Пензенский государственный университет

Егоров В.Ю.

Архитектура операционных систем

Курс лекций

УДК 004.451

Состав курса в основном сосредоточен на рассмотрении структуры и принципов работы операционных систем семейства Windows и составляющего их прикладного программного интерфейса Win32 API, базирующегося на использовании языка программирования C. Ссылки на интерфейс POSIX используются только в тех случаях, когда это необходимо для рассмотрения определённых базовых понятий. Также не стоит рассматривать это пособие как учебник непосредственно по программированию, либо справочник по системным вызовам, поскольку основной аспект в пособии уделяется рассмотрению принципов функционирования различных компонентов операционных систем. Упоминание тех или иных функций из состава системных интерфейсов в некоторых случаях даётся без описаний их аргументов, исходя из предположения, что аргументы системных вызовов могут быть в любой момент проанализированы онлайн.

Настоящий лекционный курс подготовлен доцентом кафедры «Вычислительная техника» Пензенского государственного университета Егоровым В.Ю. и предлагается для использования преподавателями и студентами по направлению подготовки 09.03.01 «Информатика и вычислительная техника». Объем представленного лекционного курса подобран с расчётом, что его состав в часах может варьироваться от 50 до 80 академических часов.

Определения в тексте выделены *курсивом*. Базовые понятия выделены **полужирным шрифтом**.

Содержание

Введение в операционные системы

Функционирование современного компьютера невозможно без полного комплекса программного обеспечения, включающего в себя системное программное обеспечение и прикладное программное обеспечение. Прикладное программное обеспечение включает в себя все программы, могут быть запущены под управлением операционной системы, за исключением специализированных утилит в её составе. Под системным программным обеспечением исторически понимают сами операционные системы, а также базовые средства разработки программ, такие как компиляторы и компоновщики.

Операционная система, — это система программ, предназначенная для обеспечения определенного уровня эффективности цифровой вычислительной системы за счет автоматизированного управления ее работой и предоставляемого пользователям набора услуг (ГОСТ 15971-84).

Компоненты операционной системы (ОС) обеспечивают управление вычислениями и реализуют такие функции, как планирование и распределение ресурсов, управление вводом-выводом информации, управление данными. Размер операционной системы и число составляющих ее компонентов в значительной степени определяется типом используемой аппаратной платформы компьютера, режимом работы компьютера, составом технических средств и т. д. [1]

Понятие **системного программного обеспечения** включает в себя операционную систему и комплект программ и утилит, обеспечивающих работоспособность операционной системы, конфигурирование и настройку операционной системы, а также возможность разработки, компиляции и компоновки программ, предназначенных для запуска под управлением операционной системы.

Цели применения операционных систем:

- увеличение пропускной способности компьютера (объем работ в единицу времени);
- уменьшение времени реакции системы;
- контроль работоспособности аппаратных и программных компонентов;
- обеспечение интерфейса с пользователем;
- управление программами и данными в ходе вычислений;
- обеспечение адаптивности к различным аппаратным средствам;
- обеспечение интерфейса с прикладными программами (так называемые, API).

Цели применения операционной системы в самом простейшем виде могут быть представлены как связующий компонент между пользователем, прикладным программным обеспечением и аппаратурой компьютера (рис. 1). При этом никакие два из перечисленных компонентов не могут взаимодействовать напрямую, минуя функционал операционной системы.

Рис 1. Операционная система как связующий компонент между пользователем, прикладным программным обеспечением и аппаратурой компьютера.

Операционные системы подразделяют на собственно операционные системы и системные утилиты, предназначенные для выполнения узкоспециальных действий (например, форматирование диска и пр.). Собственно операционная система делится на ядро операционной системы и прикладные программные интерфейсы (API, аббревиатура от Application Program Interface). Ядро операционной большей частью всегда присутствует в памяти. Программные интерфейсы могут загружаться и выгружаться по мере необходимости. Из состава ядра ОС можно выделить драйверы внешних устройств, которые всегда находятся в ОЗУ, но могут быть переконфигурированы на основании данных об аппаратуре компьютера (рис. 2). Однако, исторически драйверы все же относят к ядру операционной системы, хотя бы потому, что в некоторых из них драйверы могут быть жестко скомпилированы вместе с ядром (например, в операционной системе Linux).

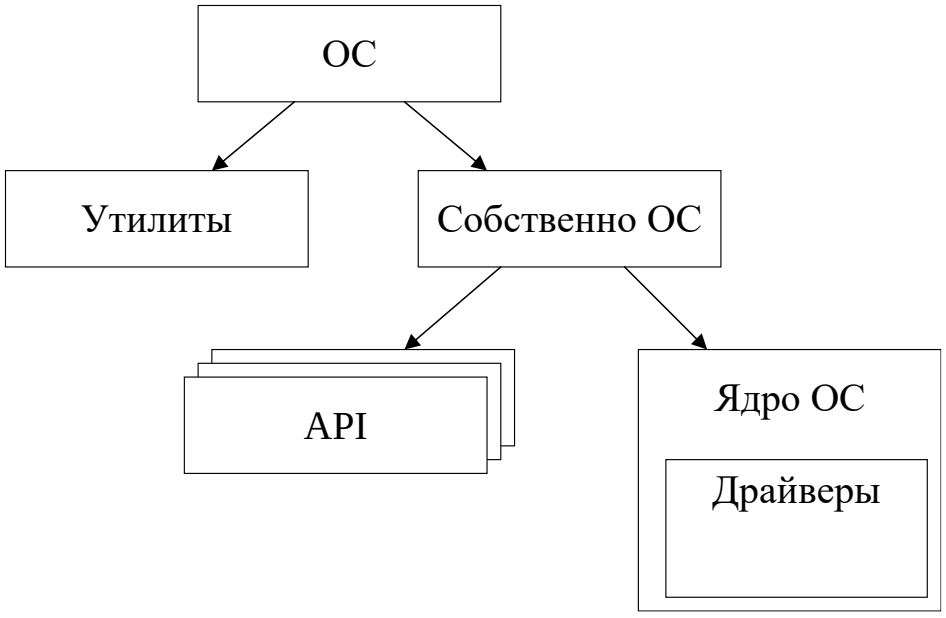

Рис. 2. Составные части операционной системы.

Более детальная структура собственно операционной системы представлена ниже.

Ядро операционной системы в укрупнённом виде включает в себя следующие компоненты:

- менеджер безопасности;
- менеджер памяти;
- подсистему синхронизации и взаимодействия процессами;
- дерево объектов системы;
- диспетчер задач;
- подсистему ввода-вывода.

Эти компоненты взаимодействуют между собой так, как показано на рисунке 3.

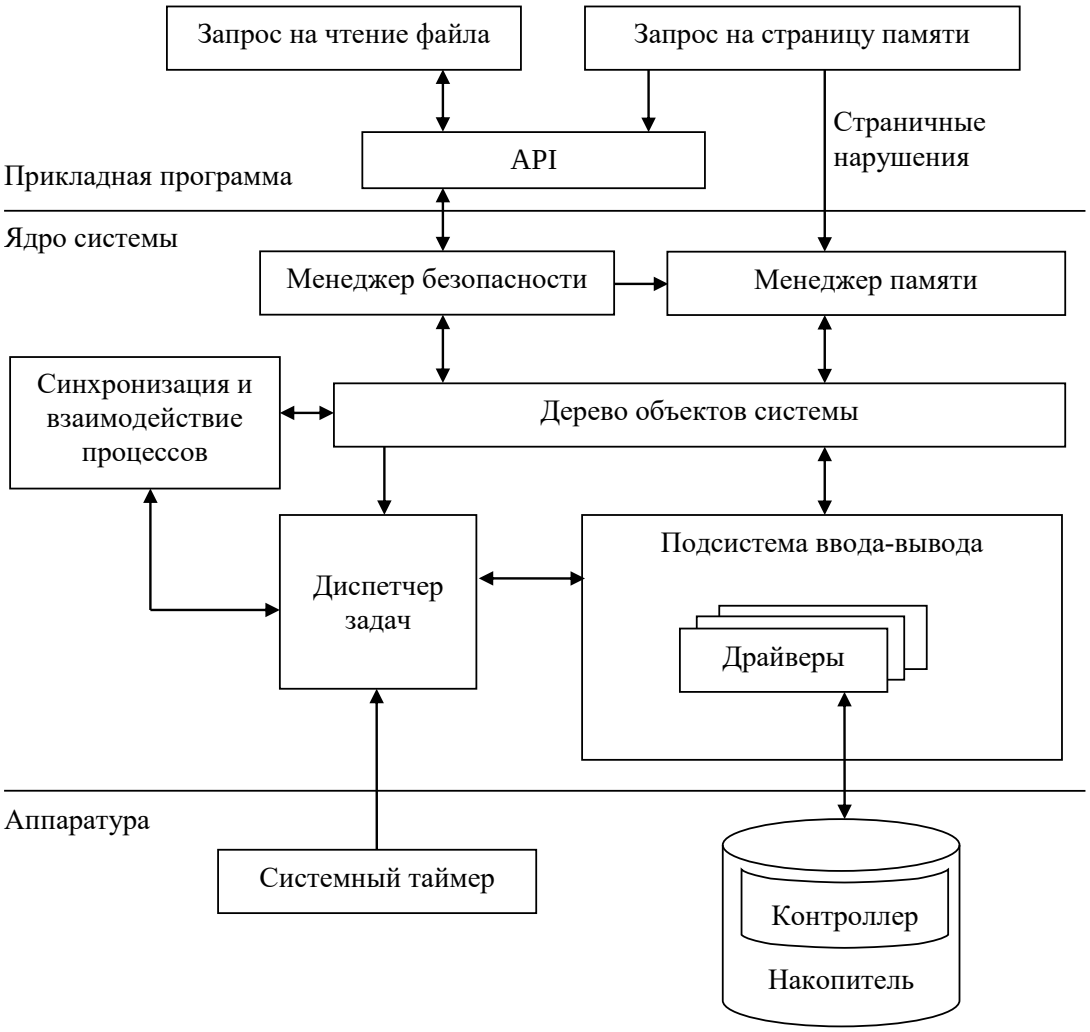

Рис. 3. Компоненты ядра операционной системы.

Диспетчер задач, получая прерывания от таймера, переключает центральный процессор (процессоры) между процессами. Процесс получает управление и по мере необходимости обращается к операционной система посредством системных вызовов. Для этого служат разнообразные прикладные интерфейсы. Интерфейсы передают управление в ядро операционной систему менеджеру безопасности. Менеджер безопасности проверяет корректность параметров функций, а затем обращается к спискам, попутно преобразуя описатели (HANDLE) в указатели на объекты системы. Затем управление получает менеджер ввода-вывода. Последний формирует данные для драйверов и передает их драйверам в составе ядра ОС. Драйверы работают с аппаратурой компьютера. Если аппаратуре требуется время на обработку запроса, то драйвер передает управление обратно диспетчеру задач системы. После того, как аппаратура заканчивает обработку запроса, она выставляет прерывание. Диспетчер задач обрабатывает прерывание и

передает управление драйверу. Драйвер заканчивает обработку запроса и передает управление выше, обратно менеджеру ввода-вывода и менеджеру безопасности. Последний копирует результат в адресное пространство процесса.

Во время своего функционирования процессу может потребоваться больше оперативной памяти. Это может произойти явно, и тогда процесс вызовет функцию выделения памяти, или неявно, и тогда произойдет исключение отсутствия страницы (например, если кончится память в стеке). В любом случае управление получит менеджер памяти и выделит процессу необходимое количество страниц оперативной памяти.

Классификация операционных систем.

Операционные системы различают по нескольким характеристикам:

1. Операционные системы **общего** и **специального** назначения. Операционные системы специального назначения в свою очередь подразделяются на операционные системы для переносных устройств и различных встроенных систем, ОС баз данных, ОС реального времени и др.
2. По режиму обработки задач различают операционные системы, обеспечивающие **однопрограммный** и **мультипрограммный режимы работы**. Под мультипрограммным понимают способ организации вычислений, когда на однопроцессорной вычислительной системе создается видимость одновременного исполнения нескольких программ. Иногда говорят о **мультизадачном режиме работы**, подразумевая под этим термином режим разделения времени. (Эти режимы мы рассмотрим в дальнейшем).
3. По организации работы в диалоговом режиме различают **однотерминальные** и **мультитерминальные** операционные системы.
4. По ограничению времени реакции операционные системы делятся на операционные системы **общего назначения** и операционные системы **реального времени**. У последних существуют ограничения на время реакции на входящие запросы. Это достигается несколькими способами, в зависимости от вида системы реального времени. Системы реального времени подразделяются на системы **мягкого реального времени**, когда допустима некоторая задержка между возникновением события и реакцией системы на его возникновение (в пределах 1-10 миллисекунд) и системы **жесткого реального времени**, когда требуется немедленная

реакция системы. Время реакции системы в системах реального времени разбивается на две составляющие: время отсечки, то есть время, когда система понимает, что истек тайм-аут на выполнение операции и время ответа, когда система начинает выполнять код по тайм-ауту. В общем случае для систем реального времени действует правило: если принятие решения не произошло в заданный интервал времени, то решение уже не нужно.

5. По способу организации ядра различают операционные системы **монолитного ядра** (например, Windows) и операционные системы с **микроядром** (например, QNX). Иногда в этой классификации выделяют понятие операционной системы с **гибридным ядром**, подразумевая под этим термином возможность загрузки и выгрузки драйверов в операционной системе с монолитным ядром.

Определения системного программного обеспечения

Базовые понятия

Прежде, чем продолжить изучение отдельных компонентов в составе операционных систем, необходимо дать определения некоторым общеупотребимым понятиям, которыми мы будем оперировать в дальнейшем.

Процесс *(process) – это выполнение отдельной программы с ее данными на последовательном процессоре.* [1]

Задача *(task) – это совокупность программных модулей и данных, требующая ресурсов вычислительной системы для своей реализации.* Существует также другое определение задачи. Задача, — это единица работы, для выполнения которой предоставляется центральный процессор. Вычислительный процесс может включать в себя несколько задач. Термин «задача» впервые был введён специалистами компании IBM, при появлении многозадачного режима работы компьютера.

Ресурс *– это всякий объект, который может распределяться внутри системы с течением времени.* Ресурсы могут быть разделяемыми, когда несколько процессов их используют одновременно или параллельно, а могут быть и неделимыми, в этом случае процессы используют ресурсы по очереди.

Изначально ресурсами считались процессорное время, память, каналы ввода-вывода, и периферийные устройства. Однако, впоследствии понятие ресурса стало шире. Теперь к ресурсам относятся также программные компоненты, информационные каналы, компоненты ОС, такие как описатели (handle) на объекты (таймеры, сигналы, семафоры и др.).

Пропускная способность системы*, – это количество стандартных задач, которые система может выполнить в единицу времени.*

В предыдущем разделе мы давали определение многозадачного режима работы. Многозадачный режим работы по определению считается лучшим по сравнению с однозадачным режимом работы. Давайте рассмотрим в чем его преимущество.

При выполнении задачи процессор часто простаивает, ожидая завершения выполнения операции ввода-вывода. Поскольку устройства ввода-вывода очень часто медленнее центрального процессора, это время весьма значительно. Мультизадачный режим работы позволяет исполнять другую задачу, пока первая ожидает завершения операции ввода-вывода.

На рисунке 4 представлены примерные графики работы вычислительной системы, функционирующей в однозадачном и многозадачном режимах.

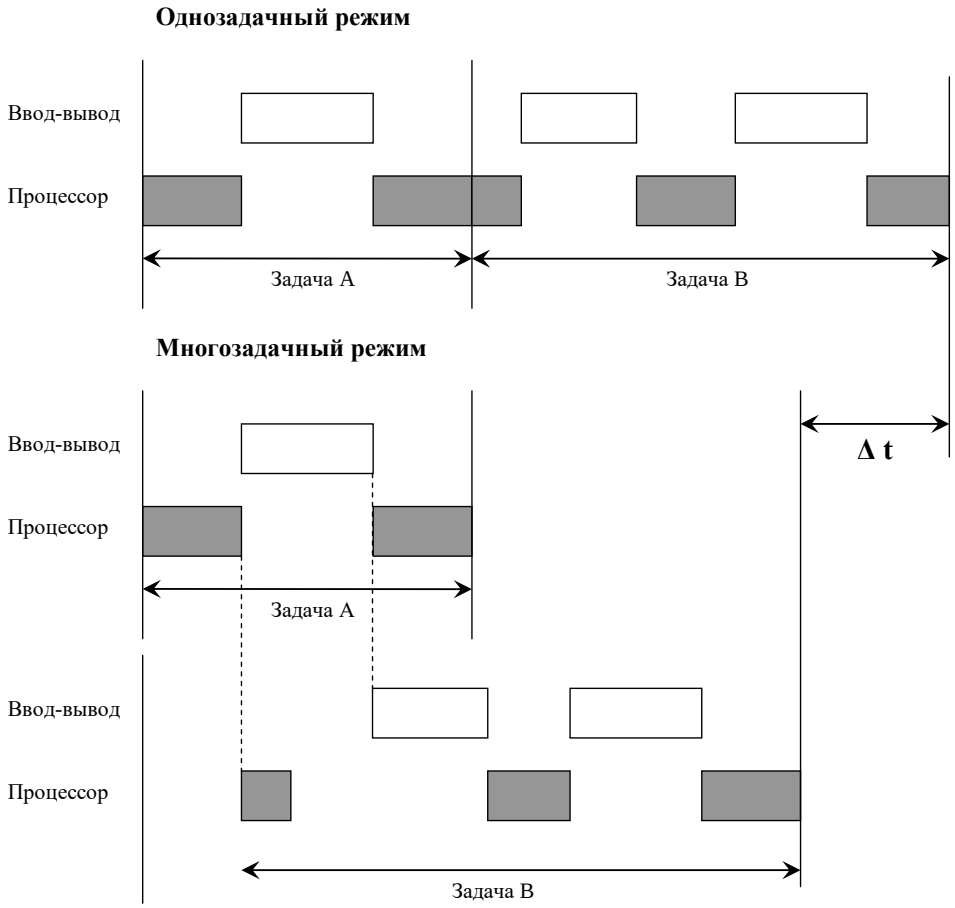

Рис. 4. Сравнительные временные характеристики однозадачного и многозадачного режимов работы.

На рисунке, поясняющем разницу между однозадачным и многозадачным режимом работы ясно видно следующее:

1. в многозадачном режиме работы задача В выполняется за БОЛЬШЕЕ время, чем в однозадачном;
2. время, затраченное системой на выполнение задач А и В МЕНЬШЕ в многозадачном режиме по сравнению с однозадачным на разницу Δt. Приведенный эффект называется **повышением пропускной способности** системы.

Таким образом, при наличии в системе только одной чётко поставленной задачи однозадачный режим работы является более предпочтительным, поскольку позволяет выполнять задачу быстрее. Но как только в системе появляется более одной задачи, сразу же

предпочтительным начинает становиться многозадачный режим работы, поскольку он повышает пропускную способность системы.

На приведенном рисунке обе задачи обращаются к одному и тому же ресурсу. Это становится легко осуществимым благодаря концепции виртуальных ресурсов.

Концепция виртуальных ресурсов.

Данная концепция позволяет унифицировать разработку программ и повысить эффективность использования ресурсов. Суть концепции: ОС использует все реальные ресурсы компьютера и на их основе создает новый, виртуальный компьютер с заранее определенными характеристиками. При этом ресурсы также становятся виртуальными. Каждый виртуальный ресурс моделируется с учетом его специфики. Так виртуальный процессор моделируется на реальном CPU путем выделения первому квантов времени, виртуальная память моделируется реальным ОЗУ на основе динамического преобразования адресов, устройства ввода-вывода используются преимущественно параллельно. В результате в рамках одного компьютера создаются несколько виртуальных вычислительных систем. Так для операционной системы Windows может быть организована вычислительная система (среда + программа), эмулирующая операционную систему MS-DOS. Для программ, работающих в самой Windows виртуальное адресное пространство равно 2 Гб для каждой программы, работающей в 32-битном режиме или 1 Тб для программ, работающих в 64-битном режиме. Однако в реальности используется имеющаяся в наличии оперативная память и файл подкачки страниц.

Нить и контекст нити

Итак, для каждого процесса в операционной системе создается собственная виртуальная вычислительная система. Однако, в рамках концепции виртуальных ресурсов можно пойти дальше и для каждой виртуальной вычислительной системы создать несколько виртуальных процессоров, каждый из которых будет выполнять свою собственную задачу. Выполнение такой задачи называется английским словом thread, что переводится на русский язык как «нить». В переводной литературе по вычислительной технике этот термин почему то переводят как «поток». В данном курсе лекций мы будем пользоваться термином «нить».

Таким образом, ***нить***, – *это виртуальный процессор, функционирующий в адресном пространстве процесса и исполняющий свою собственную задачу.*

Если компьютер реально располагает несколькими процессорами (процессорными ядрами), то две нити могут исполняться истинно

параллельно. Если процессор один, то параллельное выполнение эмулируется с помощью режима разделения времени. Поскольку все нити одного процесса выполняются в рамках этого процесса (то есть в рамках одной виртуальной вычислительной системы), то они разделяют одно и то же адресное пространство, и одни и те же глобальные переменные, имеют общие описатели на объекты ядра, на устройства ввода-вывода, на файлы и т.п. Это с одной стороны создает удобство при написании программ, а с другой требует от программиста особых навыков программирования. Кроме того, разные процессы также могут взаимодействовать между собой посредством специально разработанных механизмов межпроцессного взаимодействия.

Владение навыками программирования для параллельных процессов и нитей называется параллельным программированием. Эти навыки тем сложнее, что нужно не просто написать программу, которая будет работать с параллельными процессами и нитями, нужно написать ее так, чтобы нити взаимодействовали между собой наиболее выгодным способом с целью минимизации простоев оборудования компьютера.

Если у двух нитей в рамках одного процесса всё общее (память, глобальные переменные, описатели), то что же у них различно? Чем они отличаются? Ответ на этот вопрос заключает в себе очень глубокие обстоятельства, оказывающие влияние на всю структуру операционной системы.

У двух нитей одного процесса различаются следующие параметры:

1. регистры реального процессора и в том числе программный счетчик;
2. стек нити;
3. текущее состояние нити;
4. соотношение родитель-потомок.

В совокупности все перечисленные элементы называются **контекстом нити** или просто контекстом.

Естественно, что организация параллельно выполняющихся процессов и нитей была бы невозможна без аппаратной поддержки самой возможности их создания. Такая аппаратная поддержка в настоящее время реализована для нескольких аппаратных платформ. Так для процессоров семейства IA-32 от компании Intel такая поддержка в полном объёме была создана, начиная с процессора 80386. Для процессоров семейства ARM такая поддержка появилась в 5 версии платформы, но наибольшее распространение такие процессоры получили, начиная с 7 версии платформы, ARMv7.

Программные модули как ресурс

Реализации принципов параллельного программирования помогает представление программных модулей в виде особого рода ресурсов, которые могут использоваться нитями согласно определенному типу программного модуля.

Классификация программных модулей с точки зрения параллельного программирования выглядит следующим образом: программные модули делятся на используемые однократно или многократно. Однократно используемые программные модули разрушают себя в процессе выполнения и не могут быть использованы для параллельного программирования. Обычно такие модули используются при загрузке операционной системы. Многократно используемые программные модули делятся на привилегированные и непривилегированные. Привилегированные модули забирают на себя все ресурсы процессора путем отключения прерываний во время своей работы. Центральный процессор может переключиться на другой контекст только после завершения выполнения данного модуля. Из привилегированных модулей обычно состоит костяк ядра ОС.

Непривилегированные программные модули — это обычные программные модули, которые могут быть прерваны в процессе своего исполнения. В свою очередь делятся на нереентерабельные и реентерабельные. Нереентерабельные модули используют в процессе своего исполнения глобальные переменные, выделенные в куче процесса (или в пуле ядра системы). Поэтому, при переключении контекста внутри такого модуля нельзя допускать выполнение этого модуля другой нитью, иначе это приведет к краху процесса.

Реентерабельные модули не используют глобальных переменных. Для своего выполнения они используют только локальные переменные, заведенные в стеке. Поскольку стек входит в состав контекста, то прерывании приведет к переключению стека на стек другой нити и модуль может быть использован другой нитью. Пример реентерабельного модуля: функция вычисления абсолютного значения числа abs().

На практике, однако, практически невозможно создать реентерабельный модуль в классическом понимании этого термина. Чаще всего используется разновидность модулей, которые называются повторно-входимыми (от английского термина reentrance). Такие модули состоят из частей, некоторые из которых являются нереентерабельными, однако сам модуль является по сути реентерабельным. В составе таких модулей присутствуют части, которые называются критическими секциями. Внутри критической секции может находиться в один момент времени только одна

нить. Другие нити останавливаются на входе в критическую секцию и ждут своей очереди исполнения. Существует несколько способов реализации критических секций, к этому вопросу мы вернёмся позднее.

Повторно-входимые модули встречаются значительно чаще, чем реентерабельные. К истинно реентерабельным модулям относится только незначительное количество функций из стандартных библиотек. Практически все системные вызовы относятся к повторно входимым модулям. Пример повторно-входимого модуля – функция fopen() из состава библиотеки POSIX для языка программирования C.

Диспетчеризация задач

Обработка прерываний ядром операционной системы

Прерывания *– это механизм, позволяющий координировать параллельное функционирование отдельных устройств вычислительной системы и реагировать на особые состояния, возникающие при работе процессора, путем передачи управления от выполняемой программе к операционной системе и к соответствующему программному обработчику* [1].

Идея прерываний возникла в середине 50-х годов XX века и внесла огромный вклад в развитие вычислительной техники. Прерывания позволили реализовать асинхронный режим работы и параллельную работу устройств вычислительного комплекса.

Независимо от архитектуры системы обработка прерывания включает в себя следующие элементы:

1. Установление факта прерывания и его идентификация;
2. Запоминание состояния прерванного процесса (аппаратная часть контекста);
3. Аппаратная передача управления обработчику прерывания;
4. Запоминание программно остальной части контекста, которую не смогли запомнить на шаге 2;
5. Обработка прерывания путем вызова соответствующей подпрограммы;
6. Восстановление информации, обратное шагу 4;
7. Восстановление информации, обратное шагу 2.

Шаги 1-3,7 выполняются аппаратно, шаги 4-6 – программно.

Прерывания делятся на 2 класса: внешние и внутренние. Внешние вызываются асинхронными событиями, которые происходят вне вызывающего процесса: таймер, ввод-вывод, потеря питания, прерывание от другого процессора. Внутренние прерывания вызываются событиями, которые связаны с работой процессора и синхронны с его операциями: например, при нарушении адресации, при неизвестной операции, при делении на нуль, при попытке выполнить запрещенную в данном режиме команду и т. д. И, наконец, собственно программные прерывания, выполняющиеся по соответствующей команде прерывания. Эти команды нужны, например, для переключения процессора в привилегированный режим, либо для обработки программного прерывания.

Сигналы, вызывающие прерывания, могут возникнуть одновременно. Термин «одновременно», применительно к вычислительной технике, означает «в течение одного такта работы процессора». Выбор одного из прерываний для обработки осуществляется на основе приоритетов.

Приоритеты прерываний могут быть заданы аппаратно, программно, либо иметь смешанный характер.

Наличие сигнала прерывания не обязательно должно вызывать прерывание. Каждый процессор обладает средствами временного отказа от обработки прерываний: отключение обработки всех аппаратных прерываний, маскирование отдельных сигналов прерываний. Это позволяет операционной системе управлять возникновением прерывания на основе различных **дисциплин обработки прерываний**:

1. Дисциплина «с относительными приоритетами». Обслуживание не прерывается даже при наличии запросов с более высокими приоритетами. После окончания обслуживания данного прерывания обслуживается запрос с наивысшим приоритетом.
2. С абсолютными приоритетами. Всегда обслуживается прерывание с наивысшим приоритетом. При этом возможно многоуровневое прерывание.
3. По принципу стека: последний зашёл, первый обработан (LCFS – last come, first served).

Дисциплины 2 и 3 требуют полного запрета прерываний на шагах 1-4, 6,7.

В многозадачной операционной системе прерывания помимо своей основной функции реакции на возникновение особых состояний аппаратуры компьютера несут также дополнительную функцию. На основе прерываний функционирует диспетчер задач операционной системы. При каждом возникновении прерывания диспетчер задач анализирует необходимость переключения задач для текущего процессора. В том случае, если возникает необходимость переключения задачи, то это выполняется ещё до возврата из обработки прерывания. Поэтому в современных операционных системах можно разделить диспетчер задач на два компонента: так называемый, супервизор прерываний и собственно диспетчер задач (рис. 5).

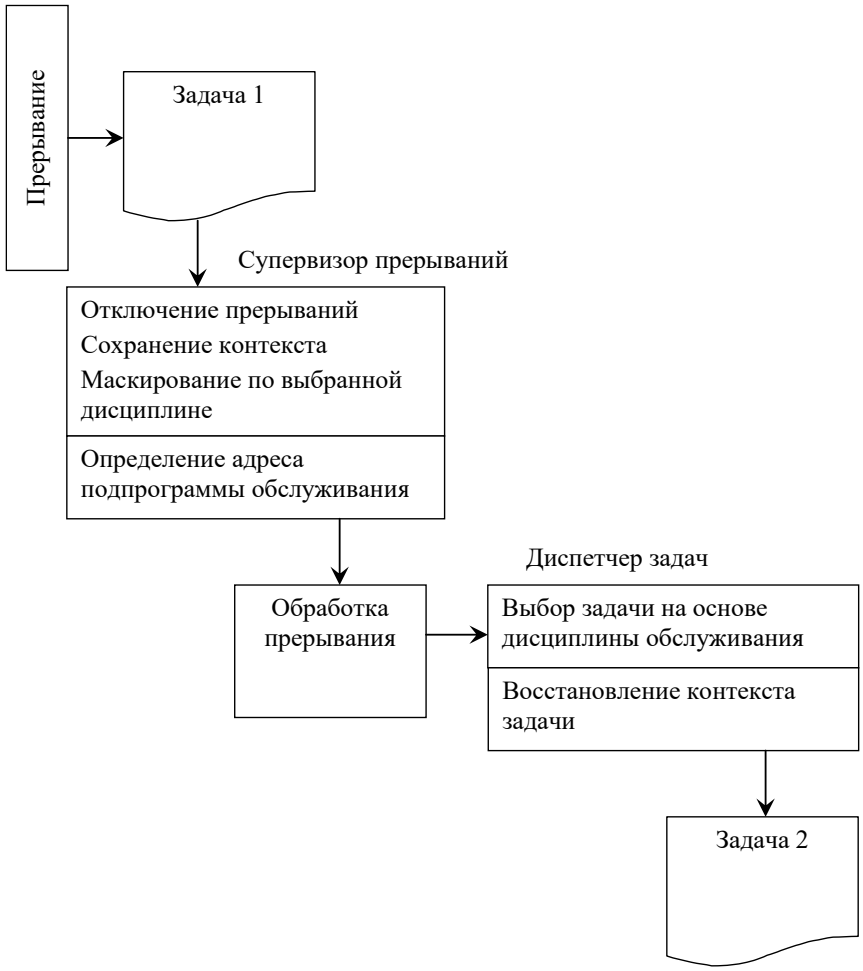

Рис 5. Последовательность обработки прерывания ядром операционной системы.

В многопроцессорных системах управление механизмом отказа от обработки прерываний в конкретном процессоре, таким образом, несёт в себе ещё одну дополнительную функцию: любой процессор может запретить прерывания и фактически перейти на время к однозадачному режиму работы, в то время как остальные процессоры системы продолжают работать обычным образом. Такой приём может использоваться в костяке ядра операционной системы.

Диспетчер задач

Диспетчер задач, — это одна из наиболее важных частей ОС. Она обеспечивает такие характеристики, как время реакции системы, скорость выполнения наиболее важных нитей и т. д. От того, как именно реализован диспетчер задач, зависит много характеристик операционной системы. Операционные системы различного типа имеют различные диспетчеры задач.

Диспетчер задач *– это компонент ядра операционной системы, обеспечивающий переключение процессора между задачами согласно выбранной дисциплине диспетчеризации.*

Диспетчер задач любой операционной системы реализует одну из возможных дисциплин диспетчеризации или их комбинацию. Ниже на рисунке 6 представлена иерархическая диаграмма дисциплин диспетчеризации. Это диаграмма имеет один недостаток — в ней не отражено разделение дисциплин диспетчеризации на вытесняющую и не вытесняющую многозадачность.

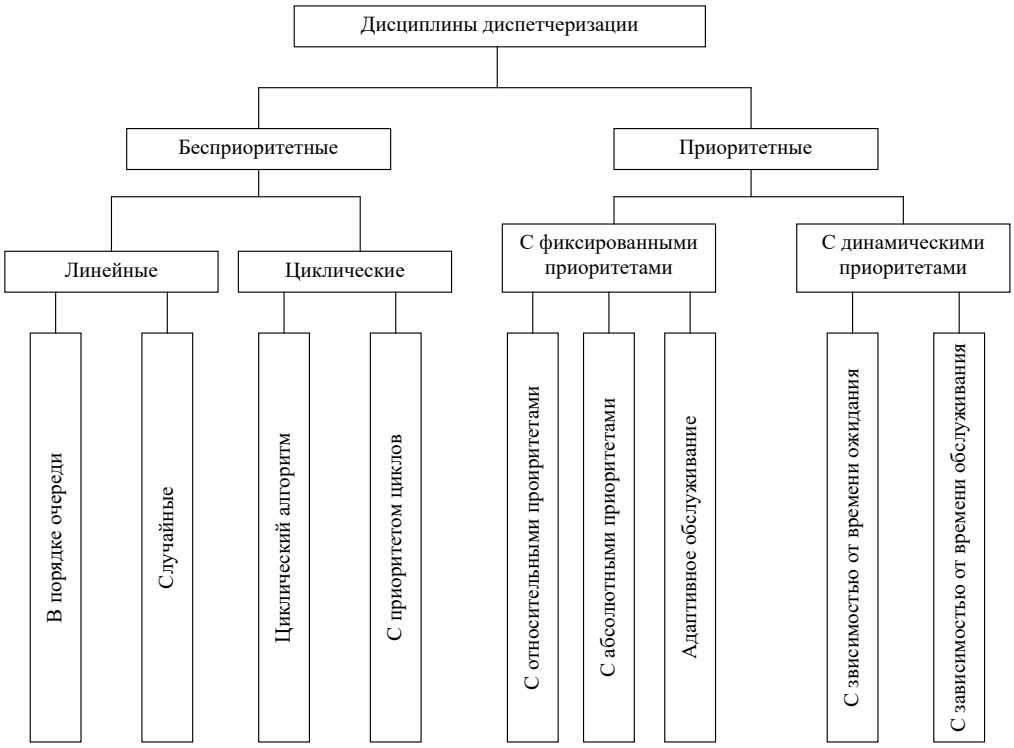

Рис. 6. Дисциплины диспетчеризации.

Как видно из рисунка 6, не существует единственной дисциплины диспетчеризации, которая удовлетворяла бы всем требованиям, предъявляемым к операционным системам. Поэтому для каждой конкретной операционной системы применяется та или иная комбинация дисциплин диспетчеризации задач.

Рассмотрим некоторые дисциплины диспетчеризации, начиная с самой простой.

Наиболее легкая в реализации дисциплина диспетчеризации, это дисциплина «в порядке очереди» (рис. 7). Многозадачность здесь не вытесняющая. Каждая нить работает до тех пор, пока сама добровольно не отдаст управление системе.

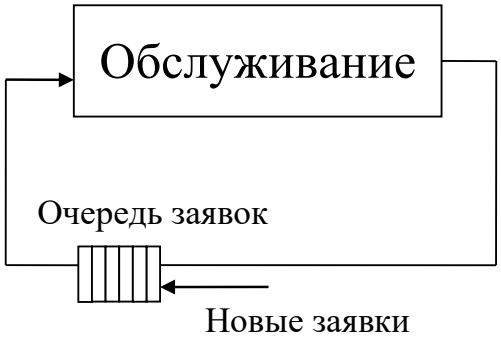

Рис. 7. Дисциплина диспетчеризации «в порядке очереди».

Недостатки такой схемы очевидны. При наличии ошибки в работе любой из программ, приводящей к бесконечному циклу, будет потеряна работоспособность всей операционной системы. Именно поэтому такая дисциплина диспетчеризации в настоящее время не используется.

Для реализации вытесняющей многозадачности используется другая дисциплина диспетчеризации — карусельная (английское название RR – Round Robin). Внешне рисунок будет выглядеть практически также, однако отличие в подходах принципиальное. Ни одна заявка не может привести к блокировке системы, поскольку обрабатывается не более чем заданный интервал времени обработки, который называется **квантом времени процессора** (рис. 8).

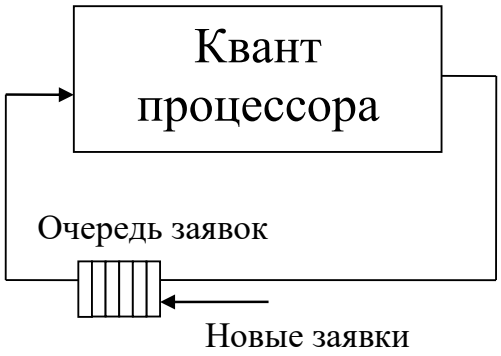

Рис 8. Карусельная дисциплина диспетчеризации.

Величина кванта времени процессора выбирается как компромисс между приемлемым временем реакции системы на запросы пользователя и накладными расходами на частую смену задач. В операционной Windows квант времени равен 20 мс для рабочих станций и 30 мс для серверов.

В операционной системе OS/2 (предшественницы Windows, совместной разработке IBM и Microsoft) величину кванта можно было настраивать вручную в специальном файле config.sys. Пример: параметр

TIMESLICE =32,356. Это минимальный и максимальный размеры кванта в миллисекундах. Размер кванта времени при этом динамически изменялся. Сначала нити давали 32 мс. Если за это время она сама не отдала управление, то в следующий раз 32*2 мс, затем 32*3 мс и т. д.

Карусельная – одна из самых распространенных дисциплин диспетчеризации. Однако в операционных системах реального времени обычно используют абсолютные приоритеты. В современных операционных системах чаще всего применяется совмещение карусельной дисциплины диспетчеризации и дисциплин с приоритетами.

Дисциплины с приоритетами, как было показано выше, могут быть различные. Однако, рисунок, иллюстрирующий применение дисциплин с приоритетами, будет примерно одинаковый (рис. 9). Различия заключаются в подходе выбора той или иной заявки на исполнение.

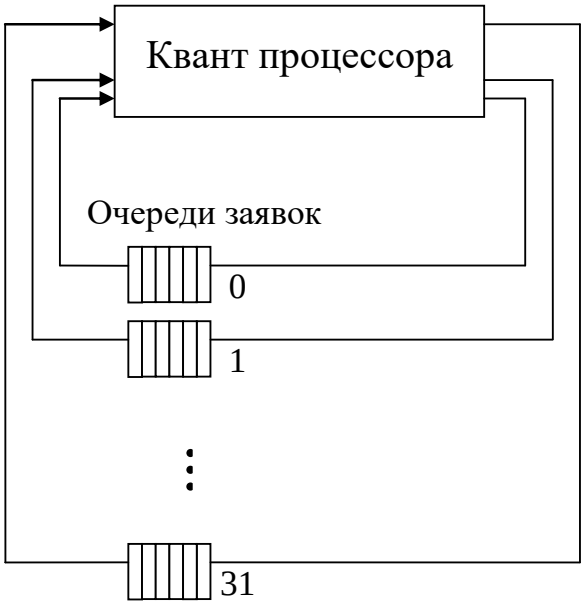

Рис. 9. Дисциплины диспетчеризации с приоритетами.

При использовании дисциплины диспетчеризации с абсолютными приоритетами сначала будет полностью опустошена очередь заявок с максимальным приоритетом (например, приоритет 0). Затем будет опустошена очередь заявок со следующим по порядку приоритетом 1 и так далее. Нетрудно видеть, что вероятность обработки заявок с минимальным приоритетом очень низка, поэтому на практике такая дисциплина диспетчеризации в классическом виде не используется.

При использовании дисциплин диспетчеризации с фиксированными приоритетами можно использовать другой способ, например, после обработки двух заявок с приоритетом 0, обрабатывать одну заявку с

приоритетом 1, после обработки двух заявок с приоритетом 1 обрабатывать одну заявку с приоритетом 2 и т. д.

Диспетчеризация задач в современных операционных системах

В операционной системе Windows все приоритеты разделены на два интервала. Более приоритетный интервал отвечает за работу ядра системы, в нем используется дисциплина с абсолютными приоритетами и невытесняющей многозадачностью. Менее приоритетный интервал предназначен для прикладных процессов. В нем используется карусельная дисциплина диспетчеризации с приоритетами, как показано на рисунке 10.

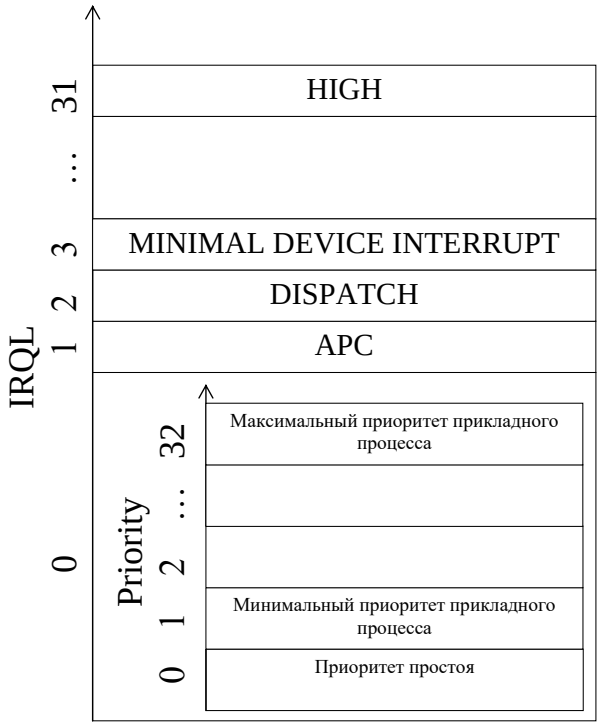

Рис. 10. Приоритеты в операционной системе Windows.

Использование динамических приоритетов позволяет реализовать гибкую стратегию для качественной диспетчеризации программ. В операционных системах реального времени есть понятие аварийного приоритета. Когда срок подходит к концу, а задача еще не выполнена, ей присваивается такой приоритет. В интерактивных ОС намеренно повышается приоритет нити, которая в данный момент осуществляет взаимодействие с пользователем, так называемая адаптивность обслуживания.

В операционных системах семейства Unix используется примерно следующий механизм распределения приоритетов. Предположим, что 0 —

это наинизший, 127 — наивысший приоритет, при этом для задач отводится интервал 0-65, для ядра 66-95, для задач реального времени 96-127.

При выходе из сна приоритет процесса максимален. Диспетчер каждую секунду повышает приоритет спящих процессов. А приоритет выполняющихся линейно уменьшает с каждым квантом. Текущий приоритет изменяется внутри отведенных границ. Фоновые процессы при этом имеют меньший приоритет автоматически.

В Windows используется другой способ. Приоритет нити задается приоритетом процесса и может немного колебаться относительно него. Приоритет процесса задается при его создании. Всего имеется 32 уровня приоритетов. 1-15 – приоритеты обычных процессов. 16-32 приоритеты реального времени. После выполнения кванта приоритет понижается и нить уходит в другую очередь. Если нить сама отдала управление, то ее приоритет не понижается. Если нить имеет активное окно, то к её приоритету прибавляется 1. (Любопытно, что в операционной системе Windows 10 компания Microsoft отказалась от повышения приоритета нитей процесса, имеющего активное окно). Программа в Windows также может сама изменять приоритет нити в интервале +2 -2. Если у нити нет активных окон и она выполняет громоздкие вычисления, не отдавая квант добровольно, то ее приоритет постепенно снижается. Если нить долго ждет, то ей начинает постепенно увеличиваться приоритет.

Критерии оценки дисциплин диспетчеризации

Какова бы ни была дисциплина диспетчеризации, ее требуется оценивать. Однако производить оценку эффективности дисциплин диспетчеризации разных операционных систем довольно затруднительно, в первую очередь из-за отсутствия единых механизмов выполнения такой оценки. Поэтому сравнительный анализ различных дисциплин диспетчеризации удаётся выполнить только для тех операционных систем, у которых имеются схожие системные интерфейсы.

Дисциплину диспетчеризации оценивают по качеству диспетчеризации и предоставлению гарантий обслуживания. Если используется дисциплина с абсолютными приоритетами, то возникает реальная дискриминация низкоприоритетных процессов и нитей. В истории известны случаи, когда процессы ждали обслуживания в течение нескольких лет и так и не были выполнены.

Для сравнения алгоритмов диспетчеризации исторически используются следующие критерии:

1. **Загрузка процессора** (CPU utilization). Измеряется в процентах. 2-3% — рабочая станция, 15-40% — ненагруженный сервер, 90-100% — нагруженный сервер.
2. **Пропускная способность** (CPU throughput). Число обрабатываемых заявок в единицу времени.
3. **Время оборота** (turnaround time). Общее время выполнения типового процесса.
4. **Время ожидания** (waiting time). Суммарное время, проводимое нитью в очередях.
5. **Время отклика** (response time). Время, прошедшее от запуска процесса до его появления на терминале.

Гарантировать обслуживание можно различными способами. Перечислим некоторые из них.

1. Каждому классу процессов выделять некую минимальную долю процессорного времени (например 20% от каждых 10 мс процессам реального времени, 40% от каждых 2 с. — интерактивным процессам, и 10 % от каждых 5 мин фоновым процессам).
2. Каждому процессу выделять долю процессорного времени, если он готов к выполнению.
3. Каждому процессу столько времени, чтобы закончить вычисления к сроку и не занимать ресурсы.

Синхронизация процессов и нитей

Основные понятия критических секций

Процессы называются **независимыми**, если их множества переменных не пересекаются. В противном случае процессы называются **взаимодействующими**. Взаимодействующие процессы совместно используют некоторые общие переменные и выполнение одного процесса может повлиять на выполнение другого.

Ресурсы, которые не допускают одновременного использования несколькими процессами, называются ***критическими***. Если нескольким вычислительным процессам необходимо пользоваться критическим ресурсом, им следует синхронизировать свои действия таким образом, чтобы ресурс всегда находился в распоряжении не более чем одного из процессов. Если ресурс используется одним из процессов, то остальные процессы, которым требуется этот ресурс, должны ждать, пока он не освободится.

Участок программы, в которой организуется доступ к критическому ресурсу, называется **критической секцией**.

Взаимодействующие процессы делятся на 2 класса: конкурирующие и сотрудничающие. Конкурирующие процессы действуют независимо, но имеют доступ к общим переменным. Сотрудничающие процессы работают так, что результат действий одного процесса передается другому по схеме «поставщик-потребитель». В значительном большинстве случаев взаимодействие нитей в рамках одного процесса является примером конкуренции. Существуют, однако, исключения из этого правила, когда взаимодействующие нити можно рассматривать по схеме «поставщик-потребитель».

Некорректное использование критических секций приводит к возникновению ошибок, поиск и исправление которых является одной из самых трудных задач в программировании. Критические секции являются одной из классических задач параллельного программирования. До настоящего момента интерес к ним не угасает.

Рассмотрим два примера, как неправильная организация взаимодействия двух нитей в рамках одного процесса может привести к краху программы. Первый пример намеренно содержит ошибки.

Пример 1

```
int x=0;
```

1 нить

```
while(x != 100) x++;
```

2 нить

```
for(;;) {
      if(x==100) {
            printf("%d", x);
            x++;
      }
}
```

Мы не можем утверждать, 2 нить выведет на экран число. Если вторая нить выведет число, то не обязательно она выведет именно 100. Мы не можем также утверждать, что число будет выведено однократно.

Пример 2

Имеется стандартный односвязный список. Первая нить производит добавление и удалении записей в списке. Вторая нить выполняет поиск по списку, как показано на рисунке 11.

Например, если первой нити необходимо выполнить занесение нового элемента в список, то возможны ситуации, когда поиск, выполняемый второй нитью, либо приведёт к краху процесса, либо из её рассмотрения на короткое время будет исключена вторая часть представленного списка.

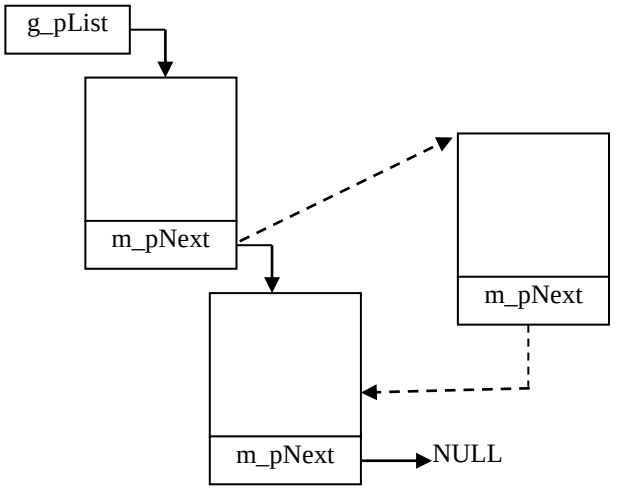

Рис. 11. Добавление элемента в односвязный список.

В общем случае без синхронизации с использованием критических секций результат работы нитей непредсказуем. В данном случае весь список является критическим ресурсом.

Приведенные примеры показывают наиболее частые ситуации, которые возникают у новичков в параллельном программировании. Чаще всего для разрешения подобных ситуаций используется механизм критических секций.

В одной программе может быть несколько критических ресурсов, доступ к которым организуется посредством критических секций.

Исключением из предложенной схемы, как уже упоминалось выше, является взаимодействие по схеме «поставщик-потребитель». Например,

одна нить заполняет переменную, другая нить использует значение переменной. В частности, вторая нить может производить отображение на экране прогрессивной шкалы копирования файла в процентах, значение которых заполняется первой нитью.

Для конкурирующих процессов и нитей имеется набор правил, нарушение которых будет приводить с некоторой вероятностью либо к краху процесса, либо к особой ситуации, называемой deadlock или тупик. Перечислим эти правила:

1. **Взаимное исключение.** В любой момент времени только один процесс должен находится в любой из критических секций, связанных с одним критическим ресурсом.
2. **Отсутствие блокировки.** Ни один процесс не должен находится в критической секции бесконечно долго.
3. **Возможность входа.** Ни один процесс не должен ждать бесконечно долго входа в критическую секцию (в том числе, решение, какой из процессов должен занять КС, не должно откладываться бесконечно долго).
4. **Вход по завершению процесса.** Если процесс, захвативший КС, завершается, в том числе аварийно, то КС должна освободиться.

Необходимо руководствоваться этими правилами при создании критических секций. Однако даже соблюдение этих правил может привести к тупику в особой ситуации, рассмотренной ниже (рис. 12).

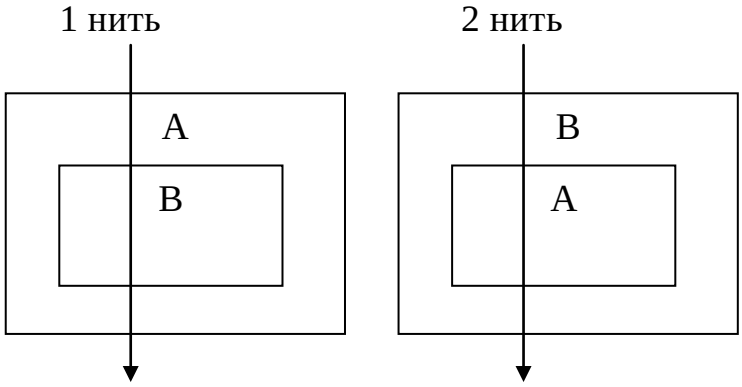

Рис. 12. Последовательный захват критических ресурсов двумя нитями.

Пусть имеются две нити, каждой из которых необходимо произвести обработку информации, связанной с двумя критическими ресурсами A и B. Для каждого из этих ресурсов в коде программы организована своя критическая секция. Отличия заключаются в том, что первой нити необходимо сначала получить доступ к критическому ресурсу A, а затем B, а второй нити, наоборот, необходимо сначала получить доступ к критическому ресурсу B, а затем A. Следствием подобной организации критических секций будет ситуация, с некоторой вероятностью приводящая

к тупику. Например, первая нить захватывает ресурс А и тут же теряет квант времени процессора, переходя в режим готовности к исполнению. Управление получает вторая нить, которая сначала захватывает ресурс В, а затем пытается захватить ресурс А, но, поскольку он захвачен первой нитью, вторая нить переходит в режим ожидания освобождения ресурса. Затем управление возвращается первой нити, которая пытается захватить ресурс В, но не может этого сделать, поскольку он захвачен второй нитью. В результате обе нити переходят в бесконечное ожидание, то есть налицо ситуация тупика.

Выход из проблемы заключается в последовательном захвате критических ресурсов обеими нитями в одинаковом порядке, например, АВ-АВ, либо ВА-ВА. Представленная проблема является частным случаем классической задачи информатики, которая носит название «задачи об обедающих философах» [2].

Решением проблемы обнаружения тупиков и выхода из них занимается целое направление в прикладной математике, которое возникло в период начала активного использования многопроцессорных компьютеров. Начало направлению было положено Карлом Петри в 1962 году. Оно называется «Сети Петри».

Принцип организации критических секций

Критические секции могут быть организованы полностью программно. К алгоритмам, позволяющим это сделать программно, относятся алгоритмы разрыва узла и поликлиники. Впервые задача критической секции была описана Дейкстрой в 1965 году. Он усовершенствовал алгоритм организации критических секций, предложенный Деккером. Этот алгоритм был еще усовершенствован Дональдом Кнутом в 1966 году. Алгоритм разрыва узла был изобретен Питерсоном в 1981 году и усовершенствован Блоком и Ву в 1990 году. Алгоритм поликлиники был изобретен Лампортом в 1974 году [2].

Однако, в современных вычислительных системах программные алгоритмы организации критических секций не используются. В частности, это происходит потому, что программные алгоритмы потребляют значительное количество процессорного времени и плохо масштабируются. В современных вычислительных системах используются алгоритм, который требует поддержки со стороны процессора и является, таким образом, программно-аппаратным.

В составе команд современных процессоров присутствуют специальные инструкции, объединяющиеся под общим названием «проверить-установить». Такие инструкции выполняют одновременно два

операции: считывание данных из памяти и занесение данных в память. Такая инструкция появилась впервые в платформе IBM360. Она называлась TS (test and set). Это была двухоперандная инструкция. Значение второго операнда присваивалось первому, а во второй записывалась единица.

В семействе процессоров x86 от компании Intel основная подобная инструкция – это инструкция XCHG. В данном контексте нас интересует разновидность этой инструкции, выполняющая работу с памятью. В течение выполнения данной инструкции регистр и память обмениваются значениями. Во время выполнения этой инструкции на шину управления всегда подается сигнал #lock вне зависимости от наличия или отсутствия префикса lock у инструкции. Этот сигнал блокирует доступ к оперативной памяти со стороны других устройств в составе вычислительной системы (контроллеры, другие процессоры).

Пусть ячейка памяти называется Status. Она содержит в себе статус доступа к критическому ресурсу. Когда Status равен 0 – критический ресурс свободен. Когда Status равен 1 – критический ресурс занят. Инициализация статусной переменной на языке ассемблера для платформы x86 будет выглядеть следующим образом:

```
mov Status,0
```

Для простейшего захвата доступа к критическому ресурсу используется следующая ассемблерная конструкция:

```
mov eax, 1
l1:  xchg eax, Status
cmp eax, 1
je l1
// КС
```

Для освобождения критического ресурса достаточно записать в Status значение 0.

```
mov Status,0
```

Если после выполнения инструкции XCHG в EAX единица, то значит КС занята другим процессