для большинства примеров в данном пособии. Функция CreateThread имеет прототип

HANDLE CreateThread(SECURITY_ATTRIBUTES *security_attrib,
DWORD stack_size, THREAD_START_ROUTINE *proc,
VOID *arglist, DWORD flags, DWORD * tid),

где все аргументы имеют те же наименования и то же назначение, что и в описанных выше функциях (хотя не всегда совпадающий тип). Для завершения процедуры, вызванной функцией, следует использовать системную функцию с прототипом

void ExitThread(DWORD dwExitCode),

если такое завершение требуется задать до "естественного" конца программы процедуры.

Следующий пример, представленный программой в листинге 8.2.2, демонстрирует построение и использование нитей в Windows. Эта программа аналогична рассмотренным ранее для других операционных систем и ведет себя аналогично. Отметим, что для точного согласования типов (чтобы не появлялись предупреждения о возможных ошибках) потребовалось задать явное преобразование типа для возвращаемого функцией значения, а именно в тип HANDLE.

```
#include <windows.h>
#include <process.h>
#include <stdio.h>
char lbuk[ ]="abcdefghijklmnoprqstuwxy";
HANDLE hstdout;
DWORD actlen;
CRITICAL_SECTION csec;

void procthread1(void *arg)
{int k, j;
 COORD pos;
   for (k=0; k<24; k++)     {
      pos.X=20; pos.Y=k+1;
      EnterCriticalSection(&csec);
      SetConsoleCursorPosition(hstdout,pos); // установить курсор в позицию (20,k+1)
      SetConsoleTextAttribute(hstdout,FOREGROUND_BLUE); // и синий цвет
      for (j=0; j<(int)arg; j++) printf("%c",lbuk[k]);
      LeaveCriticalSection(&csec);
      Sleep(800);
     }
}

void procthread2(void *arg)
{int k, j;
 COORD pos;
   for (k=0; k<24; k++)  {
```

```
    pos.X=40; pos.Y=k+1;
     EnterCriticalSection(&csec);
     SetConsoleCursorPosition(hstdout,pos); // установить курсор в позицию (40,k+1)
     SetConsoleTextAttribute(hstdout,FOREGROUND_GREEN); // и зеленый цвет
     for (j=0; j<(int)arg; j++)  printf("%c",lbuk[k]);
     LeaveCriticalSection(&csec);
     Sleep(1300);
    }
}

void procthread3(void *arg)
{int k, j;
 COORD pos;
  for (k=0; k<24; k++)     {
     pos.X=60; pos.Y=k+1;
     EnterCriticalSection(&csec);
     SetConsoleCursorPosition(hstdout,pos); // установить курсор в позицию (60,k+1)
    SetConsoleTextAttribute(hstdout,FOREGROUND_RED); // и красный цвет
     for (j=0; j<(int)arg; j++)  printf("%c",lbuk[k]);
     LeaveCriticalSection(&csec);
     Sleep(1100);
    }
}

void main()
{HANDLE hthread1, hthread2, hthread3;
 unsigned long threadid1, threadid2, threadid3;
 int k;
 COORD pos;
  hstdout=GetStdHandle(STD_OUTPUT_HANDLE);
  InitializeCriticalSection(&csec);
  hthread1=(HANDLE)_beginthreadNT(procthread1, 4096, (void *)2, NULL, 0,
       &threadid1);
  hthread2=(HANDLE)_beginthreadNT(procthread2, 4096, (void *)3, NULL, 0,
       &threadid2);
  hthread3=(HANDLE)_beginthreadNT(procthread3, 4096, (void *)4, NULL, 0,
       &threadid3);
  Sleep(600);
  for (k=0; k<24; k++)    {
    pos.X=1; pos.Y=k+1;
```

```
    EnterCriticalSection(&csec);
    SetConsoleCursorPosition(hstdout,pos); // установить курсор в позицию (10,k+1)
    SetConsoleTextAttribute(hstdout, // и белый цвет
       FOREGROUND_BLUE|FOREGROUND_GREEN|FOREGROUND_RED);
    printf("%c++",lbuk[k]);
    LeaveCriticalSection(&csec);
    Sleep(1000); }
  getchar();
  DeleteCriticalSection(&csec);
  CloseHandle(hthread1);   CloseHandle(hthread2);   CloseHandle(hthread3);
}
```

Листинг 8.2.2. Программа с многими нитями для Windows

В программах для Windows при выводе на консоль несколькими нитями возможно нежелательное взаимное влияние их друг на друга, выражающееся в искажении выводимой информации. Для принципиального устранения такого влияния можно использовать универсальное средство, называемое критическими интервалами. В данной программе это средство представлено описанием специальной общей для процедур переменной *csec* со структурным типом CRITICAL_SECTION и системными функциями, использующими адрес этой переменной, и называемыми InitializeCriticalSection, EnterCriticalSection, LeaveCriticalSection, DeleteCriticalSection. Они относятся к средствам организации взаимодействий и позволяют правильно согласовывать параллельную работу нескольких нитей. Подробно назначение и использование этих функций будут рассматриваться в п. 9.6.

Для среды разработки MS Visual C++ последняя программа должна будет содержать несколько иные вызовы функций создания нитей. Текст главной подпрограммы модификации последней программы для указанной среды приведен в листинге 8.2.3.

```
#define PFMSTHREAD unsigned ( __stdcall *)( void * )
void main()
{HANDLE hthread1, hthread2, hthread3;
 unsigned threadid1, threadid2, threadid3;
 int k;
 COORD pos;
 DWORD actlen;

  hstdout=GetStdHandle(STD_OUTPUT_HANDLE);
  InitializeCriticalSection(&csec);
  hthread1=(HANDLE)_beginthreadex(NULL, 4096,
```

```
        (PFMSTHREAD)procthread1, (void *)2, 0, &threadid1);
  hthread2=(HANDLE)_beginthreadex(NULL, 4096,
        (PFMSTHREAD)procthread2, (void *)3, 0, &threadid2);
  hthread3=(HANDLE)_beginthreadex(NULL, 4096,
        (PFMSTHREAD)procthread3, (void *)4, 0, &threadid3);
 for (k=0; k<24; k++)
    {pos.X=1; pos.Y=k+1;
     EnterCriticalSection(&csec);
     SetConsoleCursorPosition(hstdout,pos);
     SetConsoleTextAttribute(hstdout,
        FOREGROUND_BLUE|FOREGROUND_GREEN|FOREGROUND_RED);
     printf("%c++",lbuk[k]);
     LeaveCriticalSection(&csec);
     Sleep(1000); }
  getchar();
  DeleteCriticalSection(&csec);
  CloseHandle(hthread1);   CloseHandle(hthread2);   CloseHandle(hthread3);
}
```

Листинг 8.2.3. Фрагмент программы с многими нитями для MS Visual C++

Заметим, что в компиляторе системы MS Visual C++ очень строго отслеживается соответствие типов формальных и фактических аргументов, поэтому в приведенной программе использовано предварительное соотнесение прототипа вызова процедуры нити имени PFMSTHREAD для типа этой функции, выбранное программистом. Вместо кратких указаний позиционирования и задания цвета, присутствующих в наборе дополнительных функций пакета Borland, здесь приходится использовать полные API функции для позиционирования и управления цветом в Windows. Аналогичные изменения управления позиционированием и цветом следует произвести во всех подпрограммах данного примера для Visual C++.

8.3. Уничтожение нитей

Одной из немногих операций, которые можно выполнить над нитью как системным объектом, является уничтожение нити. Для уничтожения нити в OS/2 служила системная функция DosKillThread. Эта функция описывалась прототипом

```
APIRET DosKillThread(TID tid)
```

и имела единственный аргумент – идентификатор уничтожаемой задачи. Уничтожить можно было только нить того же процесса, но не нить другого.

Для самоуничтожения нити, иными словами прекращения ее работы по инициативе процедуры самой нити в операционных системах Unix, использующих стандарт POSIX, служит функция с прототипом

int pthread_exit(int code),

параметр которой задает код возврата, а для прекращения работы нити по инициативе другой нити служит системная функция с именем pthread_cancel. Она имеет прототип

int pthread_cancel(pthread_t tid),

где в качестве аргумента задается идентификатор нити, ранее полученный от функции создания нити. Останавливаемая нить и нить, вызывающая функцию pthread_cancel, обязательно должны относиться к одному процессу (но никогда к разным процессам!).

Средства управления нитями лучше всего развиты в современных версиях Unix, в частности в Linux. Прекращение работы нити по явному приказу другой нити является довольно решительной мерой, учитывая, что нити процесса выполняют общую работу. Поэтому уничтожение нити в общем случае может иметь непредусмотренный побочный эффект на результаты работы всего процесса. Эти соображения привели в Unix к введению средств, гибко управляющих возможностью уничтожения нити. В настоящее время такие средства состоят из двух вспомогательных функций pthread_setcanceltype() и pthread_setcancelstate() и состояний нити относительно возможности уничтожения. Эти состояния называются *асинхронно отменяемым*, *синхронно отменяемым* и *неотменяемым*. Асинхронно отменяемое состояние отвечает отсутствию ограничений на уничтожение нити – нить в этом состоянии можно уничтожить в любой момент и в любом месте ее выполнения. Синхронно отменяемая нить может быть уничтожена только тогда, когда она находится в некоторых точках своего выполнения – так называемых точках отмены (cancel points). Нить, находящуюся в неотменяемом состоянии, невозможно уничтожить приказом со стороны, пока она по своей инициативе не сменит это состояние. Переключение между отменяемыми состояниями и неотменяемым производится с помощью функции pthread_setcancelstate(), имеющей прототип

int pthread_setcancelstate(int state, int *oldstate),

где первый аргумент должен задаваться одной из символических констант PTHREAD_CANCEL_DISABLE и PTHREAD_CANCEL_ENABLE, определяющими состояние после его переустановки этой функцией. Второй аргумент в случае неиспользования может задаваться как NULL, в противном случае он передает адрес переменной для записи кода старого состояния отменяемости (представимого этими же символическими константами). Заметим, что рассматриваемая функция действует на ту именно нить, которая выполняет эту функцию и не предоставляет никаких возможностей для смены состояния отменяемости другой нити.

Переключение между асинхронно отменяемым и синхронно отменяемым состояниями нити задается функцией pthread_setcanceltype(), имеющей прототип

int pthread_setcanceltype(int type, int *oldtype),

где первый аргумент должен задаваться одной из символических констант PTHREAD_CANCEL_DEFERRED и PTHREAD_CANCEL_ASYNCHRONOUS, причем первая константа дает состояние именно синхронно отменяемой нити, а второй аргумент служит для возврата предыдущего состояния отменяемости, но может задаваться значением NULL, если такая информация не требуется.

Точка отмены явно создается между двумя последовательными вызовами специальной функции pthread_testcancel(), не имеющей аргументов. Такая конструкция отмечает место процедуры нити, в которой она может быть уничтожена. Кроме явного указания точек отмены, операционная система вводит в программу неявные точки отмены, в которых по усмотрению разработчиков ОС допустимо относительно безболезненное уничтожение нитей. К сожалению, такие точки не всегда достаточно документированы в справочной системе. Рекомендуется искать информацию о них с помощью вызова справки

man pthread_cancel

Точки отмены неявно могут устанавливаться длительно выполняющими функциями ввода-вывода, в частности вывода на экран.

Описанные средства демонстрируются примером программы для Unix, приведенной в листинге 8.3.1.

```
#include <pthread.h>
#include <stdio.h>
#include <signal.h>
char lbuk[ ]="abcdefghijklmnoprstuwxyz";
pthread_t tid1, tid2, tid3;

void procthread1(void *arg)
{int k, j;
  for (k=0; k<24; k++)
    {printf("\033[%02d;20H",k+1);
     printf("\033[1;34m");
     for (j=0; j<(int)arg; j++) printf("%c",lbuk[k]);
     if (k= =5) {pthread_kill(tid2, SIGSTOP);
     printf("-Suspend thread2");}
     if (k= =11) {pthread_kill(tid2, SIGCONT);
     printf("-Resume thread2");}
     printf("\n");
     usleep(800000);
```

```
    }
}

void procthread2(void *arg)
{int k, j;
  for (k=0; k<24; k++)
    {printf("\033[%02d;40H",k+1);
     printf("\033[1;32m");
     for (j=0; j<(int)arg; j++) printf("%c",lbuk[k]);
     if (k= =5) {pthread_cancel(tid3);
                 printf("-Terminate thread3");}
     printf("\n");
     usleep(1300000);
    }
}

void procthread3(void *arg)
{int k, j;
//   pthread_setcancelstate(PTHREAD_CANCEL_DISABLE, NULL);
///   pthread_setcanceltype(PTHREAD_CANCEL_DEFERRED, NULL);
  for (k=0; k<24; k++)
    {printf("\033[%02d;60H",k+1);
     printf("\033[1;31m");
     for (j=0; j<(int)arg; j++)
         printf("%c",lbuk[k]);
     printf("\n");
     usleep(1100000);
    }
}

int main()
{int k;
 int rc;
  printf("\033[2J\n");
  rc=pthread_create(&tid1, NULL, (void*)procthread1, (void*)2);
  rc=pthread_create(&tid2, NULL, (void*)procthread2, (void*)3);
  rc=pthread_create(&tid3, NULL, (void*)procthread3, (void*)4);

  for (k=0; k<24; k++)
    {printf("\033[%02d;1H",k+1);
```

```
    printf("\033[1;37m");
    printf("%c++",lbuk[k]);
    printf("\n");      usleep(1000000);
   }
  pthread_join(tid1,NULL);  pthread_join(tid2,NULL);  pthread_join(tid3,NULL);
  printf("\033[0m");
  return 0;
}
```

Листинг 8.3.1 Уничтожение нити по явному приказу другой нити в Linux

В этой программе вторая нить, реализующая действия процедуры procthread2 на пятом шаге (при k=5), отдает приказ на уничтожение третьей нити (нити с процедурой procthread3).

Если убрать символы комментариев перед первым оператором третьей нити, то вызов функции pthread_setcancelstate() в этом операторе будет переключать эту нить в состояние неотменяемости, и при запуске программы можно наблюдать, что приказ на уничтожение нити в действительности не выполняется. Попытки же перевести указанную нить в состояние синхронной отменяемости не имеют явного эффекта из-за того, что операторы printf вывода на экран неявно создают точки отмены, на которых и происходит прекращение работы этой нити по приказу на ее уничтожение.

В операционных системах Windows для уничтожения процессов служит системная функция TerminateThread, которая имеет прототип

BOOL TerminateThread(HANDLE hThread, DWORD dwExitCode).

Код возврата из завершившейся или принудительно прекращенной приказом TerminateThread нити может быть получен в другой нити того же процесса путем вызова вспомогательной функции GetExitCodeThread, которая имеет прототип

BOOL GetExitCodeThread(HANDLE hThread, DWORD *pExitCode).

Если задача, указанная аргументом hThread, в момент запроса кода возврата еще работает, то функция возвращает значение STILL_ACTIVE в качестве значения второго аргумента. Код возврата нормально завершающейся нити формирует функция завершения ExitThread, которая имеет прототип

void ExitThread(DWORD dwExitCode).

Обычно функцию уничтожения потока используют только тогда, когда управление потоком утеряно и он ни на что не реагирует.

Аналогично рассмотренным выше примерам, функцию уничтожения нити в операционных системах Windows можно продемонстрировать изменением программы из листинга 8.2.2, вставив оператор

```
if (k= =5) TerminateThread(hthread3, 0)
```

после операторов

```
        for (j=0; j<(int)arg; j++)     printf("%c",lbuk[k]);
```
в процедуре procthread2.

8.4. Приостановка и повторный запуск нити

Для нитей существует один вид системного действия, аналогов которому нет для процессов в операционных системах Windows и OS/2. Это – приостановка и повторный запуск отдельной нити. По смыслу действия одна из нитей процесса отдает приказ на приостановку другой нити этого же процесса. Приостановить нить другого процесса программным приказом невозможно (так как в подобном действии при внимательном анализе нет никакого практического смысла).

В операционной системе Unix для приостановки отдельной нити можно и нужно использовать уже знакомую читателю функцию pthread_kill со вторым параметром, задаваемым символической константой SIGSTOP, а для возобновления работы нити - ту же функцию, но уже с параметром SIGCONT. Следующий пример, представленный программой в листинге 8.4.1, демонстрирует приостановки и возобновление работы отдельной нити в операционной системе Unix, поддерживающей стандарт POSIX.

```
#include <pthread.h>
#include <stdio.h>
#include <signal.h>
char lbuk[ ]="abcdefghijklmnoprstuwxyz";
pthread_t tid1, tid2, tid3;

void procthread1(void *arg)
{int k, j;
  for (k=0; k<24; k++)
    {printf("\033[%02d;20H",k+1);
     printf("\033[1;34m");
     for (j=0; j<(int)arg; j++)
         printf("%c",lbuk[k]);
     if (k= =5) {pthread_kill(tid2, SIGSTOP);
     printf("-Suspend thread2");}
     if (k= =11) {pthread_kill(tid2, SIGCONT);
     printf("-Resume thread2");}
     printf("\n");      usleep(800000);
    }
}
```

```
void procthread2(void *arg)
{int k, j;
  for (k=0; k<24; k++)
    {printf("\033[%02d;40H",k+1);
     printf("\033[1;32m");
     for (j=0; j<(int)arg; j++)
         printf("%c",lbuk[k]);
     usleep(1300000);
    }
}

void procthread3(void *arg)
{int k, j;
   for (k=0; k<24; k++)
    {printf("\033[%02d;60H",k+1);
     printf("\033[1;31m");
     for (j=0; j<(int)arg; j++)
         printf("%c",lbuk[k]);
     printf("\n");
     usleep(1100000);
    }
}

int main()
{int k;
 int rc;

  printf("\033[2J\n");
  rc=pthread_create(&tid1, NULL, (void*)procthread1, (void*)2);
  rc=pthread_create(&tid2, NULL, (void*)procthread2, (void*)3);
  rc=pthread_create(&tid3, NULL, (void*)procthread3, (void*)4);
  for (k=0; k<24; k++)
    {printf("\033[%02d;1H",k+1);    printf("\033[1;37m");
     printf("%c++",lbuk[k]);
     printf("\n");      usleep(1000000);
    }
  pthread_join(tid1,NULL);   pthread_join(tid2,NULL);   pthread_join(tid3,NULL);
  printf("\033[0m");
  return 0;
}
```

Листинг 8.4.1. Программа с приостановкой нитей для Unix

В этой программе первая нить на 5-м шаге своей работы отдает приказ приостановки второй нити (идентификатор которой в tid2). Второй нитью мы здесь называем нить с идентификатором tid2, работающую на основе процедуры procthread2, а главную нить условно считаем нулевой. Приказ такой приостановки выдается в виде

pthread_kill(tid2, SIGSTOP).

Сама первая нить после такого приказа продолжает работать, а вторая, как легко наблюдать на экране по выводимым результатам, временно ничего не выводит, будучи приостановленной. Далее на 11-м шаге работы первой нити – в процедуре procthread1 – вызывается функция возобновления работы приостановленной второй нити. Эта функция в программе присутствует в виде pthread_kill(tid2, SIGCONT). После ее выполнения, как можно наблюдать по выводу процесса, который работает по программе из листинга 8.4.1, вторая нить продолжает свой вывод, а следовательно, работает дальше. Небольшие дополнения в программе относительно ее аналога в листинге 8.2.1 осуществляют поясняющий вывод в ходе ее выполнения и предназначены для наглядности наблюдаемого процесса.

Программные средства для приостановки и возобновления нитей в операционных системах типа Windows состоят из двух функций ResumeThread и SuspendThread. Обе они имеют единственный аргумент – хэндл нити, на которую они должны подействовать – приостановить или возобновить ее работу. Прототипы этих функций описываются в виде:

DWORD SuspendThread(HANDLE hthread);
DWORD ResumeThread(HANDLE hthread).

Использование этих функций не имеет никаких особенностей в сравнении с другими рассмотренными операционными системами. В листинге 8.4.2 приведены фрагменты программы, отсутствующие части которой должны быть взяты из листинга 8.2.2 (начальная часть, тексты процедур procthread1, procthread2 и опущенные детали остальных). Эта программа создает три нити, кроме главной, и в ходе их работы в одной из них (третьей по условному счету) приостанавливает работу первой из них. Затем через несколько шагов внутреннего цикла в процедуре третьей нити отдается приказ на возобновление работы приостановленной ранее нити.

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

DWORD WINAPI procthread3(void *arg)
{int k, j;
```

```
 COORD pos;
  for (k=0; k<24; k++) {
     . . .
     for (j=0; j<(int)arg; j++)       printf("%c",lbuk[k]);
     if (k==5)
       {SuspendThread(hthread1);
        pos.X=50; pos.Y=k+1; SetConsoleCursorPosition(hstdout,pos);
        printf("Suspend thread1 into step=5");
       }
     if (k==16)
       {ResumeThread(hthread1);
        pos.X=50; pos.Y=k+1; SetConsoleCursorPosition(hstdout,pos);
        printf("Resume thread1 into step=16");
       }
     . . .
     Sleep(1000);
    }
}

void main()
{unsigned long threadid1, threadid2, threadid3;
 int k;
 COORD pos;

  hstdout=GetStdHandle(STD_OUTPUT_HANDLE);
 . . .
  for (k=0; k<24; k++)      {
     . . .
     printf("%c++",lbuk[k]);
     if (k==9)
       {TerminateThread(hthread2, 0);
        pos.X=1; pos.Y=k+1; SetConsoleCursorPosition(hstdout,pos);
        printf("Kill thread2");
       }
     LeaveCriticalSection(&csec);
     Sleep(1500);
    }
. . .
 CloseHandle(hthread1);   CloseHandle(hthread2);   CloseHandle(hthread3);
}
```

Листинг 8.4.2. Фрагменты программа с уничтожением нитей для Windows

Заметим, что в программе для управления поведением нитей используются их хэндлы, но никак не используются идентификаторы нитей. Практически в Windows всегда заметна некоторая избыточность средств, которая частично оправдывается большой суммарной сложностью системы.

8.5. Ожидание завершения нити

Другой формой управления нитями внутри программы являются функции ожидания завершения нити. В определенной степени эта функция аналогична функциям ожидания окончания процессов, но относится к уровню нитей. В конечном счете она предназначена для той же цели – согласования совместной работы нитей на основе самого глобального по смыслу действия – полного завершения работы отдельной выполняемой программной единицы.

В операционной системе OS/2 функция ожидания завершения нити называется DosWaitThread и имеет прототип

APIRET DosWaitThread(PTID ptid, ULONG option).

Эта функция в данной операционной системе предназначена для целого спектра действий. Она может быть использована для ожидания завершения конкретной нити того же процесса, заданной идентификатором в первом аргументе. (Отметим, что этот аргумент – возвращаемый!) При этом второй аргумент должен быть задан символической константой DCWW_WAIT или просто ее значением, равным нулю. Она может быть использована для получения информации, завершена ли конкретная нить данного процесса. Тогда второй аргумент должен задаваться константой DCWW_NOWAIT, равной 1. Результаты выполнения функции DosWaitThread при этом выдаются через код возврата. Именно, если указанная в вызове нить завершена, возвращается нулевой код, если не завершена, возвращается код XXXX, представляемый символической константой XXX_XXXXX.

Эта же функция DosWaitThread может быть использована для приостановки выполняющей ее нити до тех пор, пока какая-то другая нить этого же процесса не будет завершена. Для этих целей функция DosWaitThread должна быть вызвана с нулевым значением переменной для идентификатора нити, на которую указывает первый аргумент функции. С этой целью можно использовать последовательность операторов

tid=0; DosWaitThread(&tid, DCWW_WAIT),

где переменная tid определена как имеющая тип TID. В результате такого выполнения функции DosWaitThread будет получено значение идентификатора (в переменной tid – для указанного примера), обозначающего ту нить, которая была завершена после вызова этой функции.

Наконец, эта же функция может быть использована без ожидания для получения информации, какая нить в данном процессе была завершена последней. В этом применении вызов функции должен иметь второй аргумент, равный DCWW_NOWAIT, а первый аргумент быть указателем на нулевое значение переменной для идентификатора нити.

В операционной системе Unix, поддерживающей стандарт POSIX, для ожидания нити служит функция pthread_join, имеющая прототип

int pthread_join(pthread_t tid, void** status).

Эта функция возвращает код возврата из нити, переданный через функцию pthread_exit(), завершающую нить. Функция пригодна для ожидания только конкретной нити, идентификатор которой задается первым ее аргументом. Второй аргумент вызова функции (указатель на указатель) может быть использован в виде последовательности

тип_результата restatus;
void *status=&restatus;
. . .
pthread_exit(&restatus); // в конце процедуры нити
. . .
pthread_join(tid, &status);
<*анализ значения* *status>

либо в простейшем случае может быть использовано явное преобразование типа.

```
#include <stdio.h>
char lbuk[ ]="abcdefghijklmnoprstvuwxy";
pthread_t tid1, tid2, tid3;

void procthread1(void *arg)
{int k, j;
  for (k=0; k<24; k++)      {
    // установить вывод в позицию (20,k+1) и синий цвет
    printf("\033[%02d;20H",k+1);      printf("\033[1;34m");
    for (j=0; j<(int)arg; j++)      printf("%c",lbuk[k]);
    printf("\n");
    usleep(1000000);
   }
}

void procthread2(void *arg)
{int k, j;
  for (k=0; k<24; k++)      {
```

```
      //установить вывод в позицию (40,k+1) и зеленый цвет
      printf("\033[%02d;40H",k+1);      printf("\033[1;32m");
      for (j=0; j<(int)arg; j++)      printf("%c",lbuk[k]);
      printf("\n");
      usleep(1000000);
     }
}

void procthread3(void *arg)
{int k, j, *rc;
  for (k=0; k<7; k++)      {
      // установить вывод в позицию (60,k+1) и красный цвет
      printf("\033[%02d;60H",k+1);  printf("\033[1;31m");
      for (j=0; j<(int)arg; j++)      printf("%c",lbuk[k]);
      printf("\n");
      usleep(1000000);
     }
     printf("\033[%02d;60H",k+1);     printf("\033[1;31m");
     printf("Waiting End thread1\n");
     pthread_join(tid1, (void**)&rc);
     printf("\033[%02d;60H",k+1); // установить вывод в позицию (60,k+2)
     printf("\033[1;31m");              // и красный цвет
     printf("End waiting\n");
    for (k=9; k<24; k++) {
      printf("\033[%02d;60H",k+1); // установить вывод в поз. (60,k+1)
      printf("\033[1;31m");                    // и красный цвет
      for (j=0; j<(int)arg; j++)      printf("%c",lbuk[k]);
      printf("\n");
      usleep(1000000);
     }
}

void main()
{int k;
  printf("\033[2J\n");//clrscr();
  pthread_create(&tid1, NULL, (void*)procthread1, (void*)2);
  pthread_create(&tid2, NULL, (void*)procthread2, (void*)3);
  pthread_create(&tid3, NULL, (void*)procthread3, (void*)4);
  for (k=0; k<24; k++)      {
     printf("\033[%02d;1H",k+1); //установить вывод в позицию (1,k+1) и белый цвет
```

```
    printf("\033[1;37m");
    printf("%c++\n",lbuk[k]);
    if (k==9)
      {pthread_cancel(tid2); //  !!!
       printf("\033[%02d;1H",k+1); // установить вывод в позицию (1,k+1)
       printf("Kill Thread2\n");
      }
    usleep(1500000);
   }
  printf("\033[%02d;30H",24); // установить вывод в позицию (30,24)
  pthread_join(tid1,NULL);   pthread_join(tid2,NULL);   pthread_join(tid3,NULL);
  printf("\033[0m");  // вернуть вывод на экран к стандартному цвету
}
```

Листинг 8.5.1. Программа с ожиданием завершения нитей для Unix

В программе с целью упрощения (ввиду того, что код возврата из процедуры не используется) значение кода завершения, возвращаемое с помощью функции pthread_join, описано как целочисленное (в виде *int *rc*;), а при вызове функции для второго аргумента используется преобразование типов в виде *(void**)&rc*. После вызова функции ожидания *pthread_join* третья нить, обратившаяся к этой функции, приостанавливается до тех пор, пока не завершится указанная в нем нить, т.е. первая нить. Завершение этой первой нити приводит к возобновлению действий приостановленной третьей нити. Для более легкого восприятия содержательных ситуаций программа процесса снабжена выдачей сообщений о приостановке и возобновлении нити.

В операционных системах Windows для ожидания завершения нити служит уже частично рассматривавшаяся универсальная функция ожидания WaitForSingleObject. Для ожидания завершения нити в качестве первого аргумента этой функции следует взять хэндл ожидаемой нити, а в качестве второго - значение предельного времени ожидания, в простейшем случае INFINITE. Каких-либо особенностей, отличающих ее от других форм ожидания, эта функция не имеет при использовании для ожидания нити.

Следующие фрагменты программы, представленные в листинге 8.5.2, демонстрируют простейшее ожидание завершения нити. Отсутствующие части программы совпадают с ее прототипом, записанным в листинге 8.2.2. Программа, восстановленная по указанным фрагментам, функционирует в Windows совершенно так, как было описано выше для программы в листинге 8.5.1, предназначенной для Unix.

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

```
. . .
DWORD WINAPI procthread3(void *arg)
{int k, j;
 COORD pos;
   for (k=0; k<7; k++)
     {// установка позиции вывода (60,k+1) и красный цвет вывода
      for (j=0; j<(int)arg; j++)        printf("%c",lbuk[k]);
      . . .
      Sleep(1000);
     }
    // установка позиции вывода (60,k+1) и красный цвет вывода
    . . .
    printf("Waiting End thread1");
    LeaveCriticalSection(&csec);
    WaitForSingleObject(hthread1, INFINITE);
    // установка позиции вывода (60,k+1) и красный цвет вывода
    . . .
    printf("End waiting");
    LeaveCriticalSection(&csec);
    for (k=9; k<24; k++)     {
      // установка позиции вывода (60,k+1) и красный цвет вывода
      . . .
      for (j=0; j<(int)arg; j++)        printf("%c",lbuk[k]);
      LeaveCriticalSection(&csec);
      Sleep(1000);
     }
}

void main()
{DWORD threadid1, threadid2, threadid3;
 . . .
 for (k=0; k<24; k++)    {
      . . .
      printf("%c++",lbuk[k]);
      if (k==9)
        {TerminateThread(hthread2, 0);
         pos.X=1; pos.Y=k+1; SetConsoleCursorPosition(hstdout,pos);
         printf("Kill Thread2");
        }
     LeaveCriticalSection(&csec);
```

```
    Sleep(1500);
    }
  pos.X=30; pos.Y=24; SetConsoleCursorPosition(hstdout,pos);
  . . .
}
```

Листинг 8.5.2. Программа с ожиданием завершения нитей для Windows

Данная программ принципиально не отличается от рассмотренных.

Считается, что разработка программ со многими нитями требует гораздо более высокой квалификации, чем обычное программирование. Отчасти это связано с инерционностью мышления программистов, сформировавшихся под влиянием идеи алгоритма. Напомним, что в алгоритме действия выполняются строго одно за другим, и никаких одновременных действий не допускается. При том, что поведение отдельной нити допускает естественное описание с помощью алгоритма, совокупное поведение программы со многими нитями оказывается гораздо сложнее, чем может описать отдельный алгоритм. Именно, в общем случае недетерминированное взаимодействие множества нитей и обусловливает трудность представления и мысленного моделирования поведения программы с многими нитями.

В то же время, многопоточные программы гораздо более естественно и полнее моделируют поведение объектов реального мира, где в действительности происходит множество взаимодействующих процессов и явлений, использующих общие элементы окружающей среды. Поэтому профессиональному программисту совершенно необходимо овладеть рассматриваемой разновидностью программирования.

Упражнения

1. Разработать программу для Windows с тремя нитями, дополнительными к главной и создаваемыми системными функциями CreateThread. Каждая из нитей должна использовать общие для всех нитей данные, заданные 23 первыми буквами латинского алфавита. Каждая из этих нитей на своем k-м шаге выводит со своей случайной задержкой на место "своего" столбца экрана k-ю букву из указанного массива латинских букв, причем с числом повторений, равным условному номеру нити. На 6-м шаге главной нити она приостанавливает первую из дополнительных нитей, а на 13-м шаге задает возобновление выполнения этой нити. Вторая дополнительная нить на 11-м своем шаге отдает приказ на уничтожение третьей дополнительной нити. Все управляющие указания должны отображаться сообщениями без прокрутки экрана (в фиксированные позиции экрана).

9. СРЕДСТВА ВЗАИМОДЕЙСТВИЯ ПРОГРАММНЫХ ЕДИНИЦ

9.1. Абстрактные критические секции

Если несколько процессов или нитей используют общие данные, то результат совместного использования этих данных может быть совершенно непредсказуем. Примером таких общих данных является использование общей для процессов очереди, куда эти процессы включают запрос на некоторое действие.

Как известно, установка звена в начало очереди состоит из двух совершенно необходимых операций:

1) значение указателя *top* (на начало очереди) записывается в поле связи во вставляемом звене;

2) адрес вставляемого звена записывается в указатель начала очереди *top*.

Пусть в участок программы с этими действиями входят два процесса, причем процесс А успевает выполнить шаг 1, а затем диспетчер ОС отбирает у него процессор, и далее выполняется процесс В. После того как процесс В вставит свое звено в очередь, рано или поздно диспетчер даст возможность продолжить выполнение процессу А. Он, продолжая начатые ранее действия, выполнит шаг 2 установки в очередь своего звена. В результате в поле связи обоих вставленных звеньев будет одно и то же значение, бывшее до начала их вставки в служебной переменной *top*, т.е. оба вставленных звена показывают полем связи на звено, вставленное до них последним, а указатель начала очереди top показывает только на звено, вставленное процессом А. Таким образом, звено процесса В как бы "потерялось", к нему нет доступа, начиная со значения указателя *top*. (Читателю рекомендуется для четкого представления происходящего выполнить рисунки очереди запросов и описанных операций по вставке звеньев в изложенной последовательности действий.)

Введем некоторые термины, сложившиеся в теории параллельных процессов. Зависимость результатов процесса от непредусмотренного в программе взаимодействия с другими процессами называют *состязанием* процессов. Теоретическим средством устранить состязание между процессами является решение запретить прерывания, на основе которых организована параллельность процессов. Но такое решение запрещает (хотя бы на время) параллельное выполнение процессов, к тому же годится только для однопроцессорных систем.

Более общим понятием, чем общие данные является, понятие *ресурса*, которое кроме общих данных может быть и общедоступной аппаратурой.Участок программы, в котором нить производит обращение к общему (разделяемому с другими нитями) ресурсу, называется *критическим интервалом нити*.

Для устранения состязаний процессов (или нитей) используют монополизацию ресурса. Монополизация ресурса – это временное предоставление его в исключительное использование одной нити. Как пояснялось в гл. 4 и 5, в современных ОС ресурсы являются собственностью процесса, а для нитей одного процесса – общими. Критический интервал нити поэтому ограничивают с начала и конца

специальными управляющими конструкциями операционной системы, в абстрактном изложении называемыми *прологом* и *эпилогом* критического интервала. Назначение этих конструкций – обеспечить нахождение в критическом интервале только одной нити.

9.2. Абстрактные семафоры

Общий подход к монополизации ресурсов с помощью специальных системных операций предложил голландский математик Е. Дейкстра (E. Dijkstra). Этот подход состоит в построении и использовании *семафоров*. Семафор – это защищенная от прямого доступа переменная, значения которой можно опрашивать и изменять только с помощью специальных операций P и V; кроме того, в самом начале к семафору применяется операция *инициализации*.

Простейшей формой семафоров являются двоичные семафоры, которые могут принимать только значения 0 и 1. Операции P и V над двоичными семафорами определяются следующим образом. Пусть S - двоичный семафор.

Для удобства и сокращения формулировок будем временно называть абстрактный процесс – *задачей*.

Операция P(S) выполняет действия: если S равно 1, то занести в S значение 0, иначе БЛОКИРОВАТЬ нить по семафору S, и если очередь готовых задач не пуста, то УСТАНОВИТЬ на процессор задачу из этой очереди.

Операция V(S) выполняет действия: если список задач, заблокированных по семафору S, не пуст, то ДЕБЛОКИРОВАТЬ задачу из этой очереди (т.е. перевести ее в очередь готовых задач), иначе занести в S значение 1.

Инициализация семафора – это установка значения семафора в 0 или в 1.

Для двоичных семафоров значение семафора 1 обозначает, что ресурс, защищаемый семафором, свободен, а значение семафора 0 – что ресурс занят. Операции P(S) и V(S) предназначены для использования в качестве конструкций *пролог* и *эпилог* критических интервалов. В начале критического интервала ставится операция P(S), а в конце критического интервала - операция V(S), иначе говоря операции P и V – это ограничители критического интервала, его "операторные скобки". Семафоры – не просто логические ключи, проверка значений которых позволяет продолжить или нет текущую задачу, они путем операций P и V глубоко связаны с внутренними действиями ОС над задачами. Для понимания существа операций P и V в них следует видеть две стороны: монополизацию ресурсов для избежания состязаний и управление внутренними очередями состояния задач.

С помощью семафоров просто решается классическая задача теории параллельных процессов, называемая "задачей читатели-писатели" (Readers-Writers). В этой задаче требуется обеспечить правильное функционирование многих абстрактных процессов, которые могут читать данные из общей области, и абстракт-

ных процессов, которые могут записывать данные в эту область. Причем допускается параллельное чтение несколькими читателями общих данных, когда никакой абстрактный процесс не записывает данные в эту область, но допускается одновременная работа только одного абстрактного процесса, записывающего данные, причем одновременно с ним не должен работать ни один абстрактный процесс чтения из общей области данных.

Для решения задачи с помощью абстрактных двоичных семафоров используются два семафора: семафор, который обозначим W, предназначенный для управления доступа абстрактными процессами – писателями, и семафор S, используемый в абстрактных процессах – читателях. Дополнительно используется обычная переменная N, подсчитывающая число абстрактных процессов, занимающихся параллельным чтением общих данных. Семафор S будет применяться для доступа к этой переменной N. Программы для читателя и писателя можно схематически изобразить в виде, приведенном на рис. 9.1.

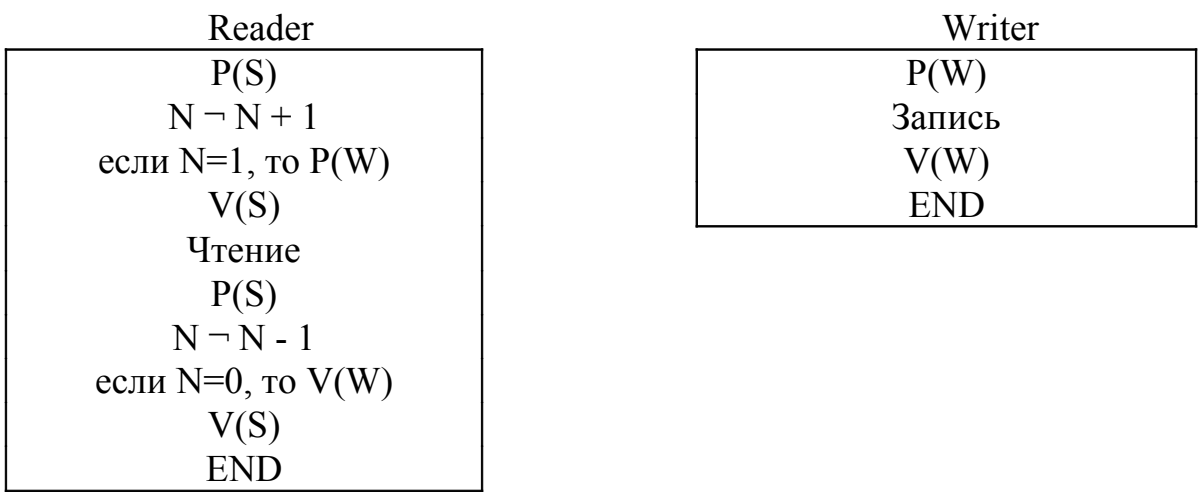


Рис. 9.1. Схемы программ "читателя" и "писателя"

Перед началом работы программ для всех читателей и писателей должна быть выполнена инициализация

S ¬ 0 , N ¬ 0, W ¬ 1.

Более сложной формой семафора является считающий, который по определению принимает любые неотрицательные значения. Операции P и V для считающего семафора S определяются следующими описаниями действий:

P(S) : если S > 0, то S ¬ S - 1, иначе WAIT(S);

V(S) : если список задач, заблокированных по семафору S, не пуст, то ДЕБЛОКИРОВАТЬ задачу из этой очереди (т.е. перевести ее в очередь готовых задач), иначе S ¬ S +1,

где системная функция WAIT(S) определяет ожидание освобождения семафора S, т.е. БЛОКИРОВАТЬ задачу по семафору S, и если очередь готовых задач не пуста, то УСТАНОВИТЬ на процессор задачу из этой очереди, иначе ждать освобождение семафора S.

Эта форма семафоров обусловила название операций: P – proberen (проверить), V – verholen (увеличить) на голландском языке. Считающий семафор показывает число единиц свободного ресурса.

Считающий семафор может быть обобщен путем распространения его значений на отрицательные числа следующим определением операций P и V:

P(S) : S ¬ S - 1; если S < 0, то WAIT(S);

V(S) : S ¬ S +1; если S £ 0, то ДЕБЛОКИРОВАТЬ задачу из этой очереди (т.е. перевести ее в очередь готовых задач).

Обобщенный считающий семафор своим значением равен разности между числом единиц наличного ресурса и числом запросов на них. При S ³ 0 значение такого семафора показывает число свободных единиц ресурса, при S < 0 – число задач, заблокированных по семафору S.

Использование считающих семафоров демонстрирует задача ПРОИЗВОДИТЕЛЬ – ПОТРЕБИТЕЛЬ, к виду которой относится, в частности, управление буферным пулом. Решение этой задачи, рассматриваемое для процессов $P_{пр}$ и $P_{потр}$, использует два семафора S1 и S2, причем перед запуском процессов $P_{пр}$ и $P_{потр}$ должна происходить инициализация:

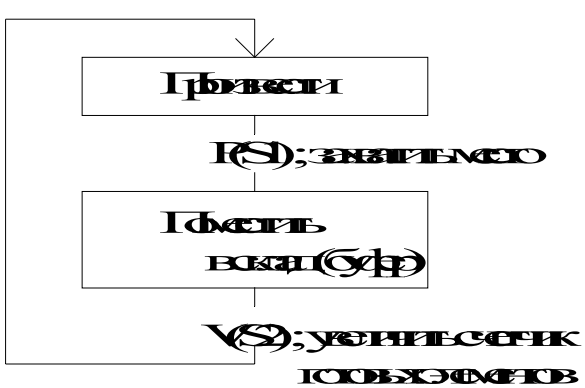
S1 ¬ *число_свободных_мест*,

S2 ¬ 0 (нуль элементов для использования).

Неотрицательные значения семафора S1 дают число свободных мест на складе (в буфере), а неотрицательные значения семафора S2 – число элементов, готовых для использования на складе (в буфере).

Схематическое решение для программ этой задачи имеет следующий вид, приведенный на рис. 9.2.

Процесс – производитель $P_{пр}$



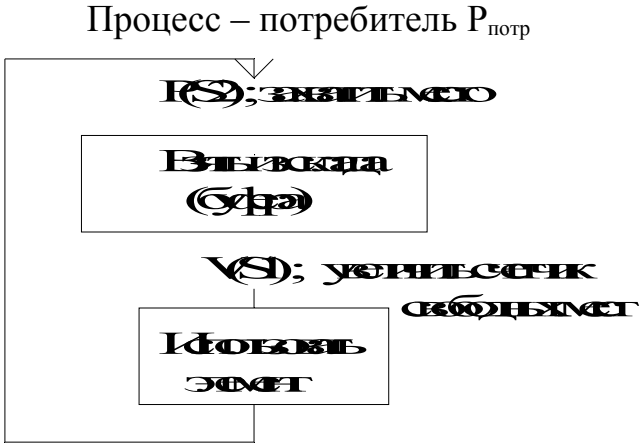


Рис. 9.2. Схема программ процессов производителя и потребителя

Следующая таблица демонстрирует последовательность возможного взаимодействия процессов $P_{пр}$ и $P_{потр}$ в предположении, что допустимое число свободных мест равно двум.

Табл. 9.1. События во взаимодействии процессов $P_{пр}$ и $P_{потр}$

Событие	$P_{пр}$	$P_{потр}$	S1	S2
0	-	-	2	0
1		P(S2) блокир.	2	-1
2	P(S1)		1	-1
3	V(S2) деблок.		1	0
4	P(S1)		0	0
5		V(S1)	1	0
6	V(S2)		1	1
7	P(S1)		0	1
8	V(S2)		0	2
9	P(S1) блокир.		-1	2
10		P(S2)	-1	1
11		V(S1) деблок.	0	1

Следует иметь в виду, что совершенно не обязательно, чтобы реальное взаимодействие приводило именно к такой последовательности событий. Конкретная последовательность определяется тем, насколько быстро один процесс выполняется относительно другого, и в конечном счете эта скорость может зависеть от выполнения других процессов.

Семафор – вовсе не идеальное средство для организации взаимодействия параллельных процессов (хотя лучшего не известно). Если при использовании двоичных семафоров S1, S2 в программах встретятся последовательности

Процесс A	Процесс B
P(S1)	P(S2)
P(S2)	P(S1)
.	.
.	.
.	.
V(S2)	V(S1)
V(S1)	V(S2)

то при следующей последовательности выполнения шагов этих процессов :

P(S1)-процесса A,

затем переключение на выполнение процесса B,

P(S2)-процесса B

возникает взаимное блокирование процессов. При этом процесс A ждет освобождения семафора S2 процессом B, а процесс B – освобождения семафора S1 процессом A и это ожидание может продолжаться сколько угодно. Такая взаимная блокировка процессов называется *тупиком*, клинчем или дедлоком (от английского слова deadlock).

9.3. Семафоры взаимоисключения

В современных ОС идея семафора подверглась дальнейшей детализации, в результате было получено несколько видов семафоров, заметно отличающихся своим поведением друг от друга. Наиболее простой и естественной формой стали семафоры взаимоисключения (mutual exclusion semaphore), сокращенно именуемые (по сокращению из английского названия – mutex) семафорами. Семафоры mutex отличаются от двоичных семафоров Дейкстры тем, что операцию V над семафором может осуществить только тот процесс, который выполнил предыдущую операцию P над ним без блокировки от такого действия. Для описания и использования этого ограничения вводится понятие *владения* семафором. (Возможно, что происхождение семафоров mutex в качестве действительной причины – скорее психологической, чем технической – имеет понятия собственности и владения как базовых понятий западной цивилизации.)

Операции, аналогичные абстрактным P и V, над семафорами взаимоисключения называются по-разному. Так, в операционной системе OS/2 запрос на владение таким семафором (аналогичный операции P) называется DosRequestMutexSem, в Windows такой запрос использует универсальную функцию ожидания любого объекта с именем WaitForSingleObject, в операционной системе Unix запрос на владение нитью семафором mutex называется pthread_mutex_lock. Для аналога операции V в рассматриваемых ОС разработчиками приняты следующие названия: в OS/2 – DosReleaseMutexSem, а в Windows – ReleaseMutex, в Unix –

pthread_mutex_unlock. Прежде чем перейти к детальному описанию упомянутых системных функций, сделаем некоторые пояснения. Запрос на владение mutex-семафором, выполняемый некоторой нитью, приостанавливает нить, выдавшую такой запрос, если этот семафор в настоящий момент принадлежит другой нити. (Конкретные действия с этими семафорами в современных ОС выполняются всегда нитью, в частности, главной нитью процесса, а процесс является собственником ресурсов.) Приостанавливается именно нить, а не весь процесс, если он состоит более чем из одной нити. Действия, выполняемые функциями DosReleaseMutexSem и ReleaseMutex в операционных системах OS/2 и Windows, называются освобождением семафора его владельцем (нитью, временно владеющей семафором). Если нить, не владевшая перед этим семафором взаимоисключения, вызывает функцию RequestMutexSem, то эта функция в качестве кода возврата дает отличное от нуля значение, обозначаемое символической константой ERROR_NOT_OWNER (с числовым значением 288). В той же ситуации функция ReleaseMutex в Windows возвращает значение ошибки, а последующий запрос кода ошибки функцией GetLastError также возвращает значение 288, символически задаваемое константой ERROR_NOT_OWNER.

Владение чем-то – дело хорошее (по крайней мере для владельца), но строгая необходимость отказа от владения только явным приказом – вызовом функции освобождения ресурса – влечет иногда серьезные проблемы применения семафоров, обеспечивающих в ОС монополизацию ресурсов. Если обратиться к описанной выше абстрактной задаче ЧИТАТЕЛИ-ПИСАТЕЛИ, то можно даже без экспериментов с отлаживаемой программой уяснить, что процесс – читатель, выполнивший вызов операции P(W), т.е. первый из процессов, начавших параллельное чтение данных, совсем не обязательно окажется и последним из группы процессов, осуществляющих одновременное чтение. Скорее даже наоборот, процесс, первым начавший чтение общих данных, через какое-то время завершит свою работу, но до ее завершения начнет чтение другой процесс и, может быть, еще какие- то процессы одновременно займутся чтением этих данных, защищаемых семафором W. Тогда последний из процессов, одновременно читавших данные (он обнаружит это по нулевому значению вспомогательного счетчика – переменной N), выполнит вызов функции, реализующей операцию V(W). Последнее должно привести в ОС OS/2 и Windows к возникновению ошибки – как результату попытки освобождения от владельца чужого владения! Вернемся теперь к описанию конкретных программных средств для использования семафоров взаимоисключения.

Семафор – не просто какая-то специальная переменная, а системный объект, особым образом используемый операционной системой. Поэтому для его создания необходимо явное приказание операционной системе. Это приказание имеет имена DosCreateMutexSem, CreateMutex и pthread_mutex_init в операционных системах OS/2, Windows и Unix соответственно. В результате этих вызовов исполняемая

программа (точнее, конкретный процесс, работающий по этой программе) получают хэндл семафора взаимоисключения в качестве результата вызова. Тип этого хэндла в OS/2 и в Unix специализирован и обозначается соответственно HMTX и pthread_mutex_t. В ОС Windows для всего многообразия хэндлов предназначен единственный тип хэндла с именем HANDLE (что является потенциальным источником ошибок программистов, работающих с Windows).

Имея хэндл, нить может использовать функции запроса владения и в дальнейшем – освобождения семафора владельцем mutex. Но создание нового семафора взаимоисключения – не единственный способ получить хэндл семафора для его использования. Получить хэндл несуществующего объекта, конечно, невозможно, но можно получить хэндл уже существующего объекта. Так как глобальные данные принадлежат всем нитям процесса, то хэндл, описанный как глобальный и созданный одной нитью в процессе, становится автоматически доступным и любой другой нити того же процесса. Поэтому получение хэндла объекта, не созданного процессом, имеет смысл только для тех объектов, которые были созданы другими процессами. Для этих целей в ОС служит функция открытия объекта, созданного другим процессом, в частности, открытие mutex-семафора, созданного другим процессом. Функции открытия mutex-семафора называются в OS/2 и Windows соответственно DosOpenMutexSem и OpenMutex. В ОС Unix нет функций открытия доступа к mutex-семафору, созданному в другом процессе. Доступ к mutex-семафору, созданному в другом процессе, возможен только тогда, когда он был создан с указанием (соответствующим флагом), что семафор позволяется использовать и другими процессами.

После использования семафора в данном процессе его следует в OS/2 и Windows явно закрывать (закрывать доступ к нему) с помощью системных функций DosCloseMutexSem и CloseHandle. Функция CloseHandle применяется Windows по завершении использования в процессе всевозможных объектов и формально закрывает хэндл объекта для прекращения возможности доступа к объекту. Внутри объектов ведется внутренний счетчик процессов, использующих данный объект. При каждом выполнении функции закрытия объекта в OS/2 и Windows значение этого счетчика уменьшается на единицу. При каждом открытии объекта в OS/2 и Windows значение этого счетчика увеличивается на единицу. При создании объекта значение этого счетчика устанавливается в единицу. Если в ходе закрытия объекта значение его внутреннего счетчика уменьшится до нуля, этот объект удаляется из системы (уничтожается).

Поскольку в Unix нет функций открытия доступа к mutex-семафору, созданному в другом процессе, то нет и функции закрытия такого mutex-семафора. Для удаления же из ОС mutex-семафора (после прекращения надобности в нем) предназначена функция pthread_mutex_destroy.

В операционной системе Windows прототипы функций, предназначенных специально для работы с mutex-семафором, следующие:

```
HANDLE CreateMutex(SECURITY_ATTRIBUTES* MtxAttrs,
                   BOOL bInitialOwner, STR* Name);
HANDLE  OpenMutex(DWORD DesiredAccess,
                   BOOL bInheritHandle, STR* Name);
BOOL ReleaseMutex(HANDLE  hMutex).
```

В простейших случаях (и большинстве других) параметр *MtxAttrs*, являющийся указателем на структуру атрибутов защиты для семафора, не используется, при этом сам параметр задается значением NULL. Параметр *bInitialOwner* определяет, будет ли владеть нить – создатель семафора – этим семафором или он будет создан никому не принадлежащим. В первом случае значение этого параметра должно быть задано равным TRUE, во втором – равным FALSE. Параметр *Name* определяет имя mutex-семафора. Семафоры здесь могут быть *именованные* и *неименованные*. Для определения неименованных семафоров параметр *Name* задается со значением NULL. Отличное от него значение определяет имя именованного семафора. Ограничения на построение этого имени практически отсутствуют, за исключением того, что в его состав не должен входить символ '\'. Для открытия семафора используется параметр *DesiredAccess*, который задается любой комбинацией флагов MUTEX_ALL_ACCESS и SYNCHRONIZE. Первый из них обозначает, что заданы все возможные флаги доступа, а второй – что полученный хэндл можно будет использовать в любых функциях ожидания события, в частности функции WaitForSingleObject. Параметр *bInheritHandle* показывает, будет ли наследоваться хэндл семафора потомками данного процесса. Чтобы такое наследование происходило, следует этот параметр задавать равным TRUE. Функции создания и открытия mutex-семафора возвращают либо нулевое значение, свидетельствующее об ошибке (код которой получается немедленным применением функции GetLastError), либо выдаваемое для использования в программе значение хэндла семафора. Функция ReleaseMutex выдает значение FALSE, если данная нить не является владельцем семафора.

Для запроса владения семафора Windows использует общую функцию ожидания WaitForSingleObject, а для его закрытия - общую функцию CloseHandle.

Следующий пример, представленный программой в листинге 9.3.1, демонстрирует простейшее использование в Windows рассмотренных функций.

```
// использование семафора для согласования доступа к экрану
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <windows.h>
```

```
#include <process.h>
#define STACK_SIZE 4096

char a1[ ] = "Земную жизнь пройдя до половины";
char a2[ ] = "Я очутился в сумрачном лесу";
char a3[ ] = "Путь правый утеряв во тьме долины.";
char b1[ ] = "Семафоры предназначены для правильного взаимодействия";
char b2[ ] = "Между процессами и нитями";
char b3[ ] = "В современных мультизадачных операционных системах.";
HANDLE hthread, hmtx;
void main(void)
{
void procthread(void *arg); //  прототип для процедуры нити
unsigned long threadid; // номер ID нити

 hmtx=CreateMutex(NULL, FALSE, NULL);
 if (hmtx= =NULL) {printf("Mutex=%d\n", hmtx); getchar();}

 hthread=(HANDLE)_beginthreadNT(procthread, STACK_SIZE, NULL, NULL,  0,
                                  &threadid);
 if (hthread= =NULL) {printf("hthread=%d\n", hthread); getchar();}

 while( TRUE )  // Повторять пока не будет нажата комбинация клавиш Cntrl-C
 { WaitForSingleObject(hmtx, INFINITE);
   printf("%s\n", a1);
   Sleep(100);
   printf("%s\n", a2);
   Sleep(100);
   printf("%s\n\n", a3);
   Sleep(100);
   ReleaseMutex(hmtx);
   Sleep(100);
 }
}

void procthread(void *arg) /* данные пишутся на экран  */
{
   while(TRUE) {
    WaitForSingleObject(hmtx, INFINITE);
    printf("%s\n", b1);
```

```
    Sleep(100);
    printf("%s\n", b2);
    Sleep(100);
    printf("%s\n\n", b3);
    Sleep(100);
    ReleaseMutex( hmtx );
    Sleep(100);
   }
}
```

Листинг 9.3.1. Программа для Windows

Программа задает функционирование двух нитей. Главная нить создает mutex-семафор, получает его хэндл в переменную hmtx, а затем создает (запускает на выполнение) нить на основе процедуры с именем procthread. Потом главная нить входит в цикл, который прекращается по нажатию любой клавиши клавиатуры. В этом цикле тремя отдельными операторами выводится три строки семантически связного текста, причем перед выводом этих строк запрашивается владение семафором с хэндлом hmtx, а после их вывода этот семафор освобождается. В процедуре, задающей функционирование нити, также тремя отдельными операторами выводятся три строки другого семантически связного текста, этот вывод повторяется в цикле с неограниченным числом повторений. Перед началом последовательности операторов вывода запрашивается владение тем же семафором (с хэндлом hmtx), что используется и в главной нити, а после вывода системной функцией этот семафор освобождается.

Все это сделано для того, чтобы вывод отдельных строк не перемежался с выводом строк другой нитью. Диспетчер ОС, распределяя процессор между нитями, предоставляет возможность выполнения команд, в частности, команд вывода на экран то одной из нитей рассматриваемой программы, то другой, и переключение это – между выполнением нитей – может произойти в любом месте. Без использования согласования работы нитей с экраном часто происходит следующее: вывод всех трех строк связного текста еще не окончен (выведена одна или две строки), а диспетчер предоставил процессор другой нити, и она стала выводить свой текст, вклинивая его в незавершенный текст другой нити. Введенный в программу семафор не позволяет начать работу последовательности операторов вывода трех строк, которые используют ресурс – экран, общий с другой нитью, пока другая нить не освободит этот ресурс (не закончит вывод цельной последовательности строк). Вывод семантически связанных строк текста выбран для данного примера, чтобы продемонстрировать, как несогласованное использование ресурса может очень заметно влиять на существо происходящего (в данном случае на смысл выводимого текста). Чтобы в полной мере убедиться в действенности использован-

ных средств, настоятельно рекомендуется после наблюдения как выполняется компьютером предложенная программа примера, изменить программу, удалив из нее вызовы функций запроса владения семафором и освобождения семафором, а затем запустить измененный вариант на выполнение. (При этом должна появиться очень заметная и досадная смесь строк из обоих выводимых текстов.) Заметим, что в реальной программе следует всегда после вызова системной функции проверять ее код возврата и по его значению предпринимать соответствующие действия, если вызов завершился неудачно. В данной программе для сокращения и с целью более легкого восприятия при первом знакомстве с новым материалом такие проверки исключены.

В операционной системе Unix функции для работы с mutex-семафором имеют прототипы:

```
int pthread_mutex_init(ptread_mutex_t* hmtx, ptread_mutexattr_* pattr);
int pthread_mutex_lock(ptread_mutex_t* hmtx);
int pthread_mutex_trylock(ptread_mutex_t* hmtx);
int pthread_mutex_unlock(ptread_mutex_t* hmtx);
int pthread_mutex_destroy(ptread_mutex_t* hmtx);
```

и описаны в заголовочном файле pthread.h. Во всех этих функциях в качестве первого аргумента используется указатель на хэндл семафора, что,в частности,позволяет получать от ОС значение хэндла в функции инициализации. В функции инициализации второй аргумент, задаваемый также указателем, определяет атрибуты для нового mutex-семафора. Функция pthread_mutex_trylock() позволяет опрашивать наличие владения семафора другой нитью (процессом) без блокировки опрашивающего процесса. При этом если заданный в вызове семафор занят, то функция вернет ненулевой код возврата. Как и в большинстве функций POSIX, значение кодов возврата у перечисленных функций, равное 0, обозначает успешное выполнение функции, а неравные нулю значения дают числовые значения кода ошибки их выполнения.

Следующий пример, представленный программой в листинге 9.3.2, демонстрирует решение описанной выше задачи для Unix.

```
// использование семафора для согласования доступа к экрану
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#define TRUE 1

char a1[ ] = "Земную жизнь пройдя до половины";
char a2[ ] = "Я очутился в сумрачном лесу";
```

```
char a3[ ] = "Утратив правый путь во тьме долины.";
char b1[ ] = "Семафоры предназначены для организации";
char b2[ ] = "Согласованного взаимодействия между процессами и нитями";
char b3[ ] = "В современных многозадачных операционных системах.";
pthread_mutex_t hmtx;

void main(void)
{void* procthread(void *arg); /* прототипы для новой цепочки */
pthread_t tid; // tid

 pthread_create(&tid, NULL, procthread, (void*)0);
 pthread_mutex_init(&hmtx, NULL);
while( TRUE) //  писать данные на экран  до нажатия Cntl-C
 {pthread_mutex_lock(&hmtx);
  printf("%s\n", a1);
  usleep(200000);
  printf("%s\n", a2);
  usleep(200000);
  printf("%s\n\n", a3);
  pthread_mutex_unlock(&hmtx);
  usleep(200000);
 }
  pthread_mutex_destroy(&hmtx);
}

void* procthread(void* arg)      // пишутся данные на экран
{  while(TRUE) {
   pthread_mutex_lock(&hmtx);
   printf("%s\n", b1);
   usleep(200000);
   printf("%s\n", b2);
   usleep(200000);
   printf("%s\n\n", b3);
   pthread_mutex_unlock(&hmtx);
   usleep(200000);
  }
}
```

Листинг 9.3.2. Программа для Unix

Кроме рассмотренных функций работы с mutex-семафорами,в некоторых ОС присутствуют дополнительные функции, облегчающие программисту разработку программ с такими семафорами.

Дополнительные функции позволяют получить информацию о текущем состоянии семафора. Для OS/2 – это функция с прототипом

APIRET DosQueryMutexSem (HMTX hmtx, PID *ppid, TID *ptid,
ULONG* pulCount),

где возвращаемый параметр pulCount позволяет получить значение внутреннего счетчика вызовов семафора, который определяет разность между числом вызовов функции запроса владения (функция DosRequestMutexSem) и числом вызовов функции освобождения семафора (функция DosReleaseMutexSem). Если в момент выполнения запроса DosQueryMutexSem о состоянии семафора он свободен (никому не принадлежит), то значение этого счетчика равно нулю. В результате вызова этой функции может быть получен код возврата, равный ERROR_SEM_OWNER_DIED (числовое значение 105), в этом случае параметр *pulCount* возвращает значение счетчика вызовов на момент завершения нити, которая не освободила семафор. Возвращаемые параметры *ppid* и *ptid* позволяют получить информацию о текущем владельце семафора – процессе и нити, которым принадлежит этот семафор. Если код возврата равен ERROR_SEM_OWNER_DIED, эти указателя дают возможность получить идентификаторы процесса и нити, прекратившейся без освобождения семафора.

Функция запроса владения семафора может выполняться в программе и над семафором, которым уже владеет данная нить, блокировка нити при этом не возникает, но увеличивается на единицу значение счетчика вызовов семафора. Поэтому для освобождения семафора данная нить должна будет выполнить функцию освобождения семафора не один раз, а столько раз, сколько она делала запрос на владение (в Windows универсальной функцией WaitForSingleObject).

В операционных системах Windows нет средств получить текущее значение счетчика вызовов, что в конечном счете неудобно для программиста. (Проникнув достаточно далеко во внутренние структуры ОС, это сделать можно, но путем достаточно большого дополнительного труда, причем в Windows эти возможности не документированы и остаются привилегией немногих профессионалов.)

В Linux семафоры взаимоисключения предоставляют дополнительные возможности. Они связаны с попытками нити захватить семафор, уже принадлежащий ей. Эти возможности называют *типами* mutex семафоров. Имеются три таких типа: *быстрый* семафор, *рекурсивный* семафор и *контролирующий* семафор. Попытка захвата быстрого семафора приводит к блокировке нити, но такая же попытка для рекурсивного семафора не приводит к блокировке. При этом для освобождения mutex семафора оказывается нужным выполнить вызов функции

pthread_mutex_unlock() столько раз, сколько перед этим выполнялось вызовов функции pthread_mutex_lock().

При использовании контролирующих семафоров операционная система обнаруживает попытки повторного захвата и сигнализирует об этом значением ошибки, возвращаемой функцией pthread_mutex_lock(), а именно возвращается значение, задаваемое символической константой EDEADLK.

По умолчанию в Linux создается быстрый семафор. Для других типов следует использовать инициализацию семафора с явно задаваемыми атрибутами. С этой целью нужно создать переменную типа *pthread_mutexattr_t* и передать указатель на нее функции pthread_mutexattr_init(). Затем следует задать тип семафора с помощью вспомогательной функции pthread_mutexattr_setkind_np(). Последняя функция имеет два аргумента, первый из которых задается адресом на экземпляр инициализированной структуры *pthread_mutexattr_t*, а второй – одной из констант PTHREAD_MUTEX_RECURSIVE_NP и PTHREAD_MUTEX_ERRORCHECK_NP – для рекурсивного и контролирующего типа соответственно. После такой подготовки экземпляра типа *pthread_mutexattr_t* следует использовать указатель на него в качестве второго аргумента функции pthread_mutex_init(), а сам экземпляр объекта атрибутов затем удалить вызовом pthread_mutexattr_destroy().

Как неявно подсказывает префикс _NP, рекурсивные и контролирующие семафоры специфичны для Linux и не могут использоваться в программах, переносимых в другие варианты ОС Unix.

9.4. Семафоры событий

Другой современной формой семафоров является семафор событий (Event-semaphore). Он предназначен для организации ожидания некоторого события, и когда такое событие произойдет, выполнение сможет продолжить не одна нить (как это происходит для mutex-семафоров), а все нити, дожидающиеся этого события. Естественно, что для программного использования event-семафоров требуются почти такие же системные функции, какие требовались для mutex-семафоров.

Прежде всего необходимо иметь средства создания event-семафоров. Для создания их в операционных системах OS/2 и Windows, предназначены соответственно функции с именами DosCreateEventSem и CreateEvent. Для открытия доступа к семафорам событий имеются функции DosOpenEventSem и OpenEvent. Напомним еще раз, что роль семафоров событий описательно близка к сигналам, посылаемым заинтересованным нитям для продолжения их действий, приостановленных до этого сигнала. Еще более описательно начинающие могут представлять себе действие событий операционной системы как звонок будильника, по которому просыпаются все запланировавшие проснуться по нему.

Для включения "механизма звонка будильника" – включения сигнала события служат функции DosPostEventSem и SetEvent. Чтобы такой сигнал-будильник (event-семафор) можно было использовать повторно для ожидания другого события, нужно иметь средство отключать такой семафор-будильник. Для принудительного выключения сигнала от event-семафора служат функции сброса, которые в операционных системах OS/2 и Windows, называются соответственно DosResetEventSem и ResetEvent. Ожидание события, связанного с event-семафором, в OS/2 выполняет функция DosWaitEventSem, а в Windows – вездесущая функция WaitForSingleObject, которая на этот раз должна в качестве аргумента использовать хэндл именно семафора событий.

Перед использованием семафора событий, созданного другим процессом – для получения доступа к нему – необходимо выполнить функцию открытия доступа к event-семафору. Для этих целей предназначены функции DosOpenEventSem и OpenEvent.

После завершения использования семафора событий (системного объекта, расходующего системные ресурсы) его следует закрыть, что и выполняется функцией DosCloseEventSem в OS/2 и универсальной функцией CloseHandle в Windows.

В операционной системе Windows функции, предназначенные специально для работы с event-семафором, имеют следующие прототипы

```
HANDLE CreateEvent(SECURITY_ATTRIBUTES*  pattributes,
                   BOOL  ManualReset,  BOOL fState,  STR*  pName);
BOOL SetEvent(HANDLE  hEvent);
BOOL ResetEvent(HANDLE  hEvent);
HANDLE OpenEvent(DWORD  access, BOOL  inherit,  STR*  pName);
BOOL PulseEvent(HANDLE  hEvent).
```

В этих функциях параметр *hEvent* обозначает хэндл семафора событий, параметр *pName* – указатель на имя семафора, которое в Windows может быть любым текстом, не включающем символа '\'. Семафоры для использования несколькими процессами в Windows должны быть обязательно именованные, неименованные допустимы только в пределах одного процесса. Неименованные семафоры событий создаются использованием нулевого указателя вместо параметра *pName* в функции создания такого семафора. Параметр *pattributes* является указателем на структуру атрибутов защиты, отличных от атрибутов по умолчанию, и в большинстве случаев задается нулевым указателем, что приводит к использованию атрибутов по умолчанию. Параметр *fState* задает начальное состояние семафора события и его начальное состояние FALSE отвечает отсутствию сигнала о событии, а состояние TRUE задает начальный сигнал о событии.

Параметр *inherit* определяет, будут ли дочерние процессы наследовать данный открываемый семафор события (если TRUE, то будут). Параметр *access* в

функции открытия семафора событий задает флаги доступа и может быть комбинацией следующих значений: EVENT_ALL_ACCESS, EVENT_MODIFY_STATE, SYNCHRONIZE. Первый флаг обозначает, что заданы все возможные типы доступа (и поэтому другие флаги программисту использовать не имеет смысла), флаг EVENT_MODIFY_STATE определяет, что после открытия хэндл можно будет использовать для функций SetEvent и ResetEvent, а последний из перечисленных флаг – что этот хэндл можно использовать в любых функциях ожидания событий.

Наиболее интересным по последствиям является применение параметра ManualReset в функции создания семафора события. С помощью его можно выбрать один из режимов работы семафора события, называемых *ручной* и *автоматический*. Ручному режиму соответствует значение TRUE параметра, а автоматическому – значение FALSE. Ручной режим отвечает необходимости использовать функцию сброса сигнала о событии для последующего использования того же семафора (функцию ResetEvent). В автоматическом режиме сигнал от семафора событий сбрасывается непосредственно при выходе из функции ожидания события от такого семафора. При этом только одна из нитей, ожидающих события, если таких нитей более одной, выходит из ожидания и продолжает свою работу. Все остальные нити, ожидающие такой семафор событий с автоматическим сбросом, по-прежнему остаются заблокированными – по крайней мере, до появления очередного сигнала от события.

По существу, вводя такие возможности, названные автоматическим режимом, разработчики неявно пристроили внутри системной реализации семафора событий еще и скрытый семафор взаимоисключения, связанный с первым семафором. Именно этот невидимый семафор взаимоисключения позволяет далее выполняться только одной из нитей, ждавших события. Заметим, что в Windows многие термины, сложившиеся в вычислительной технике часто используются очень своеобразно, в частности, семафоры событий называются просто событиями. Это является не просто большой вольностью, но даже противоречит сложившемуся во многих областях знаний понятию события, более того, противоречит использованию этого же термина в других частях Windows, например консольных функциях и изложении принципов построения графического интерфейса.

Следующий пример, представленный программой в листинге 9.4.1, демонстрирует использование в Windows основных функций работы с семафорами событий в ручном режиме.

```
#include <stdio.h>
#include <windows.h>
#include <process.h>
#define STACKSIZE 4096
#define BUFFLEN 80        // длина входного буфера
```

```
char name[BUFFLEN];     // буфер для имени
char age[BUFFLEN];      // буфер для возраста
HANDLE hthread, hevent;

int main()
{ void keyboard(void *arg); // прототип для новой нити
  unsigned long threadID;     // номер нити ID

  hevent=CreateEvent(NULL, TRUE,// флаг manual reset: будет нужен  ResetEvent
                FALSE,  NULL);// event без имени
  if (hevent= =NULL) {printf("Event=%d\n", hevent); getchar();}
  hthread=(HANDLE)_beginthreadNT(keyboard, STACKSIZE,  NULL, NULL,  0,
                                    &threadID);
  if (hthread= =NULL) {printf("hthread=%d\n", hthread); getchar();}

  printf("Введите ваше имя: ");
  WaitForSingleObject(hevent, INFINITE);
  printf("Введите ваш возраст: ");
  ResetEvent(hevent);
  WaitForSingleObject(hevent, INFINITE);
  printf("Поздравляю Вас %s в возрасте %s лет\n", name, age );
  getchar();
  CloseHandle(hevent);
  return 0;
}

void keyboard(void *arg)
{ gets(name);
  SetEvent(hevent);
  gets(age);
  SetEvent(hevent);
}
```

Листинг 9.4.1. Программа для Windows

В программе организована работа двух нитей, главная создает вспомогательную нить и семафор событий. Вспомогательная нить предназначена для обслуживания клавиатуры в данном примере. Она последовательно запрашивает с помощью функции gets() две строки вводимого текста (текст для символьного массива *name* и текста для символьного массива *age*). О получении очередного текста (с

клавиатуры) эта нить сигнализирует главной нити путем использования семафора события. Для этого после каждого ввода она выполняет функцию SetEvent, выдавая как бы "звонок" в главную нить.

Используется неименованный семафор с именем хэндла *hevent*, для которого при создании указан ручной режим работы. Поэтому после завершения ожидания события ввода очередной порции информации, осуществляемой другой нитью, которая устанавливает сигнал семафора с помощью функции SetEvent, выполняется вызов функции сброса сигнала семафора (функцией ResetEvent). Заметим, что за последним использованием семафора событий в главной нити нет необходимости его сброса, так как далее он уже не будет применяться для синхронизации действий нитей, поэтому за последним вызовом функции ожидания такой сброс не записан.

Главная нить, изобразив приглашения для ввода с консоли на экране, обращается к функции WaitForSingleObject, которая приостанавливает эту нить до тех пор, пока с помощью event-семафора вспомогательная нить не просигналит о событии. Тогда главная нить использует данные, введенные вспомогательной нитью, и выдаст результат их обработки (следующее приглашение) на экран. Сразу после этого она сбрасывает сигнал события функцией ResetEvent, чтобы дать вспомогательной нити тем же способом, как и раньше, сообщать о событии – готовности очередной строки вводимого текста.

В программе параметром FALSE при создании семафора события в функции CreateEvent указано, что начальное состояние отвечает отсутствию сигнала о событии. Иное значение этого параметра привело бы в данной программе к тому, что в главной нити не происходила бы приостановка на ожидании события и эта нить выводила на экран значение символьного массива, в который еще не введены данные, что резко отразилось бы на поведении программы. Для ясного осознания влияния семафора на процесс выполнения программы рекомендуется рассматриваемую программу запустить на выполнение с помощью системы разработки, а затем модифицировать, удалив вызовы функций ожидания события, и запустить на выполнение снова. Сравнение поведения двух таких вариантов программы должно помочь ясно осознать роль семафоров событий в подобных программах.

В операционной системе Unix нет даже термина семафор событий, но в ней имеются универсальные семафоры, даже более мощные, чем считающие абстрактные семафоры Дейкстры, и еще одна интересная вариация идеи семафора, названная условными переменными. В основном условные переменные соответствуют семафорам событий рассмотренных выше операционных систем и особенно семафорам событий с автоматическим режимом работы. Условные переменные используются для того, чтобы заблокировать нити до выполнения определенных условий, а момент выполнения такого условия можно рассматривать как событие.

Для создания условной переменной используется системная функция с именем pthread_cond_init. Она имеет прототип

int pthread_cond_init(pthread_cond_t * pcond, pthread_condattr_t * pcond_attr),

где аргумент *pcond* задает адрес условной переменной, значение которой возвращает данная функция, а аргумент *pcond_attr* дает указатель на структуру атрибутов, которые определяют свойства этой условной переменной. Если предполагается использовать свойства по умолчанию для такой переменной, то указатель *pcond_attr* должен задаваться нулевым (значение NULL).

Для задания срабатывания условной переменной, т.е. появления события выполнения условия этой переменной, существует функция pthread_cond_signal с прототипом

int pthread_cond_signal(pthread_cond_t * cond).

В результате выполнения этой функции всем нитям, ожидающим срабатывания условной переменной, посылается сигнал об этом событии.

Ожидание события срабатывания условной переменной задается функцией pthread_cond_wait с прототипом

int pthread_cond_wait(pthread_cond_t * cond, pthread_mutex_t * mutex).

Эта функция для своего применения требует не только созданной ранее условной переменной, указываемой аргументом *cond*, но и созданного ранее семафора взаимоисключения, который здесь указывается аргументом *mutex* (адресом mutex-семафора). Причем перед использованием для данных целей – использованием вместе с условной переменной – mutex-семафор должен быть первоначально захвачен во владение той же нитью, которая вызывает ожидание условной переменной. Функция ожидания условной переменной (срабатывания условной переменной) блокирует нить до тех пор, пока не появится сигнал установки этой переменной (от функции pthread_cond_signal). Как только такой сигнал появится, происходит разблокировка нити, сброс сигнала от условной переменной, и тут же внутри системной процедуры pthread_cond_wait() происходит захват во владение mutex-семафора, используемого этой процедурой. Причем из нескольких нитей, использующих вызов ожидания срабатывания условной переменной, только в одной удается захватить во владение этот mutex-семафор, именно той, которая вызовом pthread_mutex_lock() перед ожиданием условной переменной уже захватывала его во владение. Остальные нити продолжают оставаться приостановлены. Эти приостановленные нити ждут теперь очередного сигнала от условной переменной, т.е. нового выполнения функции pthread_cond_signal(). Заметим, что внутренний сброс условной переменной во внутренней процедуре pthread_cond_wait() как раз аналогичен автоматическому режиму семафора событий в Windows, но тут используемый mutex-семафор присутствует и функционирует явно.

Введение в стандартный набор системных функций не только семафоров взаимоисключении и семафоров событий с ручным сбросом, но и конструкций авто-

матического режима семафоров событий и условных переменных обусловлено недостаточностью первых двух типов для всего многообразия использования семафоров. Недостаток mutex-семафоров был отмечен ранее, обычные семафоры событий являются другой крайностью относительно абстрактных семафоров – они пробуждают все нити, ожидающие события. Ни те, ни другие не позволяют непосредственно воспроизвести операции P и V для семафоров, когда эти операции выполняются не в одном, а в различных процессах. Поэтому и появились условные переменные в Unix и автоматический режим семафоров событий в Windows, и эти конструкции могут быть использованы для реализации операций P и V в указанных ситуациях.

После использования, условная переменная должна быть уничтожена вызовом функции с прототипом

int pthread_cond_destroy(pthread_cond_t * cond).

Эта функция освобождает системные ресурсы, занятые условной переменной.

Для смыслового эквивалента ручного режима семафоров событий Unix содержит функцию с именем pthread_cond_broadcast, которая разблокирует все нити, которые ожидают события срабатывания условной переменной. Эта функция имеет прототип

int pthread_cond_broadcast(pthread_cond_t * cond)).

Следующий пример, представленный программой в листинге 9.4.2, демонстрирует решение в Unix описанной выше задачи с помощью только что рассмотренных функций.

```
#include <stdio.h>
#define TRUE 1
#include <pthread.h>
#define BUFFLEN 80      // длина входного буфера
char name[BUFFLEN];     // буфер для имени
char age[BUFFLEN];     // буфер для позиции

pthread_cond_t hevent;
pthread_mutex_t hmtx;

void main()
{ int rc;
  void* keyboard(void *arg); // прототип для новой нити
  pthread_t tid;

  rc=pthread_create(&tid, NULL, keyboard, (void*)0);   // запуск нити клавиатуры
  pthread_cond_init(&hevent, NULL);
```

```
  pthread_mutex_init(&hmtx, NULL);

  printf("Введите ваше имя: ");
  pthread_cond_wait(&hevent, &hmtx);
  printf("Введите ваш возраст: ");
  pthread_cond_wait(&hevent, &hmtx);
  printf("Поздравляю Вас %s в возрасте %s лет\n", name, age );
  pthread_cond_destroy(&hevent);
  pthread_mutex_destroy(&hmtx);
}

void* keyboard(void* arg)
{ gets(name);
   pthread_cond_signal(&hevent);
   gets(age);
   pthread_cond_signal(&hevent);
}
```

Листинг 9.4.2. Программа для Unix

Заметим, что в этой простой задаче можно было бы использовать вместо функции pthread_cond_signal() функцию pthread_cond_broadcast(). В данном простейшем примере установки события ожидает лишь одна нить, и поэтому нет особой необходимости использовать взаимную блокировку нескольких нитей, зависящих от общего события. В более общем примере условные переменные используют для построения сигнальных переменных. Такие переменные своим значением определяют, следует или нет выполнять какой-то участок программы. К сожалению, простейшая проверка сигнальной переменной, выполняемая непосредственно средствами языка программирования, приводит к совершенно нерациональному расходованию компьютерных ресурсов, особенно досадному при многопоточном построении процессов.

В качестве примера рассмотрим следующую задачу. Пусть программа, использующая несколько параллельно работающих нитей, в некоторых из них должна периодически выполнять некоторую вычислительную процедуру any_do(), но вызов этой процедуры следует выполнять только при установленном ключе - переменной *kflag*.

Основная часть такой программы, существенная для данного обсуждения, будет иметь вид

```
#include <stdio.h>
#include <pthread.h>
int kflag=0;
```

```
pthread_mutex_t  hmtx;

void* proc(void* arg)
{ int cur_kflag;
   while(1) {
   pthread_mutex_lock(&hmtx);
   cur_kflag=kflag;
   pthread_mutex_unlock(&hmtx);
  if (cur_kflag) any_do();
 }
 void setkflag(int val) {
  pthread_mutex_lock(&hmtx);
   kflag=val;
   pthread_mutex_unlock(&hmtx);
 }
```

где в остальной части программы предполагается создание исключающего семафора с хэндлом *hmtx*, запуск некоторого числа нитей на основе процедуры *proc* и периодическое обращение к процедуре *setkflag* для разрешения и запрещения выполнения внутренних процедур *any_do*(). Заметим, что неизбежное использование mutex семафора и вызовов pthread_mutex_lock(), pthread_mutex_unlock() следует из общности ресурса *kflag* для параллельно выполняемых нитей. В данной реализации будет происходить многократная проверка ключа *cur_kflag* при его нулевом значении, причем каждая из нитей, делающих это, будет заниматься только доступом к текущему значению ключа и проверкой, напрасно расходуя время процессора.

Чтобы устранить эти непроизводительные затраты, следует организовать ожидание установления сигнальной переменной, а не ее активный опрос. С этой целью можно использовать условную переменную. Возникающее при этом решение представлено в следующих строках:

```
#include <stdio.h>
#include <pthread.h>
int kflag=0;
pthread_mutex_t  hmtx;
pthread_cond_t  hev;

void* proc(void* arg)
{while(1) {
   pthread_mutex_lock(&hmtx);
   while (!kflag) pthread_cond_wait(&hev, &hmtx);
```

```
pthread_mutex_unlock(&hmtx);
       any_do();
     }
     void setkflag(int val) {
      pthread_mutex_lock(&hmtx);
kflag=val;
pthread_cond_signal(&hev);
pthread_mutex_unlock(&hmtx);
     }
```

В последнем решении процедуры *proc*, когда обнаруживается нулевое значение сигнальной переменной (вместо следующей попытки опроса ее), то запрашивается ожидание сигнала от условной переменной, обозначенной хэндлом *hev*. Нить, выполняющая эту процедуру, блокируется на этом ожидании и не расходует процессорного времени. Ее действия возобновляются лишь по поступлении сигнала от установки сигнальной переменной с помощью функции pthread_cond_signal(), причем в активное состояние переходит только одна из нитей, ожидающих этого сигнала.

Особенностью условных переменных в Unix является то, что сигнал, посылаемый такой переменной, действует только в том случае, если его в этот момент ожидает какая-то нить. Если же этот сигнал был послан в отсутствие ожидающей нити, а позже некоторая нить запросила ожидание этого же сигнала, то она будет заблокирована до момента следующей выдачи сигнала. Тем самым, условная переменная Unix все же заметно отличается от семафоров событий в Windows.

9.5. Средства группового ожидания

Множественные ожидания событий или освобождения общих ресурсов возникают, когда по существу решаемой задачи может оказаться необходимым ожидать не одного, а более чем одного события или ресурса. Например, продолжить выполнение можно только после освобождения ресурса R1 или ресурса R2. Если в программе вначале поставить функцию ожидания ресурса R1, а затем функцию ожидания ресурса R2, то при освобождении ресурса R2, когда можно было бы по условию задачи продолжать работу, нить процесса по-прежнему будет заблокирована на функции ожидания ресурса R1, который все еще занят. Поставив вызовы функций в обратном порядке – вначале ожидания ресурса R2, затем ресурса R1, попадаем в аналогичную ситуацию невыполнения условий задачи, когда ранее освободится ресурс R1, а ресурс R2 все еще будет занят. Без помощи операционной системы подобную задачу решить оказывается невозможным. Поэтому в со-

став стандартных системных функций введены функции множественного ожидания.

В Windows и OS/2 входят специальные функции множественного ожидания. В OS/2 эти функции базируются на понятии и типе данных *семафоров множественного ожидания* – muxwait semaphore, сокращенно обозначаемых в наименовании системных функций как MuxWaitSem.

В Windows для множественного ожидания предназначена универсальная функция WaitForMultipleObjects с прототипом

```
DWORD WaitForMultipleObjects(DWORD  cObjects,
        CONST HANDLE *phObjects,// address of object-handle array
        BOOL  fWaitAll,   // wait flag
        DWORD  Timeout),        // time-out interval in milliseconds,
```

где параметр *cObjects* для данного применения задает число семафоров в наборе, параметр *phObjects* – адрес массива хэндлов отдельных семафоров в наборе, параметр Timeout – время ожидания в миллисекундах или записывается символической константой INFINITE – для бесконечного ожидания. С помощью параметра *fWaitAll* определяется вариант ожидания – ожидать срабатывания всех семафоров в наборе (значение параметра для этого должно быть TRUE) или ожидание завершается при срабатывании хотя бы одного семафора в наборе (при значении FALSE этого параметра). Возвращаемые значения этой функции, равные сумме константы WAIT_OBJECT_0 и числового значения k, информируют программу, что ожидание было прекращено по причине срабатывания k-го семафора в наборе.

В операционной системе Unix не предусмотрено стандартных средств для множественного ожидания срабатывания набора mutex-семафоров или семафоров ожидания (условных переменных). Вместо этого присутствуют мощные программные средства, позволяющие строить произвольные наборы считающих семафоров, которые будут рассматриваться далее в п. 9.7.

9.6. Программные критические секции

Программный интерфейс новых ОС не только предоставляет сложные универсальные средства решения для системных проблем, как это было в более ранних ОС, но и ориентирован на средства, позволяющие программисту решать отдельные относительно частные задачи. Следует, впрочем, отметить что последние версии давно созданных ОС также включают системные средства, облегчающие программисту его программное взаимодействие с ОС.

Критические интервалы процесса и нити традиционно организуются с помощью семафоров, но в последних ОС для их построения введены дополнительные системные функции. Эти функции получили названия функций работы с критическими секциями.

В операционной системе Windows критические секции необходимо предварительно описывать в программе структуры данных для функций, применяющих критические секции, и использовать функции инициализации таких секций. Функция инициализации имеет прототип

VOID InitializeCriticalSection(CRITICAL_SECTION *pCriticalSection),

где параметр функции задает адрес структуры данных типа CRITICAL_SECTION. Поля этой структуры, как правило, явно не используются и не изменяются.

Вход в критическую секцию задается в программе функцией EnterCriticalSection, которая имеет прототип

VOID EnterCriticalSection(CRITICAL_SECTION *pCriticalSection),

а завершение критической секции программы должно указываться вызовом функции с прототипом

VOID LeaveCriticalSection(CRITICAL_SECTION *pCriticalSection).

После использования объект критической секции должен быть удален из системы явным вызовом функции

VOID DeleteCriticalSection(CRITICAL_SECTION *pCriticalSection).

Критические секции, построенные с помощью рассмотренных функций, могут быть вложенными. Следует отметить, что в Windows повторный вход в ту же критическую секцию функцией EnterCriticalSection не приводит к переходу задачи в состояние ожидания, но для выхода из критической секции после этого необходимо выполнить функцию LeaveCriticalSection для того же аргумента столько раз, сколько выполнялась функция входа в такую секцию.

9.7. Программные семафоры с внутренним счетчиком

В операционной системе Windows считающие семафоры создаются функцией CreateSemaphore. Функция имеет прототип

HANDLE CreateSemaphore(
SECURITY_ATTRIBUTES *pSemaphoreAttributes,
LONG lInitialCount, LONG lMaximumCount, LPCTSTR lpName).

Эти семафоры могут быть неименованными или именованными и тогда параметр *lpName* должен задавать адрес имени семафора. Неименованные семафоры задаются путем записи значения NULL в качестве значения параметра *lpName*. Рассматриваемые семафоры в Windows могут принимать неотрицательные целые значения, но эти значения ограничиваются величиной параметра *lMaximumCount*, которая задает максимальное значение для данного семафора. Параметр *pSemaphoreAttributes* задает адрес структуры атрибутов защиты и в простейших случаях берется равным NULL, что заставляет ОС использовать для него атрибуты по умолчанию. Параметр *lInitialCount* задает начальное значение семафора,

точнее его внутреннего счетчика, так как семафор имеет более сложное строение, чем просто переменная (это значение не может быть отрицательным и должно быть не больше значения *lMaximumCount*). Функция возвращает хэндл созданного семафора при удачном выполнении, и нулевое значение – при неудаче.

Для выполнения операции, аналогичной операции P абстрактных семафоров, следует использовать все ту же универсальную функцию WaiForSingleObject. Каждое выполнение этой функции, когда ее первым аргументом является хэндл считающего семафора, уменьшает внутренний счетчик семафора на единицу, но только в том случае, если он был больше нуля. Блокировка нити при этом не производится. Если же значение внутреннего счетчика семафора на момент выполнения функции WaiForSingleObject было равно нулю, то производится блокировка данной нити. Блокировка действует до тех пор, пока какая-то другая часть текущего комплекса программ (другая нить процесса, другой процесс) не выполнит функцию ReleaseSemaphore, освобождающую семафор. Функция эта имеет прототип

BOOL ReleaseSemaphore(HANDLE hSemaphore,
LONG cReleaseCount, LONG *plPreviousCount).

Функция освобождения считающего семафора увеличивает значение внутреннего счетчика семафора на заданное при вызове функции значение аргумента *cReleaseCount*, причем параметр *plPreviousCount* должен содержать адрес переменной, в которой возвращается предыдущее значение этого счетчика или значение NULL, если такое предыдущее значение не требуется в программе. Параметр *hSemaphore* должен задавать при вызове хэндл семафора, созданного или открытого в данном процессе. Значение аргумента *cReleaseCount* обязательно должно быть больше нуля. Если это значение такое, что его сумма с предыдущим значением счетчика семафора (значением перед выполнением функции освобождения) больше максимального значения (разрешенного параметром *lMaximumCount* при создании семафора), то значение счетчика никак не меняется, а функция возвращает значение FALSE, что свидетельствует об ошибке выполнения.

Когда значение внутреннего счетчика семафора больше нуля, он считается открытым, т.е. выполнение функции ожидания для такого состояния счетчика не блокирует нить.

Все нити процесса, создавшего считающий семафор, могут им пользоваться. Если считающий семафор создан другим процессом, то для доступа к нему этот семафор необходимо *открыть* с помощью функции OpenSemaphore, которая имеет прототип

HANDLE OpenSemaphore(DWORD fdwAccess, // access flag
BOOL fInherit, LPCTSTR lpszSemName).

Необходимым условием применения такой функции к считающему семафору является наличие у семафора имени. При открытии семафора адрес этого имени задается параметром *lpszSemName*. Параметр *fInherit* определяет,будет ли наследо-

ваться открытый семафор дочерними процессами для процесса, выполняющего открытие (наследуется при значении параметра TRUE). Параметр *fdwAccess* задает флаги доступа к открываемому семафору и может принимать значение символической константы SEMAPHORE_ALL_ACCESS или любую комбинацию констант SEMAPHORE_MODIFY_STATE и SYNCHRONIZE. Флаг SEMAPHORE_MODIFY_STATE разрешает использование функции ReleaseSemaphore() для открываемого семафора в данном процессе, а флаг SYNCHRONIZE дает аналогичное разрешение на использование любых функций ожидания, в частности WaiForSingleObject. Флаг SEMAPHORE_ALL_ACCESS заменяет комбинацию двух других флагов и разрешает все виды доступа – любые функции работы с этим семафором.

Напомним, что после использования считающий семафор, как и любой другой системный объект, должен быть явно закрыт вызовом функции CloseHandle для хэндла этого семафора.

В операционной системе Unix ее классические семафоры представляют собой не отдельные объекты, а целые группы (массивы) счетчиков семафоров, причем эти внутренние счетчики семафоров могут принимать любые неотрицательные значения. Отдельный внутренний семафор в такой группе описывается структурой *sem*, заданной как

```
typedef struct sem
{ushort semval;      //Значение семафора
 pid_t sempid;       // ID процесса, выполнившего последнюю операцию
 ushort semncnt;     // Число процессов,ожидающих увеличения счетчика
 ushort semzcnt;     // Число процессов, ожидающих обнуления семафора
};
```

Для получения доступа к семафору и для его создания, если он не существовал, предназначена системная функция semget(), имеющая прототип

int semget (key_t key, int nsems, int semflag).

Применение этой функции требует включения в программу заголовочных файлов с помощью директив

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
```

Этих же заголовочных файлов требуют и другие функции для работы с рассматриваемыми семафорами, эти функции будут описываться далее уже без упоминания о заголовочных файлах.

Параметр *key* функции semget() выполняет ту же роль, что и имя семафора в других ОС, практически это просто числовой код, однозначно идентифицирующий конкретный семафор. Этот код следует в простейших случаях просто выбирать согласованно (т.е. с одинаковым значением) для процессов, использующих

такой семафор. Для более серьезного использования в Unix имеется специальная функция задания уникального кода для создаваемого семафора. Параметр *nsems* определяет число счетчиков (внутренних одиночных семафоров), которые требуется создать, а параметр *semflag* – права доступа к создаваемому семафору и флаги его создания. Права доступа определяются так же, как и для файлов и являются комбинацией битов, задающих возможность чтения, записи и выполнения отдельно для категорий пользователей (владельца, членов группы, всех остальных). Флаги создания семафоров задаются символическими константами IPC_CREAT и IPC_EXCL. Первая из них задает режим создания семафора для функции semget(), а при отсутствии этой константы выполняется открытие семафора с указанным параметром *key* идентификатором. Флаг IPC_EXCL при создании семафора требует, чтобы при наличии одноименного семафора (уже существующего семафора с тем же значением идентификатора *key*) функция semget() возвращает ошибку. Неудачное выполнение этой функции влечет возвращаемое значение, равное -1, а при успешном выполнении функция возвращает в качестве своего значения хэндл созданного или открытого (в зависимости от того, что запрашивалось) семафора. Заметим, что в Unix для подобных возвращаемых значений, идентифицирующих системный объект, используется чаще всего термин *идентификатор*, а не термин *хэндл*.

Для основных операций над семафорами используется функция с именем semop. Собственно вид операции и характеристики ее выполнения задаются вспомогательной структурой данных, имеющей тип *sembuf*. Заметим, что такой многоступенчатый подход дает в результате ряд очень общих возможностей, но резко отличается от рассмотренных ранее способов управления системными объектами. Для начинающего он не очень нагляден и довольно сложен в освоении, требуя дополнительного сосредоточения на деталях. Структура *sembuf* описывается в виде

```
struct sembuf
{short sem_num;     // номер семафора в группе
 short sem_op;      // операция
 short sem_flg;     // флаги операции
};
```

Сама функция операций над семафорами имеет прототип

```
int semop (int hsem, struct sembuf* opsem, size_t nops),
```

где *hsem* – хэндл (идентификатор) семафора, созданного или открытого ранее функцией semget(), *opsem* – адрес массива описания структур операции над отдельным семафором в группе, а параметр *nops* задает число внутренних операций, выполняемых в вызове функции, т.е. число используемых элементов в массиве, заданном *opsem*.

Допускаются три возможные операции над семафорами, определяемые полем *sem_op*. Если значение поля *sem_op* положительно, то текущее значение счетчика

для указанного номером семафора увеличивается на эту величину (semval += sem_op), это первая операция. Если значение *sem_op* равно нулю, то процесс ожидает, пока счетчик семафора не обнулится (вторая операция – ожидание). Если значение поля *sem_op* отрицательно, то процесс ожидает, пока значение счетчика не станет большим или равным абсолютной величине *sem_op*, а затем значение этого счетчика уменьшается на эту абсолютную величину (semval -= abs(sem_op) – третья операция). Первой операции в основном соответствует функция ReleaseSemaphore из Windows, второй функции – чистое ожидание обнуления семафора (именно обнуления, а не достижения положительных значений). Третья функция не имеет близких аналогов в других ОС. Первая функция – безусловное изменение семафора, вторая – проверка и ожидание (условное выполнение), третья – проверка и изменение (условное выполнение).

При работе с такими универсальными семафорами программист сам решает, как из функции semop() сделать ту или иную производную функцию действий над семафором. В частности, разработчик взаимодействующих программ сам волен решать, какие значения счетчика сделать приостанавливающими процесс, а какие - разрешающими дальнейшее выполнение.

Например, можно предложить следующие два варианта двоичного семафора, получающегося из универсального соответствующим подбором структуры *sem_op* [14].

Сделаем значение 0 разрешающим, а значение 1 – запрещающим. Для этого опишем следующие массивы записей типа sem_op:

struct sembuf sop_lock[2] = { {0, 0, 0), {0, 1, 0}};

struct sembuf sop_unlock[1] = {0, -1, 0};

Элемент sop_lock[0] задает ожидание обнуления семафора (в нем sem_op=0), элемент sop_lock[1] – увеличение семафора на 1 (положительное, т.е. увеличивающее значение sem_op равно 1). Оба действия предполагается выполнять над начальным (нулевым по индексу) семафором в группе, т.к. поле *sem_num* равно 0, последнее поле – для флага – не используется (также нулевое). Элемент sop_unlock[0] задает обнуление значения семафора

Теперь для запирания ресурса (операции P над семафором) следует выполнить вызов

semop(hsem, &sop_lock[0], 2),

а для освобождения ресурса (операции V над семафором) следует выполнить вызов

semop(hsem, &sop_unlock[0], 1).

В первом вызове указано, что число внутренних операций равно 2, именно столько записей подготовлено для действий в массиве *sop_lock*, во втором вызове третий параметр задан числом 1, т.е. предполагается выполнение только одной внутренней операции, которую описывает одноэлементный массив *sop_unlock*.

В другом варианте изменим смысл значений счетчика семафора: значению 0 будет отвечать доступ к ресурсу (семафор открыт), а значению 1 – закрытый доступ к ресурсу (семафор закрыт). Для этого опишем структуры операций для семафора следующим образом:

```
struct sembuf  sopp_lock[2]= {0, -1, 0};
struct sembuf  unsopp_lock[1]= {0, 1, 0};
```

Теперь ресурс запирается (операция P над семафором) вызовом функции в виде

```
semop(hsem, &sopp_lock[0], 1),
```

а освобождается (операция V над семафором) вызовом

```
semop(hsem, &unsopp_lock[0], 1).
```

Второй вариант кажется проще, но на самом деле в его использовании имеется потенциальная опасность, так как в Unix семафоры создаются с начальными нулевыми значениями внутренних счетчиков. Задание произвольных начальных значений здесь не предусмотрено. Поэтому сразу после создания второй вариант семафора запирает ресурс, что не всегда удобно для программиста. Использование операции unsopp_lock сразу после создания семафора не снимает всех проблем, так как между вызовами создания и освобождения возможно переключение процессора на другой процесс, который, в свою очередь, может вызвать ту же операцию освобождения семафора. Последнее приведет после операции unsopp_lock в процессе, создавшем семафор, к значению внутреннего счетчика, ставшего в результате двух таких последовательных вызовов (хотя и в разных процессах) равным 2, а не 1, как можно было рассчитывать.

Для решения этой проблемы в Linux может быть использована функция semctl(), которая среди прочих своих возможностей позволяет устанавливать начальные значения в наборе семафоров. Данная функция имеет прототип

```
int  semctl(int semid, int num, int cmd, union semun arg),
```

где *semid* задает хэндл (идентификатор) набора семафоров, *num* – число семафоров в наборе, над которым выполняется операция, а *cmd* – тип операции над семафором. Этот аргумент *cmd* может задаваться константами IPC_STAT, IPC_SET и IPC_RMID. Первая из них задает операцию получения информации о конкретном семафоре, вторая служит для операции установки значения отдельного семафора в группе, а последняя предназначена для операции удаления семафора. Кроме того константой SETALL можно задать операцию установки значений всех семафоров в группе.

Вспомогательная структура данных типа semun определяется как

```
union semun {
  int val;
  struct semid_ds *buf ;
  unsigned short int *array;
  struct seminfo *__buf ;
```

```
};
```
и служит для получения осведомительной информации о конкретном семафоре или установке его значения. Собственно значения для набора семафоров при этом задаются во внутреннем массиве с именем array в экземпляре структуры, причем второй аргумент функции semctl() для данной операции должен задаваться нулевым значением.

В частности, установка единичного начального значения в массиве семафоров с хэндлом *hsem*, где массив состоит всего из одного элемента задается фрагментом

```
union semun semar;
unsigned short val[1]={1};
 . . .
semar.array=val;
semctl(hsem, 0, SETALL, semar);
```

С учетом редкой особенности у функции semctl() – передаче аргумента типа структуры данных не по адресу, а по значению, – последний аргумент даже в случае его содержательного неиспользования приходится задавать и нельзя в этом случае ограничиться константой NULL. В частности, удаление указанного выше семафора можно выполнить вызовом функции

```
semctl(hsem, 1, IPC_RMID, semar),
```

где содержимое экземпляра структуры *semar* совершенно не существенно.

Семафоры взаимоисключения и событий иногда оказываются не очень удобными для программиста, так как абстрактные семафоры приходится реализовывать в виде достаточно сложных комбинаций семафоров взаимоисключения и событий. Применение же универсальных считающих семафоров Unix, во первых, достаточно сложно, во вторых, они предназначены для согласования только процессов, а применение их для правильного взаимодействия отдельных нитей в рамках одного процесса не предусматривалось в первоначальных версиях. Из-за обеспечения преемственности реализация этих считающих семафоров даже в последних версиях Unix не позволяет использовать их для управления взаимодействием нитей. Отсюда возникла потребность в программных семафорах, которые возможно ближе выполняли бы функции абстрактных семафоров Дейкстры, но были бы пригодны и для согласования работы нитей.

Такие семафоры включены в последние версии стандарта POSIX на ОС Unix, причем это одно из самых поздних добавлений в него. Напомним, что традиционно ОС Unix была ориентирована на широкое использование процессов, но вначале не использовала нити, тем более не предоставляла программисту их использование.

В качестве средства организации взаимодействия нитей, предоставляющих все возможности абстрактных считающих семафоров и в то же время достаточно

простых для использования, введены семафоры, описанные в заголовочном файле semaphore.h. Все они имеют в качестве префикса имени буквосочетание sem_ . Тип данных рассматриваемых семафоров обозначается sem_t. Для создания таких семафоров предназначена функция с прототипом

int sem_init(sem_t *sem, int pshared, unsigned int value),

Эта функция при удачном выполнении возвращает семафор, адрес для которого задан первым ее параметром. Второй параметр предполагается в дальнейшем использовать для задания, будет ли данный семафор применяться для взаимодействия между процессами, а не только между нитями внутри одного процесса. Для использования семафора в пределах только одного процесса, этот параметр должен быть равен 0. Заметим, что использование данного параметра со значением, не равным нулю, вызывает ошибку выполнения функции (с соответствующим кодом возврата), если возможность согласования различных процессов с помощью таких семафоров не реализована в конкретной системе (такая ситуация имеет место в Linux). Последний параметр (параметр *value*) предназначен для задания начального значения семафора в ходе его создания.

По завершению использования рассматриваемый семафор следует уничтожить вызовом функции sem_destroy с прототипом

int sem_destroy(sem_t *sem),

где единственным аргументом служит указатель на созданный ранее семафор.

Операции P(sem) и V(sem) для данных семафоров задаются программными функциями с прототипами

int sem_wait(sem_t *sem) и

int sem_post(sem_t *sem).

Первая из этих функций приводит к блокировке нити, выполнившей ее, если внутреннее значение семафора уже равно нулю, и уменьшению на 1 внутреннего значения семафора – в ином случае. Функция sem_post() предназначена для деблокирования нити, ранее остановленной по доступу к семафору, или для увеличения внутреннего значения семафора, если нитей, блокированных по доступу к этому семафору, нет.

Дополнительно предлагается еще две программных функции с прототипами

int sem_trywait(sem_t *sem),

int sem_getvalue(sem_t *sem, int *value).

Первая из них подобна функции sem_wait, но не выполняет приостановку нити, когда семафор, указанный ее параметром, занят. Но в этом случае она возвращает ненулевой код возврата (в остальных случаях этот код – нулевой). Заметим, что само ненулевое значение кода возврата этой функции в указанной ситуации задается символической константой EAGAIN. Функция sem_getvalue() возвращает текущее – на момент запроса – внутреннее значение семафора с помощью своего второго параметра.

Упражнения

Разработать многопоточную программу, отображающую на экране взаимодействие трех нитей "читателей" из общей области данных и двух "писателей", записывающих в этот буфер данные. Буфер предназначен для хранения 10 символов. Первая нить-писатель выводит в буфер данные в латинском алфавите, вторая нить-писатель выводит в буфер данные в русском алфавите. Такой вывод эти две нити осуществляют в два приема, первый из которых записывает половину своего текста без завершающего этот промежуточный текст нуля. Между такими половинами вывода нити производят задержку на случайную величину миллисекунд, но не более 2 с. После вывода своего текста в буфер каждая нить-писатель переходит в ожидание порядка 2–3 с. до следующей попытки записи в буфер. Нити-читатели через случайный интервал порядка 300 мс. читают данные из буфера, если это позволяют средства синхронизации доступа между нитями, и выводят прочитанный текст на экран, каждая в свой столбец. Каждый вывод нити-читателя осуществляется в новую строку своего столбца, поэтому суммарные действия вывода в таких нитях предусмотреть только для 20–24 строк. Синхронизацию осуществить с помощью семафоров. (Выполнить два варианта разработки – для Windows и для Linux).

10. УПРАВЛЕНИЕ ПАМЯТЬЮ

10.1. Виртуальная память

Память является важнейшим после процессора ресурсом компьютера. Поэтому ее распределению – в простейших случаях, и управлению – в современных ОС уделяется пристальнейшее внимание.

Оперативная память по своей сущности – физический объект и поэтому существенно зависит от организации работы процессора с памятью. Иначе говоря, непосредственно зависит от особенностей архитектуры процессора. До настоящего времени наибольшее распространение в качестве процессоров компьютеров получили различные модификации процессоров с архитектурой типа Intel386, называемой также архитектурой IA32. Особенностью ее являются 32-битные *адреса* ячеек памяти, каждая из которых хранит отдельный байт. Таким образом, можно обозначать для автоматического аппаратного доступа (что называется *адресовать*) $2^{32} = 4$ Гбайт памяти. Это составляет около 4 млрд. таких ячеек, точнее 4 294 967 296. Такое огромное количество ячеек памяти редко бывает установлено в большинстве компьютеров, а отражает принципиальную возможность процессора по манипулированию такими ячейками.

Тем не менее, современным многопрограммным компьютерам и такого большого количества оперативной памяти оказывается недостаточно, если выпол-

няется много программ одновременно. Тем более может оказаться недостаточным и тех 256 Мбайт или несколько более того, как было характерно на момент написания пособия. Такая недостаточность оперативной памяти послужила одной из причин конструктивного построения *виртуальной памяти*. Другой, и более существенной, хотя не столь простой и очевидной причиной явилась необходимость глубокой изоляции программ и данных для различных процессов, одновременно выполняющихся компьютером. Такая изоляция программ и данных, принадлежащих одному процессу, от возможных изменений другим, называется *защитой памяти*.

Ошибки в прикладной программе без описанной взаимной изоляции приводят к ошибочному функционированию не только самой этой программы, но и любых других программ, одновременно выполняемых с нею. Более того, области оперативной памяти, используемые операционной системой, совершенно необходимо защищать от возможного, в том числе ошибочного доступа со стороны прикладных программ. Все эти соображения привели (еще в 1959 г.) к созданию аппаратно-программных средств виртуальной памяти. Эта память требует глубоко встроенных в аппаратуру процессора внутренних средств, которые в совокупности со служебными структурами данных, создаваемыми и поддерживаемыми программно, создают видимость для отдельной программы полного владения ею всей совокупностью имеющихся ячеек физической памяти. Эта видимость называется также *страничной переадресацией*.

Чтобы объяснить существо страничной переадресации, используем такой пример. Пусть аналог адресуемой программной ячейки есть адрес "квартиры" в некоторой вымышленной совокупности "домов", расположенных на одной очень длинной "улице". Пусть такие "дома" одинаковые и имеют по 1000 "квартир" с номерами от 0 до 999. На вымышленной улице может находиться до 1000000 таких "домов". Пусть все эти дома пронумерованы числами от 0 до 999999 в их естественном порядке месторасположения вдоль улицы. Тогда естественный физический номер квартиры можно задавать девятизначным десятичным номером, в котором три младшие цифры, в действительности, являются номером квартиры внутри дома (обычная нумерация квартир), а шесть старших цифр – номером дома на этой улице.

Реальный доступ по адресу, как известно, осуществляется не столько путем подсчета последовательной позиции дома вдоль улицы, сколько по адресным табличкам. Значит, принципиально можно развесить адресные таблички с номерами домов в некотором произвольном порядке (лишь бы были средства быстрого поиска по ним.)

Так вот, адресная переадресация в виртуальной памяти аналогична некоторому развешиванию адресных табличек по подобным домам, где функцию развешивания адресных табличек берет на себя операционная система. Причем для каждо-

го вычислительного процесса используется своя подсистема развешивания адресов на аналоги домов, и адресные таблички одного процесса совершенно не видны другому процессу. Условно говоря, адресная табличка с номером 5 для одного процесса обозначает реальный дом (совокупность ячеек), на котором для другого процесса вообще не видно никаких адресных табличек, но для последнего есть свой – другой дом, на котором этому другому процессу видится табличка с номером 5. Добираясь до любой квартиры дома с адресом 5, эти процессы попадают в совершенно различные физические объекты.

Каждому процессу для такой переадресации необходима своя система замены видимых для него адресных табличек на действительные физические места расположения, причем система – незаметная самому процессу. В качестве такой системы не придумано ничего проще, чем таблицы *страниц*. Страницей в данной переадресации называют аналог дома в нашей вымышленной модели. Номера внутри страницы обеспечивают нумерацию ячеек, находящихся в пределах одной страницы, аналогично номерам квартир внутри дома. Таблица страниц для процесса в общем случае содержит информацию для перехода от номеров страниц, видимых процессу, к номерам реального размещения этих страниц в оперативной памяти. В простейшем, точнее упрощенном случае, строка такой таблицы с номером k содержит всего лишь номер реального размещения этой страницы в оперативной памяти. Когда для нашего примера видимый процессу обобщенный номер квартиры записывается десятичными цифрами как 007345804, то это обозначает квартиру номер 804 в доме с табличкой для процесса, обозначенной 007345. Если в таблице переадресации для данного процесса в строке с номером 007345 стоит число 017421, то в действительности за указанным обобщенным номером стоит квартира 804 в доме с физическим номером расположения 017421, т.е. физический адрес 017421804.

В более ранних компьютерных архитектурах, которые содержали не очень большое число адресуемых в командах ячеек памяти, таблицы переадресации строили непосредственно с числом строк, равным числу возможных страниц. В нашем примере получается, что нужно иметь таблицу переадресации с 1000000 строк. Такая величина для вспомогательной информации даже в современных компьютерах достаточна велика или, по крайней мере, очень чувствительна.

Поэтому разработчики архитектуры IA32 пустились в свое время на хитрость. Аналог номера дома они разбили на две равные части. Старшая часть стала определять номер строки в таблице *каталога* страниц. Каждый элемент последней таблицы содержит действительный адрес (местонахождение) соответствующей таблицы страниц, так что теперь имеется не одна таблица страниц, а множество. В архитектуре IA32 может быть до 1024 таблиц страниц, в нашем вымышленном примере их должно быть до 1000. В свою очередь, каждая таблица страниц служит

для преобразования в физический номер страницы (физический номер страницы принято для сокращения называть *страничным кадром*).

В нашем примере старшие три десятичные цифры обобщенного номера квартиры задают порядковый номер строки в таблице каталога. В аналоге этой строки аппаратура компьютера берет информацию о местонахождении соответствующей таблицы страниц. В последней таблице берется та строка, номер которой задается следующими тремя цифрами десятичного номера из обобщенного номера. Эта строка дает физический номер квартиры. Например, если виртуальный обобщенный номер квартиры для процесса есть 007345804, то из таблицы каталога выбирается строка с номером 007. Из этой строки находится местонахождение таблицы страниц и в последней берется строка с номером 345, которая пусть содержит физический номер 201745. К последнему номеру приписываются три младшие десятичные цифры исходного обобщенного номера и получается физический обобщенный номер 201745804. В компьютере все перечисленные действия с частями обобщенного номера выполняет аппаратура процессора, причем эти действия осуществляются при выполнении каждой команды (существуют достаточно сложные аппаратные методы ускоренного выполнения этих действий, но на текущем уровне изложения они не существенны).

Отличие страничного преобразования в архитектуре IA32 от предложенного упрощенного изложения состоит в следующем. Виртуальный и физический адреса задаются не в десятичных цифрах, а в двоичных кодах. Эти адреса, как уже указывалось, 32-битные. В качестве номера строки таблицы каталогов берется 10 старших битов этого адреса, что дает 1024 строки этой таблицы. В качестве номеров строк таблиц страниц используются следующие десять битов за десятью старшими битами (биты с номерами от 21-го до 11-го). Оставшиеся двенадцать младших битов задают номер байта внутри страницы.

Заметим, что каждый элемент таблицы каталога и таблицы страниц содержит по 4 байта. В действительности, только часть битов из этих строк используется для хранения указанных преобразований (младшие 12 битов из этих строк служат для других дополнительных целей, часть которых служит для ограничения доступа к информации в страницах). Это ограничение доступа заключается в том, что каждой странице, описанной строкой в таблице страниц, присваиваются атрибуты доступа. Такие атрибуты записываются отдельными битами непосредственно в строках таблиц страниц. В архитектуре IA32 эти возможности оказались достаточно скромными и представляют собой возможность указания доступа либо по чтению, либо еще и по записи. В других архитектурах возможным является отдельно задаваемый атрибут доступа по выполнению (в IA32 доступ к странице памяти только по чтению и доступ только по выполнению считаются равносильными).

Если подсчитать, каков получается объем таблиц для страничного преобразования только одного процесса, то получим 4 Кбайт для таблицы каталога и 4

Мбайт для его всевозможных таблиц страниц. Это, конечно, не очень мало. Все рассмотренные построения с двухуровневым преобразованием, на самом деле, были заложены в предположении, что не вся теоретическая память может потребоваться каждому из выполняемых процессов. В таком случае целесообразно для ОС предоставить процессу пользоваться только некоторыми диапазонами виртуальных адресов, а для остальных не строить, не хранить и не использовать таблиц страниц. Строки таблицы каталога, соответствующие этим диапазонам, можно отметить внутри таблицы как не используемые (вот зачем потребовались зарезервированные биты в строках таблицы каталога).

Заметим, что в старших моделях архитектуры IA32, начиная с Pentium II, кроме рассмотренной схемы страничной переадресации возможна еще одна, когда размер страниц аппаратно выбирается равным 1 Мбайт. Хотя для некоторых программ такая величина дискретности выделения памяти оказывается достаточно расточительной, но она позволила в той же 32-битной архитектуре иметь всего 4096 страниц, таблица одноуровневого страничного преобразования для которых требует всего-то 16 Кбайт. Еще раз отметим, что последнее решение также не является панацеей, потому что так называемая подкачка страниц, о которой будет идти речь далее, в этом случае требует перемещения каждый раз по 1 Мбайту между оперативной и внешней памятью, что не может ни сказаться на быстродействии компьютерной системы.

10.2. Подкачка страниц для реализации виртуальной памяти

Кроме страничной переадресации, обеспечиваемой архитектурными средствами в совокупности с ОС, для функционирования виртуальной памяти используют так называемую *подкачку страниц*. Существо подкачки страниц в том, что часть внешней памяти используется ОС как средство временного хранения менее нужных страниц оперативной памяти, причем такое временное сохранение и восстановление информации из внешней памяти в оперативную организуется совершенно незаметно для вычислительных процессов. Практически единственное, что в этом методе (временной подмене части оперативной памяти внешней) становится заметным, так это замедление действий процесса, который ОС вынужденно приостанавливает для такого восстановления информации.

Чтобы читателю представить происходящее при подкачке страниц, используем уже применявшийся пример с вымышленной последовательностью домов и квартир. Как уже говорилось, физическую последовательность ячеек хранения использует не один процесс, а одновременно несколько. Поэтому им может просто не хватить всех физических ячеек хранения, когда каждый из них претендует на использование всей пронумерованной последовательности.

Техническое решение проблемы основывается на том, что не все страницы виртуальных адресов нужны одновременно процессу. Все реальные программы содержат множество циклов, а по ходу выполнения цикла используется относительно небольшой участок машинных кодов, даже если при этом выполняются обращения к некоторым подпрограммам. Поэтому значительной частью машинного кода исполняемого файла, а также частью области промежуточных данных можно временно "пожертвовать", позволив на какое-то время перенести их во внешнюю память. Более того, исследования показали, что заметная часть машинных кодов программ в большинстве запусков этих программ совсем не используется. Это относится к участкам обработки ошибок и действиям по редко выбираемых пунктов меню. Поэтому целесообразно эти участки исходного файла даже и не переносить в оперативную память.

Внешнюю память в нашем вымышленном примере можно рассматривать как своего рода "палаточный лагерь" очень большой вместимости, куда можно временно "отселять" содержимое "домов", потребовавшихся для размещения данных некоторому процессу. Проще представить, что какому-то процессу потребовалась новая страница для размещения его информации (не будем сейчас уточнять, потребовалась ли для новых данных или для участка уже используемой программы), но все дома на нашей выдуманной улице заняты. Тогда ОС ищет тот дом, который ни один из процессов долго не использовали для своей работы, и выселяет его содержимое в палаточный лагерь, записывая в соответствующую таблицу страниц информацию о новом местонахождении содержимого дома с одновременной отметкой в этой таблице, что дом (страница) находится во внешней памяти. Место для новой страницы, требовавшееся в начале наших действий, теперь свободно, и ОС помещает в этот дом требуемое содержимое, взятое из внешней памяти (из исполняемого файла, места временного хранения – палаточного лагеря, если нужная страница хранилась в нем и т.п.). После такого перемещения в таблице страниц, по которой к нужной странице данных требуется доступ, запоминается адрес действительного размещения нужной страницы. Перейдем на более технический стиль.

Пусть выполняется некоторая машинная команда, в которой задается обращение к некоторому месту оперативной памяти с адресом XXXYYYZZZ, где буквы X условно обозначают старшие биты адреса, задающие строку каталога страниц, буквы Y – средние биты адреса, задающие строку таблицы страниц в том каталоге, который определяется старшей частью, а буквы Z – младшие биты, задающие номер байта внутри страницы. Аппаратура процессора находит строку с номером XXX в каталоге страниц для текущего процесса и анализирует его содержимое. Может оказаться, что весь диапазон виртуальных адресов, начинающихся битами со значением XXX, не выделен процессу, тогда возникает ошибка по доступу к памяти. Если же (в большинстве случаев) данный диапазон выделен процессу, то в

этой строке содержится адрес соответствующей таблицы страниц. Аппаратура извлекает из этой таблицы строку с номером YYY и анализирует ее. Возможны два варианта.

Если в строке (соответствующим битом) указано, что требуемая страница находится в физической памяти, то из этой строки извлекается номер физической страницы, соотносимой виртуальной странице. К последнему номеру в двоичном коде приписываются младшие биты, задаваемые номером ZZZ, и полученное значение адреса используется непосредственно аппаратурой для доступа к оперативной памяти.

Если же в анализируемой строке таблицы страниц соответствующим битом указано, что страница не находится в физической памяти, то в ней присутствует информация, где же во внешней памяти находится содержимое этой страницы. Такая ситуация вызывает прерывание, называемое *страничным прерыванием*. Обрабатывая его, операционная система находит место в последовательности физических страниц, содержимое которых по ее алгоритмам можно временно вытеснить во внешнюю память. Система производит это перемещение, делает изменение в той строке таблицы страниц, которая ранее описывала вытесненное содержимое (возможно в таблице страниц другого процесса). Затем она переносит нужное содержимое страницы из внешней памяти в выбранную страницу и корректирует исходную строку таблицы страниц. По завершении прерывания вычислительный процесс возвращается к началу выполнения той исходной команды, которую начали рассматривать. Со второй попытки выполнения той же команды уже обнаруживается конкретное значение физической страницы, куда требуется доступ, и обращение осуществляется, как описано выше.

Такое перемещение содержимого страницы для области адресов, используемых в программе вычислительного процесса, и называют подкачкой страниц. В ее реализации операционной системой могут применяться достаточно сложные и "хитрые" алгоритмы, решающие, какой страницей физической памяти временно пожертвовать для размещения страницы, действительно необходимой на текущем шаге процесса. Эти алгоритмы включают тонкие средства самой архитектуры (аппаратно устанавливаемые биты в таблицах страниц и аналогичных системных структурах данных). В данном изложении эти вопросы рассматривать не будем. Остановимся лишь на более воспринимаемых проблемах и их решениях.

Область внешней памяти, служащая для автоматического сохранения содержимого страниц, которые временно удаляются из оперативной памяти, называют *областью свопинга* или *свопинг-файлом* (SWAP). От ее размера зависит, насколько много действительных данных могут использовать процессы и как много процессов можно запустить одновременно. Область свопинга в некоторых ОС задается фиксированным разделом на жестком диске. В ОС типа Windows эта область имеет размер, настраиваемый при тонкой регулировке.

С учетом изложенного встает вопрос, какие страницы виртуальной области адресов следует назначать непосредственно в физические страницы оперативной памяти при запуске программы? Если весь исполняемый файл большой программы переносить в оперативную память, то может оказаться, что часть этой работы выполнена напрасно: некоторые участки файла так и не будут использоваться. Кроме того, передача данных из внешней памяти в оперативную небольшими кусками (отдельными страницами) занимает больше времени, чем переча сразу большого последовательного участка файла. Поэтому Windows например, производит упреждающее чтение блоками размером до 64 Кбайт. Эти соображения с учетом некоторых других привели к следующим решениям.

В современных форматах исполняемых файлов (с обычными в Windows расширениями EXE и DLL) содержится доступная извне информация о частях (секциях) этих файлов. Так, в стандартном формате COFF (Common Object File Format) имеются секции с именами. Для каждой такой секции при разработке программы можно задать комбинацию атрибутов Read, Write, Execute и Shared. Если какая-нибудь секция такого исполняемого файла не имеет атрибута Write или Shared, то ее содержимое неизменяемо. Поэтому страницы виртуальной памяти процесса, использующие эту секцию, не нужно сохранять в области свопинга. Когда содержимое таких страниц потребуется процессу, их следует загрузить в страничный кадр оперативной памяти со своего исходного места в исполняемом файле. Такой подход дает возможность резко сократить реальную потребность в большом файле свопинга.

Виртуальная память позволяет выполнять программы, размер которых значительно больше размера имеющейся физической памяти. Это достигается за счет того, что из исходного файла в оперативную память переносятся только те страницы, которые действительно необходимы на текущий момент (и, может быть – некоторое число дополнительных страниц по упреждающему чтению). Остальная часть исходного файла используется путем подкачки страниц по действительному запросу на них, причем часть ранее используемых страниц обычным образом вытесняется в свопинг-файл или отмечается как неизменно присутствующая в области хранимого исполняемого файла.

10.3. Системные функции распределения памяти

Кроме незаметного для выполняемой программы использования виртуальной памяти, современные ОС содержат средства явного получения заказываемых объемов памяти. Напомним, что в языках программирования высокого уровня большая часть данных используется с помощью имен. Место в памяти для данных сопоставляется с именем транслятором. (Компиляторы выполняют только предварительное сопоставление, которое далее уточняется компоновщиком, а затем, воз-

можно, еще и загрузчиком программы.) Тем не менее, в современных языках программирования есть возможность использовать область памяти, задаваемую уже на стадии выполнения. Для этих целей служат именованные указатели, которые можно настроить на почти любое место в оперативной памяти (точнее на место, доступное по правам чтения или записи). Операционные системы содержат системные средства, которые позволяют запрашивать для работы программы дополнительные области данных, причем для последующего доступа к этим областям системная функция возвращает значение указателя, которое может быть запомнено в именованном указателе.

Проще всего такие средства организованы в Unix. Здесь имеются всего четыре системные функции, связанные с выделением и распределением оперативной памяти, называемые malloc, free, calloc и realloc. Первая из них имеет прототип

void* malloc(size_t nbytes).

Единственным аргументом этой функции служит число байтов в запрашиваемой области данных, а возвращается указатель на не детализированный тип данных. При невозможности выделить операционной системой заданный объем памяти, эта функция возвращает нулевой указатель (NULL). Данная функция имеет полные аналоги в системах программирования на языке Си для практически всех операционных систем, и использование ее обычно хорошо знакомо любому профессиональному программисту.

Как только запрошенная функцией malloc() область памяти становится ненужной программе, настоятельно рекомендуется отдать приказ об ее освобождении. Такое решение следует применять даже для персональных и "сильно накрученных" компьютеров, так как оперативная память очень важный ресурс, и ее недостаток может существенно осложнить работу других программ. Для этого освобождения предназначена функция с именем free. Она имеет прототип

void free(void* ptr),

где в качестве аргумента *ptr* необходимо применять именно то значение, которое было получено предыдущим вызовом функции malloc() или аналогичных функций calloc() и realloc(), рассматриваемых далее. Попытки использования в этой функции значений, не полученных от указанных функций, является серьезной ошибкой.

Функция calloc имеет прототип

void* calloc(size_t nelem, size_t nbytes)

и позволяет выделять область данных для размещения массива из *nelem* элементов, каждый из которых занимает *nbytes* байтов. (Практически эта функция сводится к соответствующему вызову исходной функции malloc.)

Наконец последняя из перечисленных функций имеет прототип

void* realloc(void* oldptr, size_t newsize).

Она используется для изменения размера дополнительной области памяти, ранее запрошенной у ОС и выделенной ею. При выполнении этой функции исходная область памяти может быть перемещена операционной системой в новое положение, и возвращаемое значение уже не будет совпадать со старым значением *oldptr*. Существенно, что при изменении положения области данных сохраняется содержимое той ее части, которая соответствует меньшему из старого и нового размеров.

В операционных системах OS/2 и Windows совокупность основных функций распределения памяти гораздо сложнее. В ней отразилась структурная сложность страничной переадресации для архитектуры IA32, которая построена как двухуровневая и рассматривалась в начале этой главы.

Практическое получение памяти программой, запрашивающей у ОС дополнительную область памяти (блок памяти), включает два этапа. На первом этапе у ОС запрашивается диапазон адресов виртуального адресного пространства, для которого ОС должна построить таблицы страниц в памяти и заполнить строки таблицы каталога. На этом этапе память выделяется только для хранения таблиц страничного преобразования (в лучшем случае – оперативная, но временно таблицы страниц могут находиться и во внешней памяти – в области свопинга). На втором этапе память выделяется уже непосредственно для запрошенного блока памяти, и при этом ОС заполняет соответствующие строки ранее построенных таблиц страниц текущего процесса. Вся эта сложность возникает от громоздкости суммарных таблиц страничного преобразования и благородной цели – по возможности сократить этот объем. Альтернативой было исходное построение всей теоретически возможной совокупности таблиц страниц для процесса (более 4 Мбайт на процесс) или перестраивание совокупности таблиц страничных преобразований почти при всяком запросе на новый блок данных. Первый вариант расточителен по расходу памяти, второй – по времени выполнения.

Для первого из перечисленных этапов сложилось название *резервирование памяти в адресном пространстве* или просто *резервирование памяти* (reserve). Второй этап обозначается английским словом commit. На русском языке пока не сложилось устойчивого наименования этого этапа. Его можно называть *передачей региону физической памяти* [13], что совершенно правильно, но очень длинно. Мы будем называть этот этап – *задействовать память* или *ввести в использование*.

В операционной системе OS/2 системные функции для выполнения перечисленных этапов явно разделены. Для этапа резервирования предназначена здесь функция с прототипом

APIRET DosAllocMem(void **ppmem, ULONG size, ULONG flag),

где *ppmem* – указатель на переменную, которая будет получать базовый виртуальный адрес выделенного блока памяти, *size* – заказываемый размер блока памяти в

байтах, а аргумент *flag* определяет атрибуты распределения и желаемого доступа. Он может задаваться с помощью следующих символических констант: PAG_COMMIT, PAG_EXECUTE, PAG_READ, PAG_WRITE, PAG_GUARD.

Константа PAG_COMMIT дополнительно задает выполнение второго этапа (введение в действие памяти) во время выполнения рассматриваемой функции. Остальные константы задают вид доступа к запрашиваемому блоку памяти и могут использоваться совместно как с первой константой, так и друг с другом. Константа PAG_GUARD задает особый вид доступа к блоку данных, предназначенному, как правило, для использования в качестве стека. При этом виде доступа обращение к последней странице блока памяти вызывает специальное исключение, обычно используемое программой для расширения области стека. При наличии флага PAG_COMMIT обязательно должен быть задан хотя бы один из флагов PAG_EXECUTE, PAG_READ, PAG_WRITE (первые два из них для архитектуры IA32 совершенно равносильны). Функция DosAllocMem выделяет целое число страниц оперативной памяти, это, в частности, обозначает, что при задании значения для параметра *size*, не кратного 4096 байтам, выделяется (в частности, резервируется) больше памяти, чем запрошено, а именно, до следующей границы, кратной 4096.

Для освобождения ранее выделенного блока памяти в OS/2 служит функция с прототипом

APIRET DosFreeMem(void* pb),

где аргумент задается указателем на ранее выделенный блок памяти, причем это значение должно быть получено от функции DosAllocMem.

Если для выполнения функции DosAllocMem не был задан флаг PAG_COMMIT, то после выполнения начального резервирования памяти необходимо выполнить системную функцию с прототипом

APIRET DosSetMem(void* pb, ULONG size, ULONG flag).

Здесь параметр *pb* задает значение указателя (виртуальный адрес) внутри ранее полученного блока памяти, аргумент *size* определяет размер вводимого в действие участка блока памяти, а последний аргумент задает виды доступа к этому участку и, возможно, изменение статуса диапазона адресов этого участка. Виды доступа задаются теми же константами PAG_EXECUTE, PAG_READ, PAG_WRITE, PAG_GUARD, которые уже рассматривались. Кроме того, для этих же целей допустима константа PAG_DEFAULT, применимая к участку, на котором ранее уже задавались виды доступа. Изменение статуса диапазона адресов участка задается одной из констант PAG_COMMIT, PAG_DECOMMIT. Первая из них вводит в действие участок адресов, задаваемый рассматриваемой системной функцией, а вторая выводит его из такого действия. Во втором случае физическая память, ранее выделенная участку адресов, отбирается и возвращается в резерв операционной системы.

Функция DosSetMem может воздействовать только на целое число страниц. Это значит, что если значение адреса виртуальной памяти, задаваемое параметром *pb*, не кратно 4096, то изменению атрибутов доступа и действия подвергается вся страница, внутри которой находится базовый адрес, задаваемый указателем *pb*. Если же значение адреса, получаемого от суммирования параметра *size* со значением базового адреса участка не кратно 4096, то задаваемому изменению атрибутов доступа и действия подвергается вся страница, внутри которой находится значение этого суммарного адреса. Кроме того, указанному действию подвергаются и все страницы, лежащие между двумя указанными адресами.

Действия двух рассмотренных системных функций в Windows может задавать универсальная функция с прототипом

VOID* VirtualAlloc(VOID *pvAddress, DWORD size,
DWORD type, DWORD protect).

Эта функция может выполнять как первый из перечисленных выше, так и второй этапы распределения памяти, но может выполнять и их оба одновременно. В ней аргумент *pvAddress* задает адрес виртуальной памяти, начиная с которого желательно выделить блок, называемый в этой системе *регионом*. Можно задавать этот параметр нулевым указателем, что равносильно пожеланию к ОС ею самой выбрать базовый адрес региона. Подчеркнем, что при ненулевом значении первого аргумента непосредственно в нем следует задавать адрес, а не указатель на адрес (адрес с точки зрения формализмов языка Си и является указателем). При ненулевом аргументе *pvAddress* и использовании функции распределения для резервирования региона значение этого аргумента должно быть кратно 64 Кбайт. Если функция выполняется успешно, то базовый адрес выделенного блока (региона) возвращается в виде значения функции. Параметр *size* задает желаемый размер запрашиваемого блока, причем следует иметь в виду, что память выделяется целым числом страниц, поэтому целесообразно для ясности самому программисту запрашивать резервирование региона размером в целое число страниц (иначе имеет место неявное выделение дополнительного участка за явно заказанным).

Параметр *type* может задаваться символическими константами MEM_COMMIT, MEM_RESERVE, определяя тем самым либо резервирование региона, либо введение региона в действие. Использование обеих констант, объединенных логической операцией ИЛИ задает обе эти операции при распределении памяти.

Аргумент *protect* определяет для региона тип защиты, запрашиваемый при вызове функции. Значением этого параметра может быть одна из символических констант PAGE_READONLY, PAGE_READWRITE, PAGE_EXECUTE (равносильна PAGE_READONLY). Дополнительно (объединением по операции ИЛИ) может быть указана еще и константа PAGE_GUARD.

Обратной функцией к рассмотренной служит функция с прототипом

BOOL VirtualFree(VOID *pvAddress, DWORD size, DWORD type),

где в качестве значения последнего из аргументов может быть использована одна из констант MEM_DECOMMIT, MEM_RELEASE.

Использование одной константы MEM_DECOMMIT возвращает из использования блок памяти (оставляя в памяти соответствующие таблицы страниц), а константа MEM_RELEASE отменяет и резервирование региона памяти, освобождая тем самым и память под соответствующими таблицами страниц. Возвращаемое логическое значение характеризует успешность выполнения функции.

В практической работе с перечисленными функциями распределения памяти очень полезными, особенно при отладке приложений, оказываются информационные функции запроса информации о блоках виртуальной памяти. Для OS/2 такая функция имеет прототип

APIRET DosQueryMem(void* pb, ULONG *size, ULONG *pflag),

где аргумент *pb* задает базовый адрес блока, о котором запрашивается информация, аргумент *size* определяет размер участка блока, для которого запрашивается информация, причем этот же параметр возвращает размер того участка, о котором выдается такая информация. (Если оказывается, что запрошена информация о достаточно протяженном участке, части которого имеют различные виды доступа или характер выделения, то информация выдается только о начальном из них вместе с размером такого участка в данном возвращаемом аргументе.) Параметр *pflag* задает адрес переменной, в которой возвращается информация об участке. Эта информация имеет битовое кодирование и может быть проанализирована с помощью символических констант: PAG_COMMIT (со значением 0x00000010), PAG_SHARED (со значением 0x00002000), PAG_FREE (со значением 0x00004000), PAG_BASE (со значением 0x00010000), PAG_READ (со значением 0x00000001), PAG_WRITE (со значением 0x00000002), PAG_EXECUTE (со значением 0x00000004), PAG_GUARD (со значением 0x00000008). Причем значение константы PAG_BASE обозначает, что заданный в вызове адрес отвечает самому началу выделенного региона, а PAG_SHARED – что память выделена для совместного использования различными процессами.

В операционных системах Windows аналогичная функция имеет прототип

DWORD VirtualQuery(void* pvAddress,

MEMORY_BASIC_INFORMATION *pmbiBuffer, DWORD size),

где первый аргумент также задает базовый адрес опрашиваемой области виртуальных адресов, а последний – размер этой области. Второй же параметр используется для возвращения экземпляра служебной структуры данных MEMORY_BASIC_INFORMATION для получения информации. Эта структура данных описывается в документации как

```
typedef struct _MEMORY_BASIC_INFORMATION {
        VOID *BaseAddress;          // base address of region
        VOID AllocationBase;        // allocation base address
        DWORD AllocationProtect;    // initial access protection
        DWORD RegionSize;           // size, in bytes, of region
        DWORD State;                // committed, reserved, free
        DWORD Protect;              // current access protection
        DWORD Type;                 // type of pages
    } MEMORY_BASIC_INFORMATION;
```

В поле *Protect* этой структуры отдельными битами возвращаются значения символических констант PAGE_READONLY, PAGE_READWRITE, PAGE_EXECUTE, PAGE_WRITECOPY, PAGE_GUARD, PAGE_NOACCESS. Последняя константа отражает состояние участка виртуальной памяти, при котором к нему нельзя обращаться ни для чтения, ни для записи, ни для выполнения программных кодов из этого участка. Из поля *state* можно получить присутствие в нем констант MEM_COMMIT, MEM_RESERVE или MEM_FREE. Наконец, поле *type* позволяет опросить информацию, задаваемую константами MEM_IMAGE, MEM_MAPPED, MEM_PRIVATE. (Функция возвращает число байтов в информационной структуре данных типа MEMORY_BASIC_INFORMATION.)

Дополнительно к только что рассмотренным функциям в Windows предлагается функция для изменения прав доступа к участку виртуальной памяти, имеющая прототип

```
BOOL VirtualProtect(void * pAddress,  DWORD size,
                    DWORD protect, DWORD *OldProtect).
```

В ней третий аргумент определяет новый вид прав доступа, задаваемый рассмотренными выше константами, а последний аргумент возвращает код старых прав доступа. Первый и второй аргументы задают адрес начала участка и его размер.

Использование всех перечисленных для Windows функций распределения памяти демонстрирует программа, приведенная в листинге 10.3.1.

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

void QueryMem(void *pmem, ULONG region);
void main()
{char *pm, *pm1, *pm2, ch;
 BOOL rc;
 int n;
```

```
DWORD oldProtect;

      pm=VirtualAlloc(NULL, 16000, MEM_RESERVE,
        PAGE_READONLY);  // RESERVE
      if (pm= =NULL) {printf("Error VirtualAlloc with RC=%ld\n",
                GetLastError()); getchar(); exit(0);}
      QueryMem(pm, 16384);    printf("After VirtualAlloc\n");

      VirtualAlloc(pm,1000, MEM_COMMIT, PAGE_READWRITE);
      if (pm= =NULL) {printf("Error Commit with RC=%ld\n",
                              GetLastError()); getchar(); exit(0);}
      QueryMem(pm, 16384);    printf("\nAfter Commit Mem\n");

      *(pm+999)='a'; printf("\nPause after writing into 999 offset.\n");
               getchar();
      *(pm+1000)='b';     printf("Pause after writing into 1000 offset\n");
               getchar();
      *(pm+4095)='c';     printf("Pause after writing into 4095 offset\n");
               getchar();

      rc=VirtualProtect(pm, 4096,PAGE_READONLY, &oldProtect);
      if (!rc) {printf("Error VirtualProtect with RC=%ld\n",
                              GetLastError()); getchar(); exit(0);}
      QueryMem(pm, 16384);    printf("\nAfter VirtualProtect to READONLY\n");
      ch=*(pm+999);printf("\nPause after reading from 999 offset ch=%c\n",
               ch); getchar();
      ch=*(pm+1000);
            printf("Pause after reading from 1000 offset ch=%c\n", ch); getchar();
      ch=*(pm+4095);
      printf("Pause after reading from 4095 offset ch=%c\n", ch); getchar();
      if (VirtualAlloc(pm+4096,1000, MEM_COMMIT,
               PAGE_READWRITE)= =NULL)
//можно было бы с адреса  pm+4000, но тогда обе page будут READWRITE !
        {printf("Error Commit with RC=%ld\n", GetLastError()); getchar();
               exit(0);}
      printf("\nAfter SetMem WRITE next page and READ on first\n");
      QueryMem(pm, 16384) ; printf("\n");
            QueryMem(pm+4096, 16384);

            *(int*)(pm+4096)= 137;
```

```
    printf("\nPause after writing word into 4096 offset.\n");
      getchar();
      n=*(int*)(pm+4096);
    printf("Reading from 4096 offset number=%d\n", n);
          VirtualFree(pm, 16384 ,MEM_RELEASE);  ExitProcess(0);
}

void QueryMem(void *pmem, DWORD region)
{DWORD rc;
 DWORD state, type, protect;
 MEMORY_BASIC_INFORMATION rec;
 void *pbase, *pregion;

rc=VirtualQuery(pmem, &rec, sizeof(rec));
state=rec.State;      protect=rec.Protect;
region=rec.RegionSize;
pbase=rec.AllocationBase;       pregion=rec.BaseAddress;
type=rec.Type;

      if (rc= =0) {printf("Error QueryMem with RC=%ld\n", rc); exit(0);}
      printf("Base address_=%08X,  region=%ld, BaseRegion=%ld\n",
        pbase, region, pregion);
      if (protect&PAGE_READWRITE) printf("PAGE_READWRITE ");
      if (protect&PAGE_WRITECOPY) printf("PAGE_WRITECOPY ");
      if (protect&PAGE_READONLY) printf("PAGE_READONLY ");
      if (protect&PAGE_EXECUTE) printf("PAGE_EXECUTE ");
      if (protect&PAGE_EXECUTE_READWRITE)
            printf("PAGE_EXECUTE_READWRITE ");
      if (protect&PAGE_EXECUTE_WRITECOPY)
            printf("PAGE_EXECUTE_WRITECOPY ");
      if (protect&PAGE_GUARD) printf("PAGE_GUARD ");
      if (protect&PAGE_NOACCESS) printf("PAGE_NOACCESS ");
      if (state&MEM_COMMIT) printf("MEM_COMMIT ");
      if (state&MEM_RESERVE) printf("MEM_RESERVE ");
      if (state&MEM_FREE) printf("MEM_FREE ");
      if (type&MEM_IMAGE) printf("MEM_IMAGE ");
      if (type&MEM_MAPPED) printf("MEM_MAPPED ");
      if (type&MEM_PRIVATE) printf("MEM_PRIVATE ");
}
```

Листинг 10.3.1. Распределение памяти в Windows

В последней программе вначале запрашивается резервирование региона памяти размером в 16 000 байтов, причем сразу задается, что доступ к этому региону будет только по чтению. Затем служебной подпрограммой, построенной на основе функции VirtualQuery, запрашивается информация об участке виртуальной памяти, начинающейся с возвращенного базового адреса, но охватывающего несколько больше запрошенного, а именно 4 страницы. Отображаемая подпрограммой информация свидетельствует, что действительно выделено больше чем 16000, а 16384 байта, т.е.- ровно четыре страницы.

Далее в программе перераспределяется часть уже распределенного региона, а именно с помощью той же универсальной функции VirtualAlloc задается введение в использование начального участка региона размером в 1000 байтов и с новым видом доступа по чтению и записи. Полученный результат опрашивается с помощью все той же служебной программы для того же большого интервала адресов в 4 страницы. Эта подпрограмма возвращает, как сама указывает, информацию только для участка памяти размером в 4096 байтов. Для этого участка выдается, что память введена в действие и имеет уже доступ и по записи. (Далее на большом запрошенном участке действуют уже другие права доступа и распределения.)

После этого в байты распределенного региона, имеющие смещение от его начала 999, 1000 и 4095, записываются данные. Несмотря на то, что и байт по смещению 1000 и тем более байт по смещению 4095 находятся за пределами участка в 1000 байтов, на котором явно задан доступ по записи и который, казалось бы, только и введен в действие, эти записи свободно осуществляются. Тем самым демонстрируется, что заказанные действия неизбежно распространяются на всю текущую страницу памяти.

Далее в программе функцией VirtualProtect меняется вид доступа к первой странице региона, а именно меняется на "только для чтения". Затем опрашивается состояние первых четырех страниц, получается информация только о первой, которая и введена в действие, несмотря на то, что и дальнейшие и эта страница имеют доступ только для чтения. Из записанных ранее байтов читается информация, что удается сделать, так как такой доступ разрешен и теперь.

Затем на 1000 байтов второй страницы устанавливается доступ по записи, и они вводятся в действие. Опрос с помощью служебной подпрограммы подтверждает, что состояние первой страницы региона не изменилось, а вся вторая страница введена в действие и имеет доступ и по записи. Для проверки в начало этой страницы записывается байт данных, который после этого читается и выводится на экран. Перед завершением программа освобождает весь регион, задавая константой MEM_RELEASE его полное освобождение.

10.4. Совместное использование памяти

В большинстве случаев отдельные вычислительные процессы не общаются друг с другом, выполняя самостоятельную и никак не связанную работу. Но существуют задачи, для решения которых создается не один процесс, а несколько, которые, работая совместно, выполняют общую работу. Такие процессы называются *кооперативными*.

Для взаимодействия процессов, в частности кооперативных, сложился ряд средств, часть которых – семафоры уже рассматривались, а некоторые будут изучаться в следующих главах. При всем их многообразии следует вспомнить, что основной информационный ресурс, используемый программами – оперативная память – это самый универсальный и быстрый ресурс компьютера. Поэтому не удивительно, что в состав всех современных ОС включены средства использования памяти для взаимодействия независимых процессов. Основой этих средств является *разделяемая память*, сами средства представляют набор системных функций использования этой памяти, а на более детальном уровне – и информационные средства описания этой памяти.

Изучение средств совместного использования памяти начнем с ОС Unix. Здесь разделяемая память находится под непосредственным управлением ядра, которое содержит таблицу описания областей разделяемой памяти. Каждая из областей обозначается в этой таблице целочисленным идентификатором (а не текстовым именем, как в других ОС). Кроме того, каждая такая область описывается в этой таблице атрибутами доступа и размером. Области разделяемой памяти относятся к адресному пространству ядра ОС.

Доступ к разделяемой памяти со стороны процесса осуществляется в два этапа. На первом из них получается хэндл области памяти, причем на этом этапе либо открывается доступ к уже имеющейся в ОС области памяти, либо такая область создается операционной системой. (Формально ситуация очень напоминает предварительные действия перед непосредственной работой с файлом.) На втором этапе процесс подключается к разделяемой области (to attach), используя ранее полученный хэндл. (Заметим, что сам термин *хэндл* в первоисточниках по Unix не используется, а применяется термин *идентификатор*, который в данном тексте действительно точнее. Мы же будем использовать термин *хэндл* для единообразного рассмотрения средств разделяемой памяти в различных ОС.)

На этапе подключения происходит подсоединение указанной области памяти, находящейся в ведении ОС, к виртуальному адресному пространству процесса, запросившего такое подключение. Результатом этой операции является базовый виртуальный адрес, начиная с которого в текущем процессе можно обращаться к разделяемой памяти.

Для получения разделяемой памяти предназначена функция с прототипом

int shmget(key_t key, int size, int flag),

возвращающая при удачном выполнении целочисленное значение требуемого идентификатора. Первым аргументом этой функции является уникальный номер (внешний идентификатор) области памяти, по своему смыслу аналогичный уникальному имени этого объекта. Вместо решения проблемы уникальности этого значения для новой области разделяемой памяти можно использовать специальную символьную константу IPC_PRIVATE, которая автоматически обеспечит создание внутри вызова уникального идентификатора исключительно для индивидуального пользования текущим процессом и его потомками. Кроме того, в качестве такого обозначения области разделяемой памяти может использоваться значение, полученное в другом процессе и как-то переданное в данный.

Аргумент *size* задает размер создаваемой по запросу области или ограничение по размеру на уже имеющуюся (больше, чем заданный размер по идентификатору, полученному от функции shmget, использовать будет нельзя). Аргумент *flag* в простейшем случае задается нулевым, но когда вызывающему процессу нужно создать область с заданным значением *key*, и области с таким идентификатором нет, то значение этого аргумента должно быть результатом логического сложения символьной константы IPC_CREAT и прав доступа по чтению и записи к новой области памяти. При неудаче функция возвращает значение -1.

Для подключения процесса к запрошенной ранее области разделяемой памяти служит функция с прототипом

void* shmat(int shmid, void* addr, int flag),

первый аргумент которой должен быть получен от предварительного вызова функции shmget, рассмотренной выше. Второй аргумент, обозначенный *addr*, задает виртуальный начальный адрес вызывающего процесса, в который следует отобразить разделяемую память. Для поручения ОС самой выбрать базовый адрес в виртуальном пространстве вызывающего процесса этот аргумент задается равным нулю (именно это значение рекомендуется для большинства ситуаций). Ненулевые значения следует выбирать только по крайней необходимости, например, когда в разделяемой памяти хранятся ссылки на адреса, что имеет место в связном списке.

При ненулевом аргументе *addr* последний аргумент может содержать флаг SHM_RND, приказывающий обнулить заданное значение базового адреса до границы страницы. Аргумент *flag* может также содержать флаг SHM_RDONLY, задающий режим доступа к подключаемой памяти только по чтению. При отсутствии такого флага доступ в дальнейшем будет осуществляться как по чтению, так и по записи. Функция в качестве собственного значения возвращает виртуальный адрес отображения разделяемой памяти, а при неудаче – значение -1.

Для отсоединения разделяемой памяти должна использоваться функция с прототипом

int shmdt(void* addr),

аргументом которой является адрес, ранее полученный от функции shmat. При успешном выполнении она возвращает 0, а при неудаче -1.

Кроме рассмотренных базовых функций для разделяемой памяти, в Unix имеется функция расширенного управления разделяемой памятью с прототипом

int shmctl(int shmid, int cmd, struct shmid_ds *buf).

Первым аргументом этой функции служит идентификатор, полученный от функции shmget. Собственно операции данной функции задаются вторым аргументом, который может задаваться константами IPC_STAT, IPC_SET и IPC_RMID. Первая из них задает операцию получения информации об области разделяемой памяти, вторая – изменение характеристик этой области (изменение прав доступа), а последняя служит для указания освобождения области разделяемой памяти. Третий аргумент используется только с операциями запроса информации и изменения характеристик, а для IPC_RMID задается значением NULL.

Следующий пример, приведенный в листинге 10.4.1a и 10.4.1b двумя исходными текстами программ для Unix, демонстрирует рассмотренное построение разделяемой памяти.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/stat.h>
#include <sys/shm.h>

void main()
{char *pm;
 int hmem;

      printf("Begin work\n");
      hmem=shmget(18011970, 16000, IPC_CREAT | 0600);
      if (hmem = = -1)
               {perror("Error AllocSharedMem with:"); getchar(); exit(0);}
      pm=shmat(hmem, NULL, 0);
      if (pm= =NULL) {perror("shmat"); exit(3);}
      strcpy(pm, "—");
      sleep(8);
      strcpy(pm, "Privet Shara !!!");        printf("Middle work\n");
      sleep(10);        printf("Two step\n");
      strcpy(pm+7, "- Good Bye !!!");        sleep(10);
```

```
        shmdt(pm);
        shmctl(hmem, IPC_RMID, NULL);
        exit(0);
}
```

Листинг 10.4.1a. Первая программа с разделяемой памятью в Unix

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/stat.h>
#include <sys/shm.h>

void main()
{char *pb, st[20];
 int hmem;
 int k;
        hmem=shmget(18011970, 16000, 0600);
        if (hmem= =-1)
                {printf("Error Shared Get Mem with\n"); getchar(); exit(0);}
              pb=shmat(hmem, NULL,  SHM_RDONLY);
        if (pb= =0)
                 {printf("Error Attach Shared Mem\n"); getchar(); exit(0);}
        for (k=0;k<10;k++ )
          {strncpy(st, pb, 20); st[19]='\0';    printf("%s\n", st);  sleep(2);}
        shmdt(pb);
        exit(0);
}
```

Листинг 10.4.1b. Вторая программа с разделяемой памятью в Unix

Вначале следует запускать программу, созданную из исходного текста листинга 10.4.1a, а затем уже программу, созданную из исходного текста по листингу 10.4.1b.

При использовании разделяемой памяти в Linux программисту предоставляется очень удобное системное средство – команда *ipcs*. Для получения информации о разделяемой памяти эту команду следует вызвать с опцией *m*, так что весь вызов имеет вид

```
ipcs -m
```

Эта команда выводит информацию о присвоенном ключе-идентификаторе, идентификаторе области памяти, владельце, размере и правах доступа к ней.

В операционной системах Windows для организации обмена через разделяемую память применяются функции, имеющие более универсальное назначение и относящиеся к отображению файлов на виртуальное адресное пространство. Основная из этих функций создает именованный или неименованный объект отображения в памяти и имеет прототип

```
HANDLE CreateFileMapping(HANDLE hFile,
        //хэндл файла или условное значение
   SECURITY_ATTRIBUTES *pFileMappingAttributes,
   DWORD protect, // protection for mapping object
   DWORD MaxSizeHigh,  // high-order 32 bits of object size
   DWORD MaxSizeLow,  // low-order 32 bits of object size
   CTSTR *pName);
```

Для построения области разделяемой памяти в первом аргументе этой функции должна задаваться специальная константа, равная -1 (обычно задается как (HANDLE)0xFFFFFFFF). Второй аргумент связан с атрибутами защиты и в простейших случаях берется равным нулевому указателю. Четвертый и пятый аргументы в совокупности задают 64-битное значение предельного размера отображаемого объекта, причем в аргументе *MaxSizeHigh* должны находиться старшие биты этого значения. Для 32-битных версий Windows этот аргумент можно спокойно брать равным нулю. Последний аргумент *pName* задает имя разделяемой области памяти в нашем текущем рассмотрении или имя области отображения в общем случае. Аргумент *protect* служит для задания видов доступа к используемой области виртуальной памяти и должен в нашем случае задаваться с помощью константы PAGE_READONLY или PAGE_READWRITE. При невозможности выполнения функция возвращает значение NULL, иначе она предоставляет значение хэндла, через который в дальнейшем возможен доступ к области виртуальной памяти. Заметим, что на этапе, обеспечиваемом рассмотренной функцией, получен только хэндл, но нет информации о базовом адресе виртуальной области памяти, которую с помощью этого хэндла можно использовать. Практически этот этап имеет много общего с действиями фунции shmget из Unix.

Базовый адрес разделяемой памяти получается для использования с помощью функции, имеющий прототип

```
void* MapViewOfFile(HANDLE hFileMappingObject,
                    DWORD DesiredAccess,
                    DWORD OffsetHigh, DWORD OffsetLow,  DWORD size),
```

которая и возвращает требуемый базовый адрес памяти или NULL при невозможности выполнения.

Здесь первый параметр должен быть взят от предыдущего вызова функции CreateFileMapping, предоставляющего хэндл объекта отображения. Третий и четвертый аргументы этой функции совместно задают 64-битное смещение внутри виртуальной области памяти, созданной вызовом функции CreateFileMapping. В большинстве применений это смещение берется нулевым. Аргумент *DesiredAccess* может задаваться константой FILE_MAP_WRITE или FILE_MAP_READ. Первая задает доступ как по чтению, так и по записи, вторая – только по чтению. Последний аргумент *size* задает размер разрешенного для дальнейшего использования диапазона виртуальных адресов с заданным видом доступа. Нулевое значение этого аргумента равносильно указанию на использование всего возможного диапазона адресов исходного объекта, созданного функцией CreateFileMapping.

После завершения использования разделяемой памяти для освобождения уже ненужных ресурсов следует вызывать функцию UnmapViewOfFile, имеющую прототип

BOOL UnmapViewOfFile(void* pBaseAddress),

передав ей в качестве аргумента базовый адрес разделяемой памяти. Кроме того, в общем случае следует закрыть хэндл объекта отображаемой памяти с помощью функции CloseHandle.

Следующий пример, приведенный в листинге 10.4.2a и 10.4.2b, демонстрирует применение разделяемой памяти для передачи данных между двумя процессами в Windows.

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

void main()
{char *pm;
 HANDLE hmem;

        hmem=CreateFileMapping((HANDLE)0xFFFFFFFF,NULL,
                               PAGE_READWRITE, 0, 16000, “SHAREMEM_MYY”);
   if (hmem= =0)
         {printf(“Error AllocSharedMem with RC=%ld\n”, GetLastError());
            getchar(); exit(0);}
   pm=MapViewOfFile(hmem, FILE_MAP_WRITE,0,0,0);
   if (pm= =NULL)
         {printf(“Error Mapping SharedMem with RC=%ld\n”, GetLastError());
```

```
            getchar(); exit(0);}
    Sleep(4000);
    strcpy(pm, “Privet Shara !!!”);     printf(“Middle work\n”);  Sleep(10000);
    printf(“Two step\n”);
    strcpy(pm+7, “- Good Bye !!!”);        Sleep(10000);
    UnmapViewOfFile(pm);
    CloseHandle(hmem);
}
```

Листинг 10.4.2a. Первая программа с разделяемой памятью для Windows

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

void main()
{char *pb, st[20];
 HANDLE hmem;
 int k;
      hmem=OpenFileMapping(FILE_MAP_READ, FALSE,
                “SHAREMEM_MYY”);
       if (hmem= =0)
                {printf(“Error OpenSharedMem with RC=%ld\n”, GetLastError());
                 getchar(); exit(0);}
       pb=MapViewOfFile(hmem, FILE_MAP_READ,0,0,0);
       if (pb= =NULL)
                {printf(“Error Mapping SharedMem with RC=%ld\n”, GetLastError());
                 getchar(); exit(0);}
       for (k=0;k<10;k++ )
          {strncpy(st, pb, 20); st[19]=’\0'; printf(“%s\n”, st);   Sleep(2000); }
       UnmapViewOfFile(pb);   CloseHandle(hmem);
}
```

Листинг 10.4.2b. Вторая программа с разделяемой памятью для Windows

Для демонстрации поведения программ следует вначале запускать программу, созданную из исходного текста в листинге 10.4.2a, а затем уже программу, созданную из исходного текста в листинге 10.4.2b.

10.5. Отображение файлов в оперативную память

Отображение файлов в оперативную память впервые появилось в BSD Unix. Идея такого отображения появилась как результат осмысливания избыточности действий при файловом вводе и выводе в операционной системе, использующей виртуальную память. Действительно, при вводе и выводе происходит пересылка данных между внешней памятью, хранящей файл, и служебными областями ОС (буферами ввода-вывода внутри адресного пространства ОС), и тут же неизбежная пересылка данных между буферами ОС и буферами данных прикладного процесса. Заметим, что использование системой ввода-вывода непосредственно пользовательских буферов оказалось бы нарушением правил защиты памяти с вытекающими отсюда неприятностями для функционирования системы. А двукратная пересылка данных заметно снижает быстродействие системы при работе с файлами.

В то же время сама система виртуальной памяти незаметным для процесса образом использует внутренние операции пересылки данных между оперативной памятью, воспринимаемой процессом как виртуальная память, и областями во внешней памяти. Требовалось только достроить внутренние средства, которые бы позволяли процессу автоматически отображать часть своего адресного пространства на отдельный файл, а не в область свопинга и оперативную память.

Использование в программах Unix файлов, которые отображаются в память, требует заголовочного файла mman.h, расположенного в подкаталоге sys стандартного каталога для заголовочных файлов, так что его подключение выполняется директивой

#include <sys/mman.h>

Для использования рассматриваемого отображения в Unix служит системная функция с прототипом

void* mmap(void* addr, size_t size, int prot, int flag, int fd, off_t pos),

где аргумент *fd* задается хэндлом уже открытого файла, который требуется отображать. Пожелание о значении виртуального адреса процесса, с которого предполагается отображение, задается аргументом *addr*. Если этот аргумент задается равным нулю, то сама система назначает виртуальный адрес начала области отображения, который всегда возвращается как значение функции при ее успешном выполнении. При неудаче возвращается значение -1.

Аргумент *size* определяет размер отображаемой области файла. Если его значение не кратно размеру страницы, то *size* байтов будут взяты из файла, а оставшаяся часть страницы заполнена нулями. Аргумент *pos* задает начальную позицию в файле, с которой начинается отображение, что дает возможность отображать не только весь файл, но и любой его последовательный участок. Параметр *pos* должен быть кратен размеру страницы.

Аргумент *prot* задает права доступа к отображаемой памяти и может представляться одной из следующих символьных констант: PROT_READ,

PROT_WRITE и PROT_EXEC. Аргумент флагов *flag* задает опции отображения и может представляться одной из констант MAP_SHARED, MAP_PRIVATE, MAP_FIXED. Последняя из них задает обязательность точного использования аргумента *addr* и возвращение ошибки при невозможности выполнить это указание. Константы MAP_SHARED и MAP_PRIVATE задают соответственно разделяемое несколькими процессами и сугубо индивидуальное использование области отображения файла в виртуальную память. Кроме того, следует иметь в виду, что при указании флага MAP_PRIVATE изменения в виртуальной области отображения файла не только не видны в других процессах, но и не записываются в файл (содержимое отображения файла используется как временный файл для текущего процесса). При флаге же MAP_SHARED все изменения в области отображения файла, представляющей его содержимое, тут же становятся видны другим процессам, открывшим этот же файл, а изменения в этой области записываются в файл на диске.

После завершения процесса все отображения файлов в память автоматически отменяются. Чтобы отменить отображение файла ранее завершения процесса, следует использовать вызов функции munmap, которая имеет прототип

int munmap(void* addr, size_t size).

Если при создании отображения был указан флаг MAP_SHARED, то в файл вносятся все оставшиеся изменения, при флаге MAP_PRIVATE все изменения отбрасываются. Следует иметь в виду, что функция munmap только отменяет отображение файла, но не закрывает файл, который требуется закрыть обычным вызовом close().

Следующий пример, приведенный программой в листинге 10.5.1, демонстрирует использование отображения файлов в память. В этой программе выполняется быстрое копирование файла, имя которого задано аргументом командной строки. Файл результирующей копии назван для упрощения задачи именем *mytemp*.

```
#include <stdio.h>
#include <sys/mman.h>
#include <fcntl.h>

void main(int argc, char **argv)
{ int  hin, hout;
  size_t fsize;
 void *source, *target;

        if (argc<2) {printf("Error format call program!\n"); exit(1);}
        hin=open(argv[1],O_RDONLY);
        if (hin = = -1) {printf("Error open input file\n"); exit(2);}
```

```
        hout=open("mytemp",O_RDWR | O_CREAT | O_TRUNC, 0644);
        if (hout = = -1) {printf("Error open output file\n"); exit(3);}
        fsize = lseek(hin, 0, SEEK_END);
        lseek(hout, fsize-1,SEEK_SET);
        write(hout,"!",1);
        source=mmap(0, fsize, PROT_READ, MAP_SHARED, hin,0);
        if (source = = (void*)-1) {printf("Error map input file\n"); exit(4);}
        target=mmap(0, fsize, PROT_WRITE, MAP_SHARED, hout,0);
        if (target = = (void*)-1) {printf("Error map output file\n"); exit(5);}
        memcpy(target, source, fsize);
        munmap(source, fsize);  munmap(target, fsize);
        close(hin); close(hout);
}
```

Листинг 10.5.1. Использование отображения файлов для быстрого копирования в Unix

В этой программе, чтобы правильно построить область отображения еще не существующего выходного файла, создается его фиктивное содержимое путем записи некоторого символа (выбран символ '!', но это не обязательно) в последнюю позицию файла исходя из предположения о его действительном размере. В противном случае при выполнении оператора target=mmap(. . . ,hout,0) будет создана пустая виртуальная область памяти с последующими неприятностями для дальнейшего выполнения программы.

Для собственного копирования одной области памяти в другую использована функция memcpy, которая имеет три аргумента, из которых первый дает адрес области, куда копировать, второй – адрес исходной области для копирования, а последний – число копируемых байтов.

В операционной системе Windows для отображения в память файлов используются уже частично рассмотренные в разделе 10.4 функции CreateFileMapping, MapViewOfFile и UnmapViewOfFile. Для отображения собственно файлов первый аргумент функции CreateFileMapping должен задаваться хэндлом предварительно открытого файла, третий аргумент этой функции *protect* по своим значениям должен соответствовать режиму использования файла, заданному при открытии последнего. (Если используется константа PAGE_READONLY, то файл должен быть открыт с доступом GENERIC_READ, а если используется константа PAGE_READWRITE, то при открытии файла должен был быть указан доступ константами GENERIC_READ и GENERIC_WRITE.) Напомним еще раз, что в Windows функция CreateFileMapping практически создает только область виртуальных адресов, подготовленную для последующего отображения. (Кроме того, ею создается объект ядра, который и служит информационной структурой даль-

нейшего управления и контроля за отображаемым объектом.) Собственно отображение (в отличие от Unix) здесь выполняется вызовом системной функции MapViewOfFile.

Эта последняя функция для файла должна задавать тот же режим доступа, что и указывалось на предыдущем этапе (присутствует заметная избыточность, не характерная для большинства профессиональных решений в области системного программирования, но характерная для продукции Microsoft). Практически две последние функции Windows выполняют в совокупности ту же работу, что делает в Unix единственная функция mmap. Единственным существенным отличием является возможность явного задания размера области отображения предпоследними параметрами функции CreateFileMapping. (Напомним, что эта же функция в Windows системах применяется и для побочных целей – первого этапа построения разделяемой памяти.)

После завершения работы в Windows с отображаемым в память файлом следует выполнить вызов функции UnmapViewOfFile для базового адреса области отображения и закрыть хэндл, полученный от функции CreateFileMapping. Кроме того, не следует забывать и закрывать файл, открытый для отображения.

Следующий пример, приведенный программой в листинге 10.5.2, демонстрирует решение в Windows той же задачи, что уже рассматривалась для Unix.

```
#include <windows.h>
#include <stdio.h>
void main(int argc, char **argv)
{ HANDLE  hin, hout;
  int fsize;
  void *source, *target;
  HANDLE hsource, htarget;
  DWORD actlen;

        if (argc<2) {printf("Error format call program!\n"); exit(1);}
        hin=CreateFile(argv[1], GENERIC_READ,
        FILE_SHARE_READ, 0, OPEN_EXISTING,
        FILE_ATTRIBUTE_NORMAL, 0);
        if (hin = = INVALID_HANDLE_VALUE)
{printf("Error open input file\n"); exit(2);}
        hout=CreateFile("mytemp", GENERIC_READ
|GENERIC_WRITE, FILE_SHARE_READ, 0,
                CREATE_ALWAYS, FILE_ATTRIBUTE_NORMAL, 0);
        if (hout = = INVALID_HANDLE_VALUE)
{printf("Error open output file\n"); exit(3);}
```

```
        fsize=SetFilePointer(hin, 0, 0, FILE_END);
        hsource=CreateFileMapping(hin,NULL,PAGE_READONLY,0,0,NULL);
        if (hsource = = NULL) {printf("Error CreateMap input file\n");
exit(4);}
        source=MapViewOfFile(hsource, FILE_MAP_READ,0,0,0);
        if (source = = NULL)
{printf("Error map input file\n"); exit(4);}
        htarget=CreateFileMapping(hout,NULL,PAGE_READWRITE,0,fsize,NULL);
        if (htarget = = NULL)
{printf("Error CreateMap output file\n"); exit(5);}
        target=MapViewOfFile(htarget, FILE_MAP_WRITE,0,0,0);
        if (target = = NULL) {printf("Error map output file\n"); exit(5);}
        memcpy(target, source, fsize);
        UnmapViewOfFile(source);  UnmapViewOfFile(target);
        CloseHandle(hin); CloseHandle(hout);
        CloseHandle(hsource);    CloseHandle(htarget);
}
```

Листинг 10.5.2. Использование отображения файлов для быстрого копирования в Windows

В этой программе, учитывая свойства функции CreateFileMapping, оказалось возможным обойтись без искусственного приема при создании области отображения для выходного файла. Именно последний параметр указанной функции позволяет явно задать размер области отображения вне зависимости от реального размера файла на текущий момент. В данной программе для этого параметра при создании отображения выходного файла взята длина исходного входного файла.

10.6. Динамически распределяемая память

При оперативной работе с динамическими структурами данных, не предназначенными для длительного хранения, возникает проблема эффективного использования памяти для хранения небольших структур данных. Системные функции распределения памяти в OS/2 и Windows предоставляют по запросу не менее страницы виртуальной памяти, что естественно расточительно для небольших объектов. Программист, конечно, может после запроса большого блока памяти взять на себя управление выделением из него небольших порций, но было бы удобней и надежней иметь для этого соответствующие системные средства. В основе этих средств лежит *динамически распределяемая память*, называемая также *пулом* памяти (pool).

В операционных системах Windows управление динамической памятью сделано весьма разнообразным. Во-первых, каждый процесс при его создании получает пул памяти по умолчанию, который обычно составляет 1 Мбайт. Это значение можно изменить при создании выполняемого EXE-файла с помощью соответствующей опции компоновщика. Если такого общего пула недостаточно при выполнении программы, то набор функций API Windows предоставляет возможности создания дополнительных пулов.

Основной из этих функций является функция с прототипом

HANDLE HeapCreate(DWORD options, DWORD size, DWORD maxsize),

где аргумент *maxsize* задает максимальный размер создаваемого пула для дальнейших его перераспределений, *size* – начальный размер пула, а параметр *options* задает режимы создания, которые в простейших случаях не используются. Функция эта возвращает хэндл созданного пула или NULL при неудаче. Созданный пул по исчезновению потребности в нем должен быть уничтожен функцией HeapDestroy с прототипом

BOOL HeapDestroy(HANDLE hHeap),

которая выводит из действия все страницы памяти, задействованные под пул, и освобождает диапазон виртуальных адресов, выделенных под него.

Для запроса порции памяти из пула служит функция с прототипом

void* HeapAlloc(HANDLE hHeap, DWORD flags, DWORD size),

где *hHeap* задает хэндл пула, *size* – размер запрашиваемой порции памяти, а *flags* задает символическими константами флаги выделения. Среди таких констант наиболее употребительная HEAP_ZERO_MEMORY, задает обнуление выделенного блока памяти. Функция возвращает виртуальный адрес начала запрошенной порции. Хэндл пула может быть получен от ранее выполненной функции HeapCreate или для пула по умолчанию – от функции GetProcessHeap, имеющей прототип

HANDLE GetProcessHeap(VOID).

Функция HeapAlloc аналогична по своему смыслу функции malloc из Unix, но, как видим, более сложна в использовании и привязана к Windows. Обратной ей по действиям (аналогичной функции free из Unix) служит функция HeapFree с прототипом

BOOL HeapFree(HANDLE hHeap, DWORD flags, void* pMem),

где *hHeap* задает хэндл пула, *pMem* – адрес начала порции, полученный ранее от функции HeapAlloc, а *flags* – флаги, которые чаще всего не используются, так что аргумент *flags* полагается равным нулю.

Дополнительные возможности изменения размеров блоков, полученных из пула, предоставляет функция HeapReAlloc, описываемая прототипом

void* HeapReAlloc(HANDLE hHeap, DWORD flags, void* pMem,DWORD size).

Она позволяет изменить размеры уже имеющегося блока (порции памяти) с адресом *pMem* из пула с хэндлом *hHeap* до нового размера *size*. Флаги, задаваемые в аргументе *flags*, указываются обычно константами HEAP_REALLOC_IN_PLACE_ONLY и HEAP_ZERO_MEMORY. Первая из них выдвигает требование изменить размер блока, оставляя его на собственном месте (не меняя его виртуального адреса), что выполнимо в большинстве случаев только для уменьшения размера блока. Вторая из этих констант требует заполнить нулями добавляемую часть блока при увеличении его размера.

Дополнительные возможности предоставляет функция с прототипом

DWORD HeapSize(HANDLE hHeap, DWORD *flags, void* pMem),

которая позволяет получить в качестве своего значения размер блока, выделенного по адресу *pMem* из пула с хэндлом *hHeap* (одновременно можно получить значение флагов этого блока).

Применение рассмотренных функций для работы с пулом памяти в Windows демонстрирует программа, приведенная в листинге 10.6.1.

```
#include <windows.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
void main()
{char *pblock1, *pblock2, *pblock3;
 HANDLE hheap;
        hheap=HeapCreate(0,15000, 60000);
        if (hheap= =0)
            {printf("Error HeapCreate with RC=%ld\n", GetLastError());
             getchar(); exit(0);}
        pblock1=HeapAlloc(hheap, HEAP_ZERO_MEMORY, 1);   //min= =12 !!
        strcpy((char*)pblock1, "piece1");
        printf("Block1 Adrress=%08X, its size=%ld\n", pblock1,
           HeapSize(hheap,0,pblock1));
        printf("Pause after writing into block1.\n");getchar();

        pblock2=HeapAlloc(hheap, HEAP_ZERO_MEMORY, 24);
        strcpy((char*)pblock2, "second piece");
        printf("Block2 Adrress=%08X, its size=%ld\n", pblock2,
           HeapSize(hheap,0,pblock2));
        printf("Pause after writing into block2.\n");getchar();
        pblock3=HeapAlloc(hheap, HEAP_ZERO_MEMORY, 27);
```

```
        pblock2=HeapReAlloc(hheap, 0, pblock2,47);
        printf("After realesing block2.\n");getchar();
        printf("Block2 Adrress=%08X, its size=%ld\n", pblock2,
           HeapSize(hheap,0,pblock2));
        printf("Contens of block1 = %s, its size=%ld\n", pblock1,
           HeapSize(hheap,0,pblock1));
        HeapFree(hheap, 0, pblock1);
        printf("Pause after realesing block1.\n");getchar();
        printf("Contens of block2 = %s, its size=%ld\n", pblock2,
           HeapSize(hheap,0,pblock2));
        HeapDestroy(hheap);
}
```

Листинг 10.6.1. Управление пулом памяти в Windows

Вначале программа создает новый пул памяти с возможным максимальным размером в 60 000 байтов и текущим размером в 16 000 байтов. После этого из нового пула функцией HeapAlloc запрашивается 1 байт памяти (а выделяется 12 – меньше не выделяется). Затем той же функцией запрашиваются порции по 24 и 27 байтов. Во все полученные таким образом блоки памяти записывается текстовая информация, затем второй блок изменяется в размере до 47 байтов, первый блок освобождается. В завершение созданный ранее пул уничтожается.

Для сравнения в листинге 10.6.2 приведена аналогичная программа работы с динамической памятью в Unix.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
void main()
{char *pm, *pblock1, *pblock2, *pblock3, *pblock2a;
        pblock1=malloc(7);
        strcpy((char*)pblock1, "piece1");
        printf("Block1 adrress=%08X\n", pblock1);
        printf("Pause after writing into block1.\n");getchar();

        pblock2=malloc(24);
        strcpy((char*)pblock2, "second piece");
        printf("Block2 adrress=%08X\n", pblock2);
        printf("Pause after writing into block2.\n");getchar();

        pblock3=malloc(27);
```

```
        strcpy((char*)pblock3, “third piece”);
        printf(“Block3 adrress=%08X\n”, pblock3);
        printf(“Pause after writing into block3.\n”);getchar();

        pblock2a=realloc(pblock2,47);
        printf(“After realloc pblock2a=%08X\n”, pblock2a);
        if (pblock2a!=NULL) pblock2=pblock2a;
        printf(“Block2 adrress=%08X\n”, pblock2);
        printf(“Contens of block1 = %s\n”, pblock1);
        free(pblock1);
        printf(“Contens of block2 = %s\n”, pblock2);
        free(pblock2);
        printf(“Contens of block3 = %s\n”, pblock3);
        free(pblock3);
}
```

Листинг 10.6.2. Управление пулом памяти в Unix

Непосредственное сравнение последних программ показывает, что API Unix дает самые компактные и простые для использования программистом средства управления памятью.

Упражнение

Разработать многопоточную программу, отображающую на экране взаимодействие трех нитей "читателей" из общей области данных и двух "писателей", записывающих в этот буфер данные. Буфер предназначен для хранения 10 символов. Первая нить-писатель выводит в буфер данные в латинском алфавите, вторая нить-писатель выводит в буфер данные в русском алфавите. Такой вывод эти две нити осуществляют в два приема, первый из которых записывает половину своего текста без завершающего этот промежуточный текст нуля. Между такими половинами вывода нити производят задержку на случайную величину миллисекунд, но не более 2 с. После вывода своего текста в буфер каждая нить-писатель переходит в ожидание порядка 2-3 с. до следующей попытки записи в буфер. Нити-читатели через случайный интервал порядка 300 мс. читают данные из буфера, если это позволяют средства синхронизации доступа между нитями, и выводят прочитанный текст на экран, каждая в свой столбец. Каждый вывод нити-читателя осуществляется в новую строку своего столбца, поэтому суммарные действия вывода в таких нитях предусмотреть только для 20 – 24 строки. Синхронизацию осуществить с помощью семафоров. (Возможны два вариант задания – для Windows и для Linux).

11. СРЕДСТВА КОММУНИКАЦИИ ПРОЦЕССОВ

11.1. Неименованные коммуникационные каналы Unix

Коммуникационные каналы (pipe) относятся к основным средствам межпроцессорных взаимодействий (IPC – Inter Process Communication). Эти каналы в русскоязычной технической литературе называются также каналами передачи данных. Возникли они в составе операционной системы Unix, где до сих пор играют большую неявную роль при построении командных конвейеров, широко используемых в программировании на уровне командных оболочек.

Канал передачи данных отчетливо ассоциируется с трубопроводами, которые передают данные между процессами. Такой канал (или "труба") имеет два конца, в один данные поступают, из другого выходят. С точки зрения программиста, использующего канал, чтение данных из него и запись данных в канал можно выполнять, используя обычные операции чтения из файла и записи в файл на основе хэндлов.

В ходе своего исторического развития каналы передачи данных "мутировали" и к настоящему времени мы имеем две их разновидности: неименованные каналы и именованные каналы. Первая разновидность до сих пор широко используется в Unix, где ею обычно и ограничиваются. В дальнейшей части текущего раздела будем называть для сокращения неименованные каналы просто каналами, оставляя уточнения для дальнейшего изложения.

По классическому решению в Unix канал после создания имеет два конца, один из них может быть использован только для записи, а другой – только для чтения. (Ряд версий Unix расширил такое понимание канала и теперь в общем случае можно использовать оба конца канала как для записи, так и для чтения, но делать так не рекомендуется, потому что подобные действия чрезвычайно усложняют для программиста процесс действий с каналом.) По внутреннему строению неименованный канал представляет собой кольцевой буфер в памяти, разделяемой процессами, причем этот буфер снабжен внутренними указателями. Один из них задает место, с которого можно далее писать данные, а второй – место с которого начинаются еще не считанные данные.

Для создания канала передачи данных в Unix предназначена функция с прототипом

int pipe(int hpipe[2]).

Аргументом этой системной функции служит адрес массива для двух хэндлов, младший элемент этого массива при удачном выполнении функции дает хэндл для конца канала, с которого можно считывать данные, а второй элемент дает при этом хэндл конца канала, в который можно записывать данные.

Для записи в канал используется функция write, а для чтения – функция read. Причем применение функции read к каналам имеет небольшую специфику, кото-

рая на самом деле присутствует и для аналогичной операции с обычными файлами, но там она не бросается в глаза. Дело в том, что функция read возвращает как собственное значение число реально прочитанных байтов. Для файлов отличие числа прочитанных байтов от числа запрошенных оказывается равносильным ситуации конца файла. Строго говоря, согласно документации, ситуацию конца файла определяет только возврат нулевого значения в качестве числа байтов, прочитанных функцией read чтения из файла. Но для обычных файлов после получения меньшего числа байтов чем запрошено, не существует альтернативы обязательному получению нулевого числа байтов в результате следующего выполнения функции чтения.

Для каналов же передачи данных вычислительный поток (нить процесса в общем случае) приостанавливается на функции чтения из канала до тех пор, пока канал функционирует, но данные еще не поступили. Если же в выходной конец канала поступило сколько-то байтов, то независимо от их числа функция чтения возвращает сколько возможно с учетом запрошенного. В частности, возвращает меньше байтов чем запрошено, но такое получение данных может не иметь никакого отношения к завершению данных в канале и возникновению ситуации конец файла. После чтения какого-то числа байтов при очередном запросе данных функцией read канал может выдать следующую порцию данных, в лучшем случае ровно столько, сколько запрошено, но может опять вернуть меньше запрошенного.

Можно сказать, что инженерной "изюминкой" работы с каналами передачи данных является предоставление каналом того числа байтов, которые уже поступили через канал, не дожидаясь поступления всей запрошенной партии байтов. Таким образом, при работе с каналами передачи данных ситуация конца передачи данных определяется возвращением нулевого значения функцией чтения.

Другим существенным моментом организации программистом работы с каналом является формирование ситуации конца передачи данных (конца файла для читающей стороны). Таким средством является принудительное закрытие канала со стороны конца, в который происходит запись данных. С этой стороны выполняется операция close(*хэндл_конца_записи*).

Следующая программа, приведенная в листинге 11.1.1, демонстрирует использование канала передачи данных в Unix. Для такой демонстрации создается два вычислительных процесса на основе одной общей программы. Процесс-родитель, остающийся от исходного процесса после выполнения системного вызова fork(), определив по ненулевому коду возврата этой функции, что он именно родитель, выдает сообщение об этом и закрывает свой хэндл *hpipe*[1] для конца записи. Сам передающий конец канала при этом не закрывается, так как у него остается еще продублированный в дочернем процессе хэндл этого конца канала (Системный вызов fork() дублирует все, что возможно, для дочернего экземпляра процесса, в частности таблицы, которые задают соответствия хэндлов дескрипторам

файлов.) Заметим, что для собственно закрытия передающего конца канала теперь необходимо закрытие хэндла *hpipe*[1] дочерним процессом, что последний и выполнит в свое время.

```
#include <stdio.h>
int main()
{int hpipe[2];
 char buffer[18];
 char txt[ ]="Text, for sending and demonstration pipes into Unix";
 int off=0, actread, lentxt=sizeof(txt), len, pid;

  printf("Begin working\n");
  if (pipe(hpipe)= =-1) {perror("pipe");exit(1);}
  if ((pid=fork())!=0)
   {printf("I am Parent, and my Child ID=%d\n",pid);
    close(hpipe[1]);
    while(1)
    {sleep(2);
     actread=read(hpipe[0],buffer,11);
     if (actread= =0) break;
     buffer[actread]='\0';
     printf("From pipe: %s len=%d\n",buffer,actread);
    }
    close(hpipe[0]);
    printf("End Parent\n");
   }
   else
   {printf("I am Child\n");
    close(hpipe[0]);
    while(1)
    {len=(4<lentxt-off) ? 4:lentxt-off;
     write(hpipe[1],&txt[off],len);
     off+=len;
     if (len!=4) break;
     sleep(1);
    }
    close(hpipe[1]);
    printf("End Child\n");
   }
```

```
  return 0;
}
```

Листинг 11.1.1. Использование неименованных каналов в Unix

Затем, используя дополнительную задержку на пару секунд для демонстрации продолжительности процесса, родитель запрашивает 11 байтов данных у канала. Возвращаемое значение *actread* функции read проверяется на предмет определения закрытия канала со стороны передающей стороны. При определения такого закрытия "бесконечный" цикл чтения прерывается. Любое число байтов, полученное в цикле от функции чтения из канала, выводится для наблюдения на экран консоли. По прерыванию цикла чтения (со стороны родителя) читающий конец канала закрывается, выдается сообщение о завершении родителя, и процесс завершается командой return 0.

Дочерний процесс, обнаружив по нулевому значению функции fork(), что он именно дочерний, в составном операторе после *else* сообщает о себе и выполняет закрытие *hpipe*[0]. (Читающий конец канала остается управляемым тем же исходном дескриптором этого конца канала, доступ к которому имеется в родительском процессе через хэндл *hpipe*[0].) В "бесконечном" цикле дочерний процесс подготавливает значение переменной *len*, задающей число подлежащих передачи следующих байтов данных через канал. (Это число выбирается равным 4, за исключением последней порции передачи данных, когда она может оказаться меньшей величины – в связи с некратностью длины передаваемого текста числу 4.) Очередная порция передаваемого текста, задаваемая смещением в переменной *off* и числом в переменной *len* передается посредством хэндла *hpipe*[1] через канал. После передачи корректируется величина смещения *off* и выполняется проверка, не передана ли последняя порция данных. Последняя ситуация определяется по тому, что значение вспомогательной переменной *len* оказывается меньшим начального значения 4 (последняя переданная порция меньше или равна нулю). По прерыванию оператором *break* цикла выполняется закрытие передающей стороны канала через закрытие хэндла *hpipe*[1]. Выдается сообщение о завершении дочернего процесса и он завершается оператором return 0.

11.2. Переназначение хэндлов для доступа к каналу

Когда дочерний процесс меняет программу, по которой он будет дальше работать, встает вопрос, как эта новая программа будет получать информацию о дескрипторах файлов родительского процесса. Возможным решением является передача параметров создаваемому процессу. Общие решения, принятые при организации процессов в операционной системе типа Unix, требуют передачи параметров в вызываемую программу исключительно в текстовой форме. Можно передать

текстовые параметры (теоретически любой текст) через область окружения (environment) или текстовую строку вызова программы. Поэтому передача хэндлов открытых файлов и концов канала передачи данных эти способом оказывается не удобной для программиста. Частичное решение этой проблемы получается на пути использования стандартных ввода и вывода, хэндлы которых в большинстве ОС имеют стандартные числовые значения, и поэтому не требуют фактической передачи.

Классической технологией программиста при использовании каналов передачи данных поэтому является использование стандартного ввода и вывода совместно с каналом. При этом один из концов канала, а часто и оба переобозначаются хэндлами, служащими для доступа к стандартному вводу и выводу. Для подобного обозначения предназначены специальные системные функции, в частности функция dup2 в Unix. Эта функция имеет прототип

int dup2(hsource, htarget).

Она создает новый хэндл для дескриптора, заданного хэндлом *hsource*, и размещает эту копию на месте с номером *htarget*. Возвращаемое значение, равное -1, указывает на неудачу выполнения функции. Существенно, что сами хэндлы при этом не меняются, изменяются специализированные структуры описателей открытых файлов в рабочей области ОС. Дело в том, что между числовым значением хэндла и дескриптором имеется промежуточное звено – специальная таблица соответствия, расположенная в области памяти, не доступной для прикладного процесса. Конкретный вид таблицы зависит от операционной системы. Числовому значению хэндла соответствует номер строки этой таблицы (в большинстве ОС хэндл является индексом строки в такой таблице). При дублировании хэндлов с помощью функции dup2 та из имеющихся строк этой таблицы, номер которой соответствует хэндлу *hsource*, дублируется в строку, задаваемую *htarge*.

Кроме рассмотренной функции dup2, для копирования хэндлов файлов имеется еще одна разновидность функции, называемая в Unix dup и имеющая прототип

int dup(hsource).

Последняя функция предоставляет доступ к дескриптору через новое значение хэндла, возвращаемое ею. При выполнении этой функции строится новая строка таблицы соответствия и в такую строку копируется информация соответствия из строки, заданной хэндлом *hsource*. Следует иметь в виду, что при доступе к файлу (или концу канала) через несколько хэндлов, связанных с этим файлом, закрытие файла происходит только после закрытия всех хэндлов, связанных с ним. Такой результат обеспечивается тем, что внутри дескриптора заведено специальное поле, которое учитывает текущее число хэндлов, указывающих на этот дескриптор. Таким образом, в общем случае следует различать закрытие файла или другого системного объекта (которое фактически является закрытием дескриптора) и закрытие хэндла. Каждый хэндл связан с дескриптором объекта, к

которому он обеспечивает доступ. Для файла именно дескриптор определяет текущую позицию в файле при доступе к нему.

Если для одного файла открыты два дескриптора, то последовательный доступ к такому файлу может отличаться для одного и другого из этих дескрипторов. Установка позиции в файле для последующего прямого доступа воздействует не на сам файл, а на его дескриптор.

Если же для одного файла открыт только один дескриптор, но доступ к этому дескриптору осуществляется через несколько хэндлов, то доступ через любой из этих хэндлов осуществляется к текущей позиции, задаваемой единственным дескриптором, и установка позиции в файле для последующего прямого доступа одинакова для доступа через любой хэндл к этому дескриптору.

Следующая пара программ для родительского в листинге 11.2.1 и дочернего в листинге 11.2.2 процессов демонстрирует описываемую технику переназначения хэндлов при организации работы с каналами.

```
#include <stdio.h>
#define PIPESIZE 16

int main()
{int hpipe[2], hstdout=1, hsave;
 int rc, cbRead;
 char bufread[PIPESIZE];

  if(pipe(hpipe)= = -1)   {printf("Error create Pipe\n");getchar();exit(1);};
   hsave=dup(hstdout);    dup2(hpipe[1], hstdout);
   rc=fork();
   if (!rc)
    {close(hpipe[0]); execl("childpip.exe",0);}
     close(hpipe[1]);
    dup2(hsave, hstdout);
    do
     {cbRead=read(hpipe[0], bufread, 4);
      sleep(1);
      bufread[cbRead]='\0'; printf("%s\n",bufread);
      }
   while (cbRead!=0);
   printf("\nEnd work with pipe\n");    return 0;
}
```

Листинг 11.2.1. Программа родительского процесса с переназначаемыми хэндлами

```
#include <stdio.h>
int main()
{int k,  cb;
 char text[ ]=”Мой дядя самых честных правил, \
 когда не в шутку занемог, он уважать себя заставил \
 и лучше выдумать не мог...”;
 char *ptxt;
 int hstdout=1;
   sleep(2);    ptxt=text;
   for (k=0; k<12; k++)
       {fprintf(stderr,”————I am Child ... (k=%d)\n\r”, k);
        cb=write(hstdout, ptxt, 9);
        ptxt +=9;        sleep(1);
       }
   close(hstdout);   return 0;
}
```

Листинг 11.2.2. Программа childpip.exe дочернего процесса

В программе родительского процесса после создания канала с массивом хэндлов *hpipe* выполняется дублирование хэндла стандартного вывода в хэндл с номером, выбранным программистом. (Практически копируется строка таблицы соответствия числовых значений хэндлов и дескрипторов файлов, именно строка с номером хэндла стандартного вывода.) Этот хэндл, задаваемый переменной *hsave*, имеет фиксированное значение. Далее копируется хэндл конца вывода канала в хэндл *hstdout* (более точно – строка таблицы для хэндла конца вывода в строку, соответствующую номеру *hstdout*). Последнее значение является стандартным для всех процессов Unix и численно равно 1, что позволяет в дальнейшем использовать значение этого хэндла в дочернем процессе без передачи собственно его значения. Теперь в передающий конец канала можно записывать данные, используя при этом хэндл, равный 1 (фиксированное значение переменной *hstdout* в нашей программе).

После этого процесс разветвляется на собственно родительский и дочерний с помощью вызова fork(). Дочерний процесс, идентифицируя себя по возвращаемому fork() нулевому значению, закрывает свою копию *hpipe*[0] хэндла принимающего конца канала и меняет программу на childpip.exe с помощью системного вызова execl(“childpip.exe”,0). Дальнейшие действия выполняет программа, приведенная в листинге 11.2.2. Описанный в ней текст пересылается порциями в 9 байтов через канал, используя вызов функции read в виде write(hstdout, ptxt, 9). Собственно запись в канал производится с помощью хэндла *hstdout*, который хотя и содержит стандартное значение 1, с учетом предыдущих действий по копирова-

нию хэндлов, задает передающий конец канала. Доступ к последующим порциям организуется через вспомогательный указатель *ptxt*, вначале установленный на начало текста, а затем наращиваемый на переданное число байтов. Для наглядности процесса передачи по каналу задается задержка на 1 с. после передачи каждой порции байтов. По завершению цикла передачи выполняется закрытие дескриптора с хэндлом *hstdout*, закрывая тем самым передающую сторону канала и вызывая далее на его принимающем конце ситуацию конца файла.

Родительский процесс в программе листинга 11.2.1, определив себя как именно родительский по неравному нулю значению переменной rc, которое возвращается вызовом fork(), переходит в цикл многократного чтения из принимающего конца канала с помощью оператора cbRead=read(hpipe[0], bufread, 4). В этом операторе используется хэндл принимающего конца канала и производится чтение с запросом четырех байтов. Прочитанная порция данных выдается на экран консоли функцией printf. Завершение цикла определяется по получении нулевого значения *cbRead* функции read при чтении из принимающего конца канала.

Заметим, что при необходимости дальнейшего использования стандартного вывода в родительском процессе целесообразно выполнить оператор dup2(hsave, hstdout), который восстановит ситуацию, когда хэндл *hstdout* автоматически задает дескриптор стандартного вывода. Альтернативой такому решению является использование хэндла hsave для доступа к стандартному выводу, что обеспечивает все требуемые возможности, но менее наглядно.

В операционных системах Windows также имеется системная функция дублирования хэндлов, причем здесь она имеет потенциально большее значение. Это связано с тем, что множество типов системных объектов в этих ОС доступны через их хэндлы и последние не специализированы. Указанные принципиальные решения позволяют поэтому дублировать хэндлы не только файлов, как в других ОС, но и практически любые хэндлы системных объектов. Такие возможности, по существу, вынужденно, заложены разработчиками Windows для получения принципиальных возможностей добраться из одного процесса до системных объектов других родственных процессов.

Упомянутая системная функция имеет прототип

```
BOOL DuplicateHandle(HANDLE hSourceProcessHandle,
   HANDLE hSourceHandle,      // handle to duplicate
   HANDLE hTargetProcessHandle,     // handle to process to duplicate to
   HANDLE *TargetHandle,      // pointer to duplicate handle
   DWORD DesiredAccess,       // access for duplicate handle
   BOOL InheritHandle,    // handle inheritance flag
   DWORD Options);       // optional actions
```

Параметр *Options* в простейших случаях задается равным нулю. Эта функция берет строку в таблице дескрипторов объектов для исходного процесса, зада-

ваемого хэндлом *hSourceProcessHandle*, а именно строку, определяемую в таблице хэндлом *hSourceHandle*. Она создает копию этой строки в таблице дескрипторов объектов для процесса с хэндлом *hTargetProcessHandle*, причем необходимо, чтобы процесс, выполняющий вызов функции DuplicateHandle, имел в качестве действующих хэндлы обоих упомянутых процессов. (Так, получается для процесса, создавшего дочерний процесс, который далее рассматривается как процесс с хэндлом *hTargetProcessHandle*.) Аргумент *TargetHandle* используется для возвращения значения хэндла, под номером которого размещается во второй таблице копия исходного дескриптора.

Теоретически рассмотренная функция может иметь дело с тремя различными процессами: тем, который выполняет вызов этой функции, тем, из которого берется значение дескриптора его собственного объекта, и, наконец, тем, в таблице дескрипторов которого строится копия дескриптора из второго процесса. Это дает удивительные по мощности средства, неумелое использование которых способно нанести серьезный вред функционированию приложений.

Для наших текущих целей построения и использования программного канала, соединяющего родительский и дочерний процесс, только что рассмотренная функция дублирования хэндлов оказывается бесполезной. Эта бесполезность связана с тем, что в Windows процессы изолированы друг от друга значительно сильнее, чем в других ОС и, в частности, отсутствуют средства получения действующего хэндла родительского процесса. Родительский процесс может построить копию своего дескриптора, который возникает при создании программного канала, путем вызова функции DuplicateHandle и разместить эту копию в таблице дескрипторов дочернего процесса, но последний не имеет непосредственных средств получить значение этого хэндла, а хэндлы стандартных устройств не имеют здесь постоянных значений. (Теоретически для передачи этого значения хэндла для копии дескриптора можно использовать другие средства межпроцессорных взаимодействий, но это существенно более сложный путь, чем рассмотренное далее более простое решение.)

11.3. Неименованные каналы в Windows

Для создания неименованных каналов в Windows NT (и Windows 9x) предназначена системная функция CreatePipe, имеющая прототип

```
BOOL CreatePipe(HANDLE* phReadPipe,   HANDLE* phWritePipe,
   LPSECURITY_ATTRIBUTES pAttributes,   DWORD nSize ),
```

где *phReadPipe* – адрес переменной для хэндла чтения из канала, *phWritePipe* – адрес переменной для хэндла записи в канал, *pAttributes* – адрес атрибутов защиты, *nSize* – число байтов, резервируемых для канала.

Для манипуляций с хэндлами стандартного ввода и вывода в Windows можно использовать не только универсальную функцию DuplicateHandle копирования

хэндлов (которая здесь предназначена для копирования хэндлов любых объектов ядра, а не только файлов), но и более простую специализированную функцию SetStdHandle. Она позволяет построить строку таблицы соответствия для стандартного ввода или вывода в строке таблицы соответствия, явно указываемой номером хэндла. Эта функция имеет прототип

BOOL SetStdHandle(DWORD nStdHandle, HANDLE hHandle),

где *nStdHandle* – условное обозначение хэндла стандартного ввода, вывода или ошибок, задаваемое одной из символических констант STD_INPUT_HANDLE, STD_OUTPUT_HANDLE, STD_ERROR_HANDLE. Аргумент *hHandle* задает хэндл для той строки таблицы соответствия, на место которой копируется строка таблицы соответствия для хэндла стандартного ввода или вывода.

В листингах 11.3.1 и 11.3.2 приведены программы родительского и дочернего процессов, которые демонстрируют использование неименованных каналов в Windows.

```
#include <stdio.h>
#include <windows.h>
#define PIPESIZE 16
int main()
{HANDLE hW, hR, hstdout;
 DWORD rc, cbRead;
 STARTUPINFO si;
 PROCESS_INFORMATION pi;
 DWORD CreationFlags;
 char bufread[PIPESIZE]="-";

  hstdout = GetStdHandle(STD_OUTPUT_HANDLE);
  if(!CreatePipe(&hR, &hW, NULL, PIPESIZE))
     {printf("Error create Pipe\n");getchar();exit(1);};
  SetStdHandle(STD_OUTPUT_HANDLE, hW);

  memset(&si, 0, sizeof(STARTUPINFO));
  si.cb=sizeof(si);
  CreationFlags = NORMAL_PRIORITY_CLASS | CREATE_NEW_CONSOLE;
  rc=CreateProcess(NULL, "childpip.exe", NULL, NULL, TRUE, //наследуемые
                CreationFlags, NULL, NULL, &si, &pi);
  if (!rc)
   {printf("Error create Process, codeError = %ld\n",
    GetLastError());    getchar();     return 0;
   }
```

```
   CloseHandle(hW);
   do
   {rc=ReadFile(hR, bufread, 4, &cbRead, NULL);
    if (!rc)
      {rc=GetLastError();
       if (rc= =ERROR_BROKEN_PIPE) break;
       else {printf("Error ReadFile, rc=%d\n", rc); }
      }
    Sleep(1000);
    WriteFile(hstdout, bufread,cbRead,&cbRead,NULL);
   }    while (cbRead!=0);
   printf("\nEnd work with pipe\n");
   CloseHandle(hstdout);
   CloseHandle(hR);
   return 0;
}
```

Листинг 11.3.1. Программа родительского процесса

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

int main()
{int step=0;
 DWORD cb;
 int portion=9;
 char text[ ]="У лукоморья дуб зеленый, златая цепь на дубе том.";
 char *ptxt=text, *textend=text+sizeof(text)-2;
 HANDLE hstdout;
  hstdout = GetStdHandle(STD_OUTPUT_HANDLE);
  ptxt=text;
  while(ptxt<=textend)
   {if (ptxt>textend-portion) portion=textend-ptxt+1; //last portion may be less than
9
    fprintf(stderr,"----I am Child ... (step=%d)\n\r", ++step);
    WriteFile(hstdout, ptxt, portion ,&cb, NULL);
    ptxt+=portion;
    Sleep(2000);
   }
  CloseHandle(hstdout);
  return 0;
```

```
}
```

Листинг 11.3.2. Программа childpip.exe дочернего процесса

Получение данных через канал в Windows имеет свои особенности. Попытка чтения данных из канала, передающий конец в котором закрыт, приводит не к нулевому числу прочитанных байтов, а к возвращаемому функцией ReadFile значению FALSE. Тем самым как бы фиксируется ошибка. При этом функция GetLastError() возвращает значение символической константы ERROR_BROKEN_PIPE (равное 109). Нормально построенная программа чтения данных из канала должна распознавать это значение как обычное завершение данных в канале аналогично концу файла. Именно так построена программа последнего примера.

Заметим, что в программе родительского процесса при создании дочернего процесса использован параметр наследования (пятый аргумент функции CreateProcess), равный TRUE. Это значение заставляет дочерний процесс унаследовать все дескрипторы открытых объектов и соответственно хэндлы этих дескрипторов.

11.4. Именованные каналы в Windows NT

Именованные каналы в Windows NT практически до деталей повторяют управление каналами из OS/2. Отличия касаются только вопросов синхронизации обмена данными и формы записи соответствующих функций и символических констант. Заметим, что именованные каналы полностью отсутствуют в операционных системах подсемейства Windows 9x.

Для именованных каналов принята своеобразная концепция общего именования целой группы действительных каналов. За ней стоит желание ограничиться возможно меньшим числом имен для каналов, чтобы эту информацию по человечески было проще использовать. В результате принятой концепции при первом создании именованного канала требуется указывать максимальное число реализаций для одноименной группы. Для расширения потенциальных возможностей можно задать неограниченное число каналов в группе.

Отдельный именованный канал передачи данных представляет собой два кольцевых буфера, каждый из которых имеет два указателя, обеспечивающих использование этих буферов. По сложившейся терминологии отдельный именованный канал передачи данных называют экземпляром канала или реализацией канала. Указатели для буфера задают первое следующее свободное место и место, с которого начинается информация, записанная ранее в канал. Кроме того, для каждой пары таких буферов система поддерживает специализированную структуру системных данных, описывающую характеристики такого канала.

В случае именованных каналов оба буфера, как системные структуры данных, имеют общий (или скорее одинаковый) хэндл, с помощью которого выполняются

операции чтения и записи над этими буферами. (Напомним, что в неименованных каналах каждый из концевых буферов, образующих канал, имеет собственный хэндл, и эти хэндлы для записывающего и принимающего концов буфера различаются.)

Для создания именованного канала служит функция с прототипом

```
HANDLE CreateNamedPipe(LPCTSTR  lpName,
   DWORD  dwOpenMode,  DWORD  dwPipeMode,
   DWORD  nMaxInstances,  DWORD  nOutBufferSize,
   DWORD  nInBufferSize,   DWORD  nDefaultTimeOut,
   SECURITY_ATTRIBUTES  *pSecurityAttributes ),
```

где параметр *lpName* задает адрес имени канала, параметр *dwOpenMode* – режим открытия канала, *dwPipeMode* – режим работы канала, параметр *nMaxInstances* задает максимальное число экземпляров канала, параметры *nOutBufferSize* и *nInBufferSize* определяют размеры выходного и входного буферов канала, а *nDefaultTimeOut* задает время максимального ожидания, выраженное в миллисекундах. Параметр *pSecurityAttributes*, специфичный именно для Windows NT, задает адрес атрибутов защиты канала (в большинстве ситуаций достаточно положить этот параметр равным NULL).

Значения параметров *dwOpenMode* и *dwPipeMode* выражаются через логическое побитовое объединение символических констант. Для задания *dwOpenMode* используются константы PIPE_ACCESS_DUPLEX, PIPE_ACCESS_INBOUND, PIPE_ACCESS_OUTBOUND и применяемые более редко FILE_FLAG_WRITE_THROUGH и FILE_FLAG_OVERLAPPED. Режим PIPE_ACCESS_INBOUND аналогичен режиму GENERIC_READ для обычных файлов, а режим PIPE_ACCESS_OUTBOUND аналогичен режиму GENERIC_WRITE обычных файлов. Константа PIPE_ACCESS_DUPLEX аналогична использованию GENERIC_READ | GENERIC_WRITE для обычных файлов.

Для задания параметра *dwPipeMode* предназначены следующие символические константы. Задание типа канала дается взаимоисключаемыми константами PIPE_TYPE_BYTE и PIPE_TYPE_MESSAGE, которые соответственно определяют работу канала в режиме отдельных байтов или сообщений. Режим чтения канала задается константой PIPE_READMODE_BYTE или PIPE_READMODE_MESSAGE. Использование константы PIPE_TYPE_BYTE несовместимо с константой PIPE_READMODE_MESSAG. Работа канала в режиме сообщений, задаваемая константой PIPE_TYPE_MESSAGE, позволяет применять любой из режимов чтения, которые задаются константами PIPE_READMODE_BYTE и PIPE_READMODE_MESSAGE. Практически тип канала определяется тем, в каком виде информация записывается в канал. Это может быть либо неструктурированная запись последовательностей байтов, либо запись в канал структурированных сообщений. Читать же сообщения можно как целиком в

приемный буфер либо отдельными байтами, причем в последнем случае до выхода из приемного буфера доходят только собственно данные, а информация заголовка отбрасывается.

Кроме типа канала и режима чтения, в параметре задается режим ожидания, для чего служат константы PIPE_WAIT и PIPE_NOWAIT. Первая определяет ожидание при невозможности немедленного выполнения операции с каналом, величина ожидания определяется при этом параметром nDefaultTimeOut. Вторая настраивает канал в режим немедленного выполнения управляющих функций. В последнем случае при невозможности немедленно выполнить указанное действие системная функция выполняется с ошибкой, значение которой можно получить функцией GetLastError(). Последний вариант в Windows NT предоставляется только для обеспечения совместимости и не рекомендован к использованию.

Параметр *nMaxInstances* может задаваться символической константой PIPE_UNLIMITED_INSTANCES, обозначающей неограниченное число экземпляров. Существенно, что при вызове функции создания именованного канала, когда создается очередной экземпляр его, этот параметр должен иметь то же значение, что и для первого экземпляра канала.

При неудачном выполнении функция создания именованного канала возвращает нулевое значение, а при удаче – хэндл созданного экземпляра канала.

Имя канала в Windows NT задается в универсальной форме вида

\\.\pipe*имяканала*

где *имяканала* может включать любые символы за исключением обратной черты (backslash) в частности цифры и специальные символы. Число символов в таком имени может достигать 256, причем используемые латинские буквы не зависят от регистра (строчные или прописные).

Собственно создание экземпляра именованного канала недостаточно для его немедленного использования. С учетом асинхронного характера фактического подключения к каналу двух взаимодействующих процессов для согласованной их подключенности используют универсальную методику запрос-ответ. Для удобства наименований два процесса, участвующие во взаимодействии с помощью именованных каналов, называют *серверным* и *клиентским*. Процесс, который создал экземпляр канала, называют серверным, а другой – клиентским.

В качестве запроса на образование действующего канала серверный процесс выполняет функцию запроса ConnectNamedPipe, ответом на которую со стороны клиентского процесса должно быть выполнение последним функции открытия файла CreateFile, задающей имя этого именованного канала.

Функции ConnectNamedPipe имеет прототип

```
BOOL ConnectNamedPipe(HANDLE  hNamedPipe,
                      LPOVERLAPPED  lpOverlapped),
```

где первый параметр задает хэндл ранее созданного именованного канала, а второй параметр в простейших случаях полагается равным NULL. Предназначен он для организации синхронизации с помощью объектов событий, где ожидание результата непосредственно запрашивается универсальной функцией WaitForSingleObject или WaitForMultipleObjects.

При невозможности немедленного выполнения функция подключения ConnectNamedPipe возвращает значение FALSE, после чего в программе следует предусматривать запрос кода ошибки функцией GetLastError(). Последняя функция возвращает значение ERROR_NO_DATA, если предыдущий клиент закрыл свой конец канала, и значение ERROR_PIPE_CONNECTED, если предыдущий клиент не закрыл свой конец.

Собственно операции передачи и приема данных через канал задаются универсальными функциями ReadFile и WriteFile, в которых в качестве хэндла используется хэндл именованного канала.

Для порождения ситуации конца файла на принимающем конце программа, задающая действия другого конца, должна выполнить операцию закрытия для дескриптора передающего конца канала.

Для более глубокого понимания действий в именованных каналах следует учитывать, что такой канал (точнее экземпляр канала) может находиться в следующих состояниях: *подключенном* (connected), *закрытом* (closing), *отключенном* (disconnected) и *прослушивающим* (listening). Канал находится в подключенном состоянии, если был создан и подключен серверным процессом и, кроме того, открыт процессом клиента (только подключенные к каналу процессы могут читать из него или записывать данные в в него). Канал находится в закрытом состоянии, если был закрыт клиентским процессом, но все еще не отсоединен серверным процессом. Канал находится в отключенном состоянии, если был создан серверным процессом, но не соединен или явно отсоединен и все еще не присоединен; отсоединенный канал не может принимать запрос открытия функцией CreateFile. Наконец, канал находится в прослушивающем состоянии, если был создан и соединен серверным процессом, но все еще не открыт клиентским процессом; слушающий канал готов принимать запрос на открытие. Если канал находится не в прослушивающем состоянии, функция CreateFile возвращает значение FALSE, после которого запрос функцией GetLastError() возвращает код ошибки, свидетельствующий о занятости канала.

Следующая программа, приведенная в листинге 11.4.1, задает функционирование простейшего серверного процесса для именованного канала.

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

```
int main()
{HANDLE hpipe;
 BOOL conct;
 char buffer[512];
 DWORD actread, actwrite;

  hpipe=CreateNamedPipe("\\\\.\\pipe\\OurPipe",
             PIPE_ACCESS_DUPLEX,
             PIPE_TYPE_MESSAGE|PIPE_WAIT|PIPE_READMODE_MESSAGE,
             1,512,512,2000,NULL);
  if (hpipe= =INVALID_HANDLE_VALUE)
   {printf("ErrorCreatePipe, RC=%d\n",GetLastError());
    getchar(); return 0;}
  printf("Waiting for connect...\n");
  conct=ConnectNamedPipe(hpipe,NULL);
  if (!conct)
   {printf("ErrorConnectPipe, RC=%d\n",GetLastError());
    getchar(); CloseHandle(hpipe); return 0;}
  printf("Connected! Waiting for data...\n");
  while (1)
   {if (ReadFile(hpipe,buffer, 512, &actread,NULL))
       {if(!WriteFile(hpipe, buffer, strlen(buffer)+1,
          &actwrite,NULL))
          break;
        printf("Received data: [%s]\n",buffer);
        if (!strcmp(buffer,"exit"))
          break;
       }
    else {printf("Error ReadPipe: RC=%d\n",GetLastError());
        getchar(); break;}
   }
   CloseHandle(hpipe);
   return 0;
}
```

Листинг 11.4.1. Серверный процесс для именованного канала

Открытие именованного канала со стороны клиентского процесса выполняется универсальной функцией CreatePipe, причем в ее параметрах должны быть заданы режимы доступа как по чтению, так и по записи (с помощью символических констант GENERIC_READ и GENERIC_WRITE), а пятый параметр функции (тре-

тий с конца и обозначаемый в прототипе как *dwCreationDistribution*) должен задаваться символической константой OPEN_EXISTING. Имя канала для открытия доступа к нему этой функцией должно задаваться в виде

\\servername\pipe*имяканала*

где на *имяканала* накладываются несущественные ограничения, изложенные при описании функции создания канала, а *servername* задает сетевое имя компьютера, на котором выполняется программа, создавшая канал.

Закрытие дескриптора, описывающего конец канала, производится универсальной функцией CloseHandle.

В листинге 11.4.2 приведена программа для клиентского процесса, работающего с серверным процессом по предыдущей программе.

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

int main()
{HANDLE hpipe;
 char buffer[512];
 DWORD actread, actwrite;

  hpipe=CreateFile("\\\\.\\pipe\\OurPipe",
              GENERIC_READ|GENERIC_WRITE,
              0, NULL, OPEN_EXISTING, 0,NULL);
  if (hpipe= =INVALID_HANDLE_VALUE)
   {printf("ErrorOpenPipe, RC=%d\n",GetLastError());
   getchar(); return 0;}
  printf("Connect with Server!. Type 'exit' for terminate.\n");
  while (1)
   {printf("Input data, please...>\n");
   gets(buffer);
   if (!WriteFile(hpipe,buffer, strlen(buffer)+1,
       &actwrite,NULL)) break;
   if(ReadFile(hpipe, buffer, 512,&actread,NULL))
     printf("Received back data: [%s]\n",buffer);
   else {printf("Error ReadPipe: RC=%d\n",GetLastError());
        getchar(); break;}
   if (!strcmp(buffer,"exit"))  break;
   }
   CloseHandle(hpipe);
   return 0;
```

```
}
```

Листинг 11.4.2. Клиентский процесс для именованного канала

При ее выполнении предполагаются, что серверный и клиентский процессы выполняются на одном и том же компьютере, в связи с чем в функции открытия именованного канала используется вырожденный префикс имени серверного компьютера.

Программу, использующую открытие канала клиентом с последующим выполнением записи в канал и дальнейшим чтением из него, можно заметно упростить, если воспользоваться специализированной функцией TransactNamedPipe. Последняя функция объединяет в себе все следующие операции, перечисленные в предыдущем предложении: она открывает канал, выполняет запись данных в него и дожидается ответного сообщения из канала, который затем закрывает. Ее ограничением является необходимость использовать передачу данных только сообщениями, для чего канал должен быть соответственно настроен.

Функция имеет с прототип

```
BOOL TransactNamedPipe(HANDLE  hNamedPipe,
   VOID*  pvWriteBuf,     // address of write buffer
   DWORD  cbWriteBuf,  // size of the write buffer, in bytes
   VOID*  pvReadBuf,      // address of read buffer
   DWORD  cbReadBuf,   // size of read buffer, in bytes
   DWORD*  pcbRead,     // address of variable for bytes actually read
   OVERLAPPED*  lpo).
```

В ней параметр *hNamedPipe* задает хэндл именованного канала и должен быть получен в результате выполнения функции CreateFile с именем канала. Параметры *pvWriteBuf*, *pvReadBuf*, *cbWriteBuf*, *cbReadBuf* задают адреса буферов для записи и чтения данных и их длины соответственно, параметр *lpo* в простейших случаях задается значением NULL. Практически эта функция объединяет в едином вызове обычные функции WriteFile и ReadFile. Особенностью функции TransactNamedPipe является возможность ее применения только с каналом, работающим в режиме пересылки сообщений. Поэтому для практического применения после открытия доступа к каналу следует переустановить автоматически обеспечиваемый доступ побайтного чтения на режим чтения сообщений. Для этих целей требуется воспользоваться функцией SetNamedPipeHandleState. Она имеет прототип

```
BOOL SetNamedPipeHandleState(HANDLE hNamedPipe, DWORD* pdwMode,
   DWORD*  pcbMaxCollect,  DWORD*  pdwCollectDataTimeout),
```

где *pdwMode* задает адрес двойного слова с кодом новых режимов канала, а последующие два параметра имеют достаточно специальный характер и здесь рассматриваться не будут. Достаточно иметь в виду, что для их не использования на месте

этих параметров должны стоять значения NULL. Для установки режима чтения сообщений следует использовать символическую константу PIPE_READMODE_MESSAGE.

В листинге 11.4.3 приведена программа для клиентского процесса, которая использует функцию транзактивного обмена при работе с каналом.

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

int main()
{HANDLE hpipe;
 char bufr[512];
 char bufw[512];
 DWORD actread,mode;

 hpipe=CreateFile("\\\\.\\pipe\\OurPipe",
            GENERIC_READ|GENERIC_WRITE,
            0, NULL, OPEN_EXISTING, 0,NULL);
 if (hpipe= =INVALID_HANDLE_VALUE)
  {printf("ErrorOpenPipe, RC=%d\n",GetLastError());
   getchar(); return 0;}
 printf("Connect with Server!. Type 'exit' for terminate.\n");
 mode = PIPE_READMODE_MESSAGE;
 if (!SetNamedPipeHandleState(hpipe,&mode, NULL, NULL))
  {printf("ErrorSetNamedPipeState, RC=%d\n",
     GetLastError());
   getchar(); return 0;}

 while (1)
  {printf("Input data, please...>\n");
   gets(bufw);
   if (TransactNamedPipe(hpipe,bufw, strlen(bufw)+1, bufr, 512,
      &actread,NULL))
   printf("Received back data: [%s]\n",bufr);
   else {printf("Error ReadPipe: RC=%d\n",GetLastError());
       getchar(); break;}
   if (!strcmp(bufr,"exit"))  break;
  }
 CloseHandle(hpipe);
 return 0;
```

```
}
```

Листинг 11.4.3. Использование функции TransactNamedPipe

Еще раз отметим, что без установления режима чтения из канала на чтение сообщений вызов функции TransactNamedPipe не будет исполнен и вернется код ошибки.

Кроме рассмотренных функций для расширенной работы с именованным каналам, имеются функции CallNamedPipe, GetNamedPipeHandleState, GetNamedPipeInfo, PeekNamedPipe, WaitNamedPipe. Первая из них обобщает в одном вызове открытие канала, установку режима и функцию транзакции TransactNamedPipe. Функция PeekNamedPipe позволяет считывать данные из канала, не извлекая их из него, и является средством "подсматривания" за следующими данными. Функции GetNamedPipeHandleState и GetNamedPipeInfo позволяют получать детальную информацию о канале, причем первая из них ориентирована на получение информации об экземпляре канала, изменяемой в процессе эксплуатации последнего, а функция GetNamedPipeInfo дает неизменяемую информацию о канале, заданную при его создании.

11.5. Именованные каналы в Unix

Именованные каналы в Unix называются также каналами FIFO (First Input - First Output). Для их создания предназначена системная функция mkfifo, имеющая прототип

```
int mkfifo(const char *pathname, mode_t  mode),
```

где аргумент *pathname* задает символьное обозначение именованного канала, а аргумент *mode* – режимы доступа к этому каналу, аналогичные правам доступа для файла. Использование этого вызова требует указания заголовочных файлов sys/types.h и sys/stat.h .

Функция mkfifo возвращает значение хэндла для созданного именованного канала или значение -1 при невозможности его создать.

После создания именованного канала его необходимо открыть. В отличие от других ОС здесь открытие именованного канала требуется и для того процесса, который создал этот канал.

Особенностью открытия именованного канала для записи путем использования режима открытия O_WRONLY является блокировка процесса на вызове открытия до тех пор, пока какой-то другой процесс не откроет этот же канал для чтения. (Если на момент выполнения такой функции открытия по записи другой процесс уже открыл его для чтения, то никакой блокировки не возникает.)

Альтернативой указанного варианта открытия является применение неблокирующего открытия, задаваемого в виде

open(*имя_канала* , O_WRONLY | O_NONBLOCK).

Вызов последней функции вернет значение -1, если указанный канал не открыт еще для чтения.

Работают именованные каналы в Unix в режиме сообщений. Это обозначает, что если множество процессов посылает в канал некоторые последовательности символов, каждая из которых задается своим вызовом функции write, то принимающая сторона будет принимать каждую такую последовательность целиком и никакие две из них не смещают свои символы.

Каждый именованный канал практически представляет собой буфер обмена данными между процессами, открывшими этот канал. Поэтому при необходимости обеспечить дуплексную связь между процессами следует создавать и использовать не один, а два таких канала, открывая их с соответствующим режимом с обеих сторон (задавая с одной стороны доступ по чтению, а с другой – по записи). Попытка использовать один именованный канал с открытием его для какой-то стороны как по чтению, так и по записи приводит к тому, что процесс одной стороны может читать из такого канала именно то, что он же сам в него записал.

Следующий пример демонстрирует рассмотренные средства. Программа для серверной стороны этого примера приведена в листинге 11.5.1.

```
#include <stdio.h>
#include <sys/fcntl.h>
#include <errno.h>

int main()
{int hpipeto, hpipefrom;
 char buffer[512];
 int actread, actwrite, rc;

  printf("Begin\n"); sleep(2);
  rc=mkfifo("/tmp/PipeToClient",0646);
  if (rc= = -1 && errno!=EEXIST)
        {perror("ErrorCreatePipe:"); exit(1);}
  rc=mkfifo("/tmp/PipeFromClient",0646);
  if (rc= = -1 && errno!=EEXIST)
        {perror("ErrorCreatePipe:"); exit(1);}
  printf("Waiting for connect...\n");
  hpipeto=open("/tmp/PipeToClient",O_WRONLY);
  if (hpipeto= =-1)    {perror("Error PipeToClient");  exit(1);}
  hpipefrom=open("/tmp/PipeFromClient",O_RDONLY);
  if (hpipefrom= =-1)     {perror("Error PipeFromClient");  exit(1);}
```

```
  printf("Connected! Waiting for data...\n");
  while (1)
   {actread=read(hpipefrom,buffer, 512);
    if (actread)
        {if(!write(hpipeto, buffer, strlen(buffer)+1))  break;
         printf("Received data: [%s]\n",buffer);
         if (!strcmp(buffer,"exit"))      break;
        }
    else {perror("Error ReadPipe:");  getchar(); break;}
   }
  close(hpipeto);
  close(hpipefrom);
  return 0;
}
```

Листинг 11.5.1. Программа для серверной стороны именованного канала

В этой программе системными вызовами создаются два именованных канала. Оба они создаются в каталоге Linux, предназначенном для временных файлов и имеющем наименование /tmp. В качестве собственных имен выбраны названия *PipeToClient* и *PipeFromClient*. Эти каналы создаются с правами как на чтение, так и на запись для всевозможных программ, не принадлежащих пользователю, создавшему указанные каналы.

После попытки создания каждого из этих файлов вызовом функции mkfifo, в случае когда вызов вернет значение -1, информирующее о невыполненной операции, глобальная системная переменная ошибки (переменная errno) проверяется на совпадение с константой EEXIST. Эта константа означает, что указанный для создания файл уже существует. В нашем случае существование кем-то ранее созданного именованного канала нас вполне устраивает, поэтому выполнение программы прекращается только, если при создании канала получена ошибка, отличная от константы EEXIST.

Затем канал PipeToClient открывается для записи, а канал PipeFromClient – для чтения. При удачных открытиях запускается цикл, в котором из канала PipeFromClient читается сообщение размером до 512 байтов. Если удается прочитать сообщение (а не признак конца передачи – конца файла, определяемый нулевой длиной полученного сообщения), то оно пересылается обратно клиенту через канал PipeToClient, причем при невозможности переслать через канал (возвращаемого числа пересланных байтов, равное нулю), цикл завершается оператором break. Кроме обратной пересылки, полученное сервером сообщение выводится на экран оператором printf. При обнаружении, что полученное клиентом сообщение

есть команда, задаваемая текстом exit, цикл разрывается, выполняется закрытие серверных сторон обоих используемых каналов, и программа завершается.

Программа для клиентской стороны этого примера приведена в листинге 11.5.2.

```
#include <stdio.h>
#include <string.h>
#include <sys/fcntl.h>

int main()
{int hpipein,hpipeout;
 char buffer[512],buffer2[512];
 int actread, actwrite,rc, action;

  hpipein=open("/tmp/PipeToClient",O_RDONLY);
  if (hpipein= =-1)    {perror("ErrorOpenPipe:"); exit(1);}
  hpipeout=open("/tmp/PipeFromClient",O_WRONLY);
  if (hpipeout= =-1)    {perror("ErrorOpenPipe:"); exit(1);}
  printf("Connect with Server!. Type 'exit' for terminate.\n");

  while (1)
   {printf("Input data, please...>\n");
    gets(buffer);
    if (!write(hpipeout,buffer, strlen(buffer)+1))    break;
    if((actread=read(hpipein, buffer2, 512)))
      printf("Received back data: [%s]\n",buffer2);
    else {perror("Error ReadPipe:"); break;}
    if (!strcmp(buffer,"exit"))  break;
   }
  close(hpipein);   close(hpipeout);
  return 0;
}
```

Листинг 11.5.2. Программа для клиентской стороны именованного канала

В этой программе открываются именованные каналы, имеющие имена *PipeToClient* и *PipeFromClient* в каталоге временных файлов /tmp. Причем открываются соответственно для чтения и записи. После удачного открытия и выдачи информативного сообщения пользователю о подсоединении к серверу запускается цикл посылки и получения сообщений. В цикле от пользователя запрашивается ввод данных и полученные от функции gets() данные посылаются через канал с

именем *PipeFromClient* (используя полученный при открытии хэндл *hpipeout*). При невозможности переслать данные (возвращаемое нулевое число пересланных байтов), цикл разрывается оператором break. Иначе в дополнительный буфер *buffer2* из канала с именем *PipeToClient* (с помощью хэндла *hpipein*) читаются данные от сервера. Они выдаются на экран с примечанием, что получены обратно. Текстовая команда *exit*, введенная в качестве запрашиваемых данных разрывает цикл, после которого оба канала со стороны клиента закрываются.

12. ВЗАИМОДЕЙСТВИЕ ПОЛЬЗОВАТЕЛЯ С ОС

12.1. Интерфейсы операционных систем

Одной из трех важнейших функций ОС является ее интерфейс с пользователем. Именно благодаря ему оказывается возможным запускать программы, управлять их выполнением и обеспечивать сопровождение файлов. В эти же средства взаимодействия с операционной системой входят множества других средств и функций.

К настоящему времени сложились две принципиально отличные системы интерфейса (в переводе – взаимодействия с пользователем). Первая система называется *командным интерфейсом* или *интерфейсом командной строки*. Она позволяет пользователю управлять запуском и выполнением программ, задавая из текстовой консоли управляющие тексты, т.е. команды. Такая система сложилась в начале 70-х годов XX века и наиболее полное развитие получила в рамках операционной системы Unix. Заметим, что управление с помощью специальных текстов очень близко по особенностям к обычному программированию на языках высокого уровня, и поэтому командным интерфейсом легко овладевают именно программисты, для непрофессиональных пользователей этот подход может оказаться столь же тяжелым, как и обучение программированию.

Второй из упомянутых систем интерфейса является графическая. Она сложилась в разработках PARC (Palo Alto Research Center фирмы Xerox) в конце 70-х годов XX века, но получила широкое применение вначале в операционных системах фирмы Apple (MacOS) середины 80-х годов, а затем в операционных оболочках, а позже в операционных системах фирмы Microsoft. Широким массам непрофессиональных пользователей она хорошо известна по графическим оболочкам MS Windows 3.1, Windows 9x, Windows NT и более поздним их модификациям.

Управление компьютером с помощью графического интерфейса так относится к интерфейсу командной строки, как просмотр комиксов (рассказов в картинках) относится к чтению художественных произведений. Несомненно, что оба первых варианта проще для невнимательного или малограмотного человека, а также для человека, не привыкшего утруждать себя. В то же время число пользователей

и комиксов, и графического интерфейса на порядок, если не на порядки, превышает пользователей строгих текстов.

Управление с помощью графического интерфейса психологически многократно проще и требует существенно меньше волевых усилий, внимания и запомненной информации. Практически в графическом интерфейсе необходимая информация почти всегда присутствует непосредственно на экране, где предлагается сделать один из возможных выборов. Но при этом сложная настройка требует множества диалоговых окон, и во многих случаях общая картина настройки оказывается мало обозримой. Практически этот подход способен решать только небольшое число типовых задач настройки, но не обладает ни глубиной, ни универсальностью. В идейном плане графический интерфейс очень близок к так полюбившемуся американцам языку программирования Кобол (Cobol), который предназначен для решения экономических задач и позволяет записывать алгоритмы не с помощью специализированных и достаточно абстрактных операторов, а с помощью почти обычных фраз английского языка. В частности, арифметические операции в Коболе записываются не математическими символами, а английскими словами add, subtract, multiply. Непрофессионала такой стиль очень утешает, но профессионала раздражает отсутствие компактности и четкости в информационных управляющих конструкциях.

Несмотря на широкое применение графического интерфейса в ОС типа Windows, внутреннее общение между компонентами самой операционной системы неизбежно имеет характер управляющих текстов, в частности, машинных команд и текстовых вызовов системных функций. Принципиальная сложность задания внутри программы аналога воздействия на графический интерфейс вынужденно оставляет интерфейс командной строки на некотором заднем плане, не видном непрофессиональным пользователям. Поэтому практически все внимание в данной главе будет сосредоточено на интерфейсе командной строки, а возможности графического интерфейса рассматривать не будем, отсылая для его изучения к множеству руководств по поверхностному управлению конкретными операционными системами.

12.2. Командные и операционные оболочки (shells)

Как правило, все современные операционные системы имеют внутренний набор текстовых команд, которые позволяют раздельно задавать отдельные действия в операционной системы. Эти команды относятся к средствам управления сеансом командной строки (термин MS Windows) или просто к командам командной строки (термин Unix).

Указанные команды позволяют выполнять действия над объектами операционной системы. С точки абстрактной теории, именно в этих командах сосредоточены наиболее общие абстракции объектно-ориентированного программирования,

причем применительно к самой операционной системе. Заметим, что обычные языки программирования имеют дело преимущественно с информационными объектами гораздо более примитивными по своему строению.

Основными объектами манипулирования посредством командной строки являются файлы, процессы, потоки информации и взаимодействия. Причем файлы рассматриваются как неделимые объекты, а не как совокупности байтов, требующих обработки.

Основные действия, задаваемые в командной строке, следующие: запуск процесса по задаваемой программе, манипуляции с файлами (копирование, переименование, создание нового файла, создание каталога и переход в него как в текущий), изменение текущих свойств перечисленных выше объектов операционной системы. Последние действия включают изменение атрибутов и прав доступа файлов, изменение характера выполнения вычислительного процесса (изменение приоритета, перевод его в фоновый режим или передний план), изменение характеристик работы консоли и множества других. (Опять же, как видим, для этих действий нет аналогов в обычном программировании.)

С учетом важности выполнения перечисленных команд для полноценного функционирования операционной системы под управлением опытного администратора, созданы вспомогательные программные средства – командные оболочки, называемые (по традиции Unix) shell. Эти командные оболочки позволяют, кроме приема и запуска собственно команд ОС, использовать специальные командные тексты – *сценарии команд*, называемые также *сценариями* shell или *командными файлами*. Практически эти сценарии есть текстовые файлы, содержащие последовательности команд, и, может быть, – вспомогательные средства организации таких команд для их повторений или условного выполнения. По существу, сценарии команд представляют собой специализированные программы, написанные на языке сценариев для конкретной командной оболочки.

Командные оболочки, с точки зрения программиста, представляют собой интерпретаторы команд. Иначе говоря, они выполняют команду непосредственно после ее ввода с клавиатуры или по очереди записи в командном файле.

В старой операционной системе MS DOS, вызываемой обычно из-под ОС типа Windows 9x, выполнение отдельных команд и командных файлов обеспечивается командным интерпретатором COMMAND.COM. Это же наименование имеет командный интерпретатор и в ОС типа Windows 9x. В операционных системах OS/2 и Windows NT командный интерпретатор называется CMD.EXE.

В операционных системах типа Unix – наиболее широкое семейство командных интерпретаторов (shells). Это традиционный и наиболее широко используемый интерпретатор bash (Bourn Again Shell), модификации командного интерпретатора C-Shell, приближенного к возможностям языка C, и иногда ряд других, но частично похожих на упомянутые. В применении к Linux будем рассматривать

два его основных командных интерпретатора – bash и tcsh. Первый из них представляет в Linux расширенный типовой интерпретатор Bourn'a, а второй усовершенствованную разновидность C-Shell.

Заметим, что современные командные интерпретаторы не ограничиваются базовыми или типовыми интерпретаторами для конкретной ОС. В последнее время резко возрос интерес к универсальным командным интерпретаторам, позволяющим управлять операционной системой. К ним отчасти можно отнести интерпретаторы языка JavaScript и, главное, одну из последних очень эффективных новинок – интерпретаторы языка Tcl/Tk (Tool Command Language/Tool Kit). В современной терминологии все командные интерпретаторы и подобные им программные системы относятся к классу интерпретаторов языков сценариев или script-языков. Язык Tcl/Tk называют языком системной интеграции. Он был создан вначале для ОС типа Unix и особенно для поддержки в режиме интерпретации графической оболочки X Window этих ОС. К настоящему времени Tcl/Tk реализован интерпретаторами для Windows 3.1/9x/NT, Macintosh, AIX, QNX, VMS и OS/2.

В заключение следует подчеркнуть, что базовые командные интерпретаторы ОС фирмы Microsoft очень бедны инструментальными средствами и по своим возможностям только едва-едва подтягиваются к "мощности" интерпретируемого языка Basic.

12.3. Основные команды базовых операционных систем

Для начинающего читателя необходимо привести хотя бы минимальный набор команд для всех рассматриваемых далее операционных систем, не отсылая его к другим источникам. В то же время предлагаемый далее обзор таких команд по необходимости будет не очень полон и призван в большей степени дать информацию для последующих примеров, чем для охвата всего основного набора команд, необходимого для грамотного управления операционной системой.

Заметим, что, к счастью для начинающих, базовые наборы команд операционных систем MS DOS, Windows 9x/NT и OS/2 практически совпадают (для удобства кратких ссылок будем называть эти наборы вариантом MS). Такое совпадение в значительной степени облегчит наше знакомство. Более того, так как идейным источником построения команд более поздних ОС послужила Unix, то при всех отличиях между ними сохранено немало общего. Команда для операционной системы всегда начинается с ее собственного имени. Это имя задается обязательно латинскими буквами, причем в ОС Unix эти буквы должны быть строчными, а в остальных регистр (размер) букв может быть любым (как в Паскале). После имени команды может следовать (хотя и не обязательно) перечень опций. Опции в Unix начинаются со служебного символа *дефис* (символ -), в других ОС символом начала опции должна быть наклонная черта (slash - /). Сами опции, как правило, за-

даются одной латинской буквой. Применение односимвольных опций и дефиса в Unix позволяет объединять в одной записи несколько опций. Так вместо записи

command -g -j -x

можно задавать те же указания в виде

command -gjx

Часто в составе текста команды присутствует задание файлов. Для задания их имен во всех упомянутых системах могут использоваться метасимволы ? и *. Символ ? при этом обозначает любой допустимый символ в имени, а символ * - любую допустимую, в том числе пустую цепочку символов.

Учитывая, что командный интерфейс служит для взаимодействия профессионального пользователя с ОС, важнейшими являются команды получения оперативной информации от операционной системы. Главной из них служит команда получения самой общей информации о файлах, в частности, содержимого текущего каталога. Эта команда имеет для MS наименование dir, а аргументом ее может быть обозначение некоторого каталога, в частности логического диска. Кроме того, обычно с помощью метасимволов может быть запрошена информация о каких-то файлах. Например:

dir D:\RAB*.c

запрашивает информацию о всех файлах с расширением .c в каталоге RAB логического диска D: . При отсутствии (в качестве параметра команды) имени файла (или обобщенного имени файлов) выводится информация о всех файлах текущего каталога.

В Unix для отображения информации о файлах предназначена команда ls. Она при отсутствии явного указания файла или файлов также выводит информацию о всех файлах текущего каталога. Причем запрос на вывод подробной информации о файлах требует указания опции -l, а запрос о служебных файлах каталога требуется указания опции -a.

Следующей по значимости служит команда копирования. Для MS варианта она имеет в простейшей форме вид

copy *исходный_файл выходной_файл*

Кроме того, используется форма

copy *имена_файлов имя_каталога*

или

copy *имя_файа имя_каталога\новое_имя-файла*

В Unix команда копирования задается именем cp, причем при записи каталогов следует использовать обычную наклонную черту (символ /).

Кроме копирования видную роль играет команда перемещения файла с возможным изменением его имени. Эта команда в Unix задается в виде

mv *текущее_имя-файла новое_имя-файла*

или

mv *имя-файла имя-каталога*

а в MS командой, все отличие которой заключается в использовании имени move вместо сокращенного обозначения mv, характерного для Unix.

Несколько команд связаны с действиями над каталогами. Прежде всего – это часто используемая команда перехода в указанный каталог, имеющая общее наименование cd. Ее аргументом служит обозначение каталога, в который следует перейти. Следует учитывать, что родительский каталог во всех рассматриваемых ОС обозначается двумя последовательными символами точки (команда "перейти в родительский каталог" записывается как cd ..). В ряде команд может потребоваться обозначение текущего каталога, который задается одним символом точки. (Так что копирование файла aaa.txt дочернего каталога sss в текущий запишется в виде copy sss\aaa.txt .)

Создание каталога задается командой общего наименования mkdir, единственным аргументом которой служит имя создаваемого каталога. В ОС варианта MS для этой команды можно использовать сокращенное наименование md.

Служебную роль играет команда выдачи на экран содержимого файла. В Unix в качестве таковой используется универсальная команда с именем cat, позволяющая отображать содержимое перечня файлов, заданных в ней аргументами. В ОС типа MS для этих целей предназначена более ограниченная команда type (заимствованная из более простой очень теперь старой ОС CP/M).

Для удаления файлов в ОС типа MS предназначена команда del, а для удаления каталогов здесь же – команда rmdir. В Unix для обеих целей служит одинаково называемая команда rm. Следует обратить внимание, что удаление как файлов, так и особенно каталогов, это действие, которое может быть необратимым, поэтому прибегать к нему нужно по правилу: "семь раз отмерь, один раз отрежь".

Для запроса справочной информации в Unix предназначена команда с именем man, которая должна содержать имя команды, о которой запрашиваются пояснения. Для аналогичного запроса в ОС типа MS служит команда help.

Для более детального изучения основных команд следует обратится к указанной справочной системе и внимательно прочитать выдаваемую ею информацию. В табл. 12.1 приведена краткая информация по рассмотренным командам, которая, конечно, ни в коей мере не может заменить справочную информацию.

Таблица 12.1. Основные команды базовых ОС

Обозначение в Unix	Обозначение в MS	Описание действий команды
ls	dir	Выдает список файлов и каталогов.
copy	cp	Копирование файлов.
cd	cd	Переход в другой каталог (сделать его текущим).
mkdir	mkdir	Создание нового каталога (в MS допускается со-

		кращение md).
cat	type	Вывод содержимого файла на экран. В Unix команда позволяет отображать заданный перечислением список файлов и может содержать более одного аргумента.
move	mv	Перемещение файла или его переименование в текущем каталоге (в MS имеется также специальная команда переименования ren).
rm	del	Удаление файла или каталога (в MS каталог удаляется rmdir).
man	help	Запрос справочной информации о команде.
ls -l	attrib	Получение атрибутов файла (MS) или прав доступа (Unix).
clear	cls	Очистка экрана.
date	date	Отображение на экране даты и установка новой (в Unix только администратором).
time	time	Отображение на экране времени и установка нового (в Unix только администратором).
more	more	Постраничный вывод.

Одной из задач этой таблицы является показ, что между системами команд рассматриваемых ОС есть много общего, определяемого самими функциями операционных систем. Другие средства, более связанные со спецификой ОС, будут кратко рассмотрены в следующих параграфах.

12.4. Групповое выполнение и фоновый запуск команд

Практическая деятельность администратора операционной системы требует запуска более чем одной команды одновременно. Такая потребность связана как с многопрограммными возможностями современных ОС, так и периодической потребностью задавать одновременно выполнение более чем одной команды.

С учетом краткости большинства команд ОС для практика-пользователя оказывается соблазнительным задавать в одной строке более чем одной команды. Дело в том, что в отличие от обычного программирования, где программа для компилятора может иметь большой размер, но просматривать ее удобно текстовым редактором, команды операционной системы в простейшем управлении выполняются немедленно после ввода, поэтому в поле экрана желательно иметь как можно больше команд.

Командные интерпретаторы Unix позволяют комбинировать в одной строке более одной команды, разделяя их символом "точка с запятой" (символ ';'). Если

же какая-то команда для Unix не помещается в стандартной длине строки экрана, ее можно продолжить на следующей строке, завершив предыдущую символом \, непосредственно за которым должна быть нажата клавиша "перевод строки" (символ '\n' в языке Си).

В ОС Windows NT и OS/2 для группирования в одной строке более чем одной команды служит символ &. Он используется совершенно так же как разделитель ';' в Unix. Например, задание копирования файла fff из текущего каталога в подкаталог sss и немедленное после этого переименование исходного файла в файл qqq запишется для этих ОС в виде одной группы команд (пакета команд):

```
copy fff  sss\fff  & ren fff  qqq
```

а в Unix составной командой

```
cp fff  sss/fff  ; mv fff  qqq
```

При подобном группировании нескольких команд в один пакет могут возникать проблемы из-за невыполнения каких-то составляющих команд этого пакета. Поэтому интерфейс командной строки предоставляет средства приказов выполнения следующей команды только при условии выполнения предыдущей или, наоборот, приказывается выполнение следующей команды при невыполнении предыдущей. Для этих целей предназначены соответственно обозначения && и ||. Так, копирование исходного файла fff из текущего каталога в подкаталог sss и удаление исходного файла при удачном копировании запишется в виде

```
copy fff  sss/fff && del  fff
```

а выдача командой ECHO предупреждения после неудачной попытки копирования может быть задана составной командой

```
copy fff  sss/fff || echo NOT COPY
```

Другой принципиальной возможностью, представляемой современными ОС, является запуск программы как параллельного процесса относительно последующих указаний с консоли. Для этих целей в ОС Windows NT и OS/2 предназначена команда START. В качестве параметра этой команде задается имя программы или команды операционной системы. Действие команды start в Windows NT и OS/2 заключается в создании отдельной консоли (текстового окна), в котором запускается указываемая команда или программа.

В Unix аналогичная функция командного режима задается менее выразительно: для запуска команды или программы параллельно с основной работой консоли служит символ &, записываемый после указания в командной строке этой команды или программы. Происходящее при этом называют запуском в фоновом режиме. Это название отражает тот факт, что запущенная таким образом программа не использует явно консоль, а выполняется как бы "за кадром", причем такое выполнение производится с более низким приоритетом, чем работа консоли и непосредственно запускаемых с нее команд. Запуск в фоновом режиме сопровождается выдачей на консоль номера задания и PID идентификатора нового запущенного

процесса, созданного для фонового задания. Первое из этих значений выдается в следующей строке в квадратных скобках, а значение PID за ним как обычное число. В дальнейшем пользователь с консоли может частично управлять таким фоновым процессом, используя для его обозначения либо номер задания, либо идентификатор процесса.

Управление процессом, запущенным с консоли Unix, представляют средства уничтожения этого процесса, его приостановки и перевода в приоритетный режим. Уничтожение запущенного ранее процесса осуществляется с помощью команды kill. В качестве единственного аргумента этой команды используется номер задания или идентификатор процесса, причем значение идентификатора задается непосредственно числом, а номеру задания должен предшествовать специальный символ %.

Для приостановки запущенного с консоли задания служит не просто команда, а управляющая комбинация клавиш, задаваемая нажатием клавиши z при нажатой клавиши Ctrl (нажатие, обозначаемое обычно как [Ctrl+z]. После приостановки задания оно сопровождается на консоли информацией вида

[*номер*] + Stopped *имя команды с параметрами*

где *номер* дает идентификатор приостановленного процесса. После приостановки задание можно запустить далее в обычном приоритетном режиме, вводя команду fg с аргументом, задающим приостановленный процесс (номер идентификатора или номер задания с предшествующим ему символом %) или запустить в фоновом режиме командой bg.

Получение полной информации о запущенных в системе процессах возможно с помощью специальной команды. В Unix – это команда ps, а в Windows NT и OS/2 – команда pstat.

12.5. Стандартный ввод-вывод и конвейеры командной строки

Стандартный ввод-вывод предоставляет мощные средства для операций в командной строке. Эти средства основаны на переадресации стандартного ввода-вывода. Большая часть команд операционной системы в их внутренней программной реализации построена таким образом, что для ввода и вывода данных используется исключительно стандартный ввод и вывод. В результате переадресация этого ввода или вывода в командной строке позволяет вместо экрана и клавиатуры использовать желаемые пользователю файлы.

Так, при использовании команды отображения информации о файлах, можно ее задать в виде

dir >имя_файла

(или соответственно ls >имя_файла) и желаемая информация для будущего употребления запомнится в указанном файле. Ряд команд операционной системы

строятся как *фильтры*, т.е. используют и стандартный ввод, и стандартный вывод. К таким командам относятся команды sort, more (общие для всех рассматриваемых ОС) и команда wc из Unix, которая подсчитывает число строк, слов и символов в файле, служащем для ввода информации (в частности, текста, вводимого с клавиатуры).

Практическое значение стандартного ввода-вывода особенно велико в связи с применением так называемых конвейеров в командной строке. Конвейер представляет заданное в команде связывание стандартного вывода одной команды с стандартным вводом другой – следующей в строке – команды. В качестве символа связывания команд в конвейер используется символ | (одинарная вертикальная черта). Конвейер по внешнему виду представляет запись

команда1 | *команда2*

Здесь стандартный вывод команды *команда1* автоматически связывается со стандартным вводом команды *команда2*. Поэтому данные результата первой команды не появляются на экране, а полностью передаются на вход второй команды, которая и использует их в качестве обрабатываемой ее информации. В одной составной команде может быть построен конвейер не только из двух, но и из любого числа команд, соединенных при записи символом вертикальной черты. Единственное требование при этом заключается в том, что для соединяемых таким путем команд первая из них должна использовать стандартный вывод, а вторая – стандартный ввод.

Практически наиболее часто в конвейерах используется команда more, которая оказывается заключительной в конвейере, поскольку выводит на экран, а не в стандартный вывод. Ее значение в том, что вывод на экран производится порциями по размеру экрана, это позволяет удобно наблюдать результаты объемного вывода на экране, где без ее использования часть выводимых данных исчезает – автоматически выдвигается за верхнюю границу экрана. Даже в операционных системах типа MS эта вспомогательная команда достаточно широко используется при текстовом выводе информации, в частности в составной команде

dir | more

(хотя следует помянуть, что модифицированная версия команды dir позволяет добиться того же эффекта с помощью дополнительной опции, но при этом надо помнить написание опции, а возможности команды more более универсальны и применимы с любой другой командой).

В Unix для использования конвейеров существует дополнительная команда tee, которая в качестве единственного аргумента требует указания имени файла и передавая данные со своего стандартного ввода на стандартный вывод одновременно запоминает передаваемые данные в заданном для нее файле. Она как бы кроме простой передачи с входа на выход еще и протоколирует все данные в файле.

При использовании стандартного ввода-вывода иногда возникает проблема запоминания сообщений об ошибках. Для вывода информации об ошибках предназначен стандартный поток ошибок, имеющий константное значение хэндла в Unix и OS/2, равное 2, причем этот вывод об ошибках по умолчанию поступает на экран.

Для переадресации стандартного ввода и стандартного вывода, как уже описывалось в гл. 2, в командной строке используются вспомогательные символы '<' и '>' соответственно. Для переадресации стандартного потока ошибок используется запись вида

команда 2>*имя_файла*

Предоставляется также возможность направлять данные об ошибках в стандартный поток обычного вывода. Для этого предназначена запись вида

команда 2>&1

Наконец, существует возможность при необходимости перенаправить стандартный вывод в стандартный поток ошибок, что задается записью

команда 1>&2

При использовании переназначений с помощью символов обозначений >*имя_файла* предыдущее содержимое файла с именем *имя_файла* теряется (внутренними процедурами файл открывается для записи с усечением до нуля предыдущего содержимого). В ряде случаев, особенно при протоколировании ошибок, может представлять интерес сохранение предыдущего содержимого указанного таким образом файла и приписывание в его конец новых данных. Для решений этой задачи предлагается использовать вместо одного символа '>' два таких символа подряд и без разрыва между ними.

В частности команда

команда <*файл1* >*файл2* 2>>*файл3*

указывает – брать исходные данные в файле *файл1*, помещать данные стандартного вывода в файл *файл2* и дописывать в конец файла *файл3* сообщения стандартного потока ошибок текущей команды *команда*.

12.6. Командные файлы и сценарии

Компонент операционной системы, ответственный за взаимодействие с пользователем, это, как уже пояснялось, и есть командный интерпретатор. В достаточно совершенных ОС он позволяет не только задавать на выполнение отдельные команды или их небольшие группы, но и выполняет роль командной оболочки для программирования. Это программирование имеет дело не столько с вычислениями, сколько с действиями над файлами и процессами, но управляющие возможности такого программирования могут быть близки к возможностям современных алгоритмических языков. Как уже говорилось ранее, программы, задаваемые ко-

мандному интерпретатору, принято называть сценариями команд или командными файлами (иногда используется еще термин пакетные файлы).

В операционной системе MS DOS и основанных на ней модификациях Windows 9x такие пакетные файлы должны обязательно иметь расширения .bat, в операционных системах OS/2 и Windows NT командные файлы обязаны иметь расширение .cmd. В этих ОС запуск командных файлов требует просто указания их имени в качестве имени команды. Зато в Unix никаких ограничений на обозначения командных файлов (сценарием команд) не накладывается. Но для их запуска на выполнение возможны целых три пути. Первый из них заключается в запуске командного файла с помощью запускающей команды sh, так что такой запуск имеет вид

sh *имя_командного_файла*

Близким к нему, но более кратким вариантом является запись вместо команды просто единственного символа точки (так что символ точки используется в Unix в существенно различных применениях: обозначения текущего каталога и имени запускающей команды). Таким образом, предыдущая команда может быть также введена в эквивалентном виде

. *имя_командного_файла*

Наконец, имеется и возможность для прямого запуска командного файла исключительно по его имени, но перед этих запуском для командного файла должны быть установлены права доступа на выполнение. Практически следует лишь предварительно выполнить команду

chmod +x *имя_командного_файла*

и командный файл можно будет запускать непосредственно по имени.

В простейшем случае командный файл содержит последовательность команд операционной системы, разделенных переводом строк (символом '\n') или символом "точка с запятой" в любом сочетании. В более общем случае командные файлы могут содержать специально вводимые в них переменные и управляющие структуры.

Для краткости параллельно будем рассматривать возможности трех типов командных файлов: для интерпретаторов типа MS (для MS DOS/Windows 9x, OS/2 и Windows NT), для командного интерпретатора Unix, называемого bash (в модификации, применяемой в Linux), и для командного интерпретатора tcsh из Linux.

Остановимся вначале на вспомогательных средствах, неиспользование которых или неверное использование может буквально раздражать пользователя командного режима. Для вывода информации по ходу выполнения командного файла предназначена команда ECHO (во всех рассматриваемых оболочках). Для интерпретаторов типа MS есть замечательная возможность запретить при выполнении этой команды выдачу на экран текста самой этой команды, которая по умолчанию для наблюдения за ходом действий появляется на нем. Эта возмож-

ность заключается в задании служебного символа @ перед первым символом команды ECHO, так что последняя приобретает вид @ECHO.

Запрет на отображения текста других команд задается более глобально. Именно для установки режима отмены отображения команд предназначена команда ECHO OFF, которая действует до выполнения команды ECHO ON, после которой дальнейшие команды будут отображаться (до новой команды ECHO OFF, если она появится). Причем сама команда ECHO OFF еще отобразится на экране, если ее не задать в форме @ECHO OFF.

В командной оболочке bash в Unix команды, выполняемые из сценария команд, не отображают свой текст на экране в ходе такого выполнения. Текст команд отображается только, когда команды по одиночке (или в составе пакета, задаваемого разделителями ; && ||) задаются на выполнение в обычном режиме командной строки. Практически это снимает проблемы "отображать или не отображать". В командном интерпретаторе tcsh возможности управления отображением команд аналогичны интерпретаторам типа MS (точнее вторые используют более раннее техническое решение, появившееся в свое время именно в интерпретаторах типа tcsh). Эти возможности задаются в tcsh парой команд **set echo** и **unset echo**, причем первая из них устанавливает режим отображения текста каждой из следующих команд, а вторая отменяет режим такого отображения.

Команда echo в Unix имеет одну полезную опцию, задаваемую как -n. При наличии этой опции не происходит перехода на следующую строку вывода по завершению вывода текущей командой. Это дает возможность формировать единственную строку вывода, используя несколько операторов. (В отдаленных аналогиях эта возможность аналогично использованию оператора WRITE в Паскале вместо WRITELN или использованию оператора вывода printf, аргумент формата которого не содержит управляющего символа '\n'.)

Для вспомогательных целей программирования предназначены комментарии в командных файлах. Командные интерпретаторы типа MS рассматривают как комментарии строки, начинающиеся со служебного слова REM, а в Unix интерпретаторах для задания комментариев предназначен служебный символ #.

В Unix имеется возможность с помощью команды chsh изменить командный обработчик по умолчанию или вызывать командный файл, указывая явно командный интерпретатор в виде

bash *имя_командного_файла*

tcsh *имя_командного_файла*

Тем не менее, в общем случае для удобства пользователя возникает практическая проблема автоматического определения для какого из командных интерпретаторов предназначен конкретный сценарий команд. Для этих целей служит задание вида интерпретатора непосредственно в первой строке командного файла. Это делается с помощью служебной директивы

#!*имя_ программы_нужного_интерпретатора*

Так, задание вызова интерпретатора bash должно иметь в первой строке текст

#!/bin/sh

а для вызова интерпретатора tcsh текст

#!/bin/tcsh

(использовано стандартное размещение упомянутых программ интерпретаторов в файловой системе Linux).

Более-менее сложное программирование немыслимо без использования переменных. Поэтому все рассматриваемые оболочки содержат средства задания переменных. В оболочках типа MS переменные задаются с помощью вспомогательного служебного слова SET. Практически это слово открывает оператор командной строки, задающий значение переменной (это очень похоже на язык Basic, где для таких целей используется служебное слово LET). Оператор определения переменной имеет здесь вид

set *имя_переменной=значение*

где *имя_переменной* должно начинаться с латинской буквы, за которой могут идти как латинские буквы, так и цифры, *значение* представляет собой любой текст, в том числе начинающийся с пробела, который в такой случае считается также частью текста. Обязательно нужно запомнить, что пробелы между символом равенства и именем переменной не допустимы (сообщение об ошибке в таком случае отсутствует, но переменная оказывается неопределена). Все переменные в этих оболочках могут иметь только строковый тип.

В оболочке tcsh для Unix определение переменной строкового типа также задается с помощью служебного слова set. Причем в этом интерпретаторе символ присваивания (символ =) либо совсем не выделяется пробелами (ни справа, ни слева), либо выделяется пробелами с обеих сторон.

В оболочке bash для Unix определение переменной строкового типа задается более простой конструкцией вида

имя_переменной=значение

где пробелы между именем переменной, символом равенства и значением недопустимы (также появляются ошибки без явных сообщений о них). Причем *значение* в простейшем случае представляет просто последовательность символов, но такой вариант допустим только, если среди них нет пробела. При необходимости включить пробел в состав значения текстовой переменной, параметр *значение* должен задаваться в кавычках.

Для более тонких случаев в качестве символов, окаймляющих значение для переменной, используются символы прямых или обратных апострофов (символы ' и `). Предварительно заметим, что при использовании кавычек содержимое, заданное параметром *значение*, используется в применении переменной без всяких из-

менений и подстановок. Определение переменной с помощью апострофов приводит к подстановке в тексте значений переменных.

После определения значения переменной указанными выше способами эта переменная может быть использована в дальнейших командах. Такое использование требует не просто задания имени переменной, но ее имени с уточняющими символами. В интерпретаторах типа MS такое применение имени переменной требует записи вида

%*имя_переменной*%

а в интерпретаторах Unix в простейших случаях используется единственный служебный символ $ перед именем переменной. В более сложных случаях, требующих синтаксического выделения имени переменной из последующего текста, используется форма

${*имя_переменной*}

Заметим, что никакие дополнительные пробелы в этих формах не допустимы (пробелы являются синтаксически важными символами разделения элементов конструкций в командных файлах).

Если какая-то переменная не определена, т.е. для ее имени отсутствует оператор определения значения, то командный интерпретатор автоматически считает, что значение такой переменной есть пустая строка (строка совсем без символов). Задание использования такой неопределенной переменной в последующих строках командного файла не приводит к сообщениям об ошибках, а просто вместо нее абсолютно ничего не подставляется. Это надо хорошо помнить, так как из-за ошибок в воспроизведении имени переменной (в отличие от большинства языков программирования, хорошо известных студентам) никаких сообщений не возникает, а "по-тихому" возникает ошибка отсутствия данных.

Данное изложение знакомит только с простейшими средствами командных интерпретаторов операционных систем. Не рассмотренными остались многие другие, в том числе и достаточно используемые и полезные средства. Среди них – приемы использования переменных в командных интерпретаторах, ввод данных в сценариях команд, арифметические операторы, управляющие структуры в MS, управляющие структуры разветвления в Unix сценариях, управляющие структуры циклов в Unix сценариях. Их изучение рекомендуется продолжить по списку рекомендуемой литературы [1,7,11,24].

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. Банахан М., Раттер Э. Введение в операционную систему UNIX. - М:. Радио и связь, 1986. - 344 с.
2. Бек Л. Введение в системное программирование. - М: Мир,1988. - 448 с.

3. Гордеев А.В., Молчанов А.Ю. Системное программное обеспечение. -СПб.: Питер, 2001. - 736 с.
4. Дейтел Г. Введение в операционные системы. - М:. Мир, 1987. Т.1. -357 с., Т.2. - 398 с.
5. Кастер Х. Основы Windows NT и NTFS. - М:. Издательский отдел "Русская редакция ТОО Channel Trading Ltd", 1996. - 440 с.
6. Кейлингерт П. Элементы операционных систем. - М:. Мир, 1985. - 295 с.
7. Керниган Б.В., Пайк Р. UNIX – универсальная среда программирования. - М.: Финансы и статистика, 1992. - 304 с.
8. Краковяк С. Основы организации и функционирования ОС ЭВМ. - М.: Мир, 1988. - 480 с.
9. Крэнц Дж. и др. Операционная система OS/2. - М:. Мир, 1991. - 351 с.
10. Митчел М., Оулдем Дж., Самьюэл А. Программирование для Linux. Профессиональный подход. - М.: Изд. дом "Вильямс", 2002. - 288 с.
11. Петерсен Р. LINUX: Руководство по операционной системе. - К.: Изд. гр. BHV, 1997. - 688 с.
12. Рихтер Дж. Windows для профессионалов. - М:. Издательский отдел "Русская редакция ТОО Channel Trading Ltd", 1995. - 720 с.
13. Рихтер Дж. Windows для профессионалов: создание эффективных Win32-приложений с учетом специфики 64-разрядной версии Windows. - СПб:. Питер; М.: Изд.-торг. дом "Русская редакция", 2001. - 752 с.
14. Робачевский А. Операционная система UNIX. - СПб.: BHV - Санкт-Петербург, 1997. - 528 с.
15. Теренс Ч. Системное программирование на C++ для Unix. - К.: Издательская группа BHV, 1997. - 592 с.
16. Стивенс У. UNIX: взаимодействие процессов. - СПб.: Питер, 2002. - 576 с.
17. Столлингс В. Операционные системы. - М.: Изд. дом "Вильямс", 2002. - 848 с.
18. Флоренсов А.Н. Системное программирование в многозадачных ОС. Семантический подход: Учеб. пособие - Омск: Изд-во ОмГТУ, 2000. - 102 с.
19. Флоренсов А.Н. Введение в системное программирование для 32-разрядных компьютеров: Учеб. пособие. - Омск, Изд-во ОмГТУ, 1998. - 144 с.
20. Флоренсов А.Н. Программирование для графического интерфейса. Семантический подход: Учеб. пособие. - Омск, Изд-во ОмГТУ, 2003. - 128 с.
21. Фролов А.В., Фролов Г.В. Программирование для Windows NT. - М.: ДИАЛОГ-МИФИ, 1996, - 272 с. (Библиотека системного программиста; Т.26)
22. Фролов А.В., Фролов Г.В. Программирование для Windows NT: Ч. 2. - М.: ДИАЛОГ-МИФИ, 1997, - 271 с. -(Библиотека системного программиста; Т.27)
23. Хэвиленд К., Дайна Г., Салама Б. Системное программирование в Unix. Руководство программиста по разработке ПО. - М.: ДМК Пресс, 2000. - 368 с.

24. Попов А. Командные файлы и сценарии Windows Script Host. - СПб.: BHV-Петербург, 2002. - 320 с.

СОДЕРЖАНИЕ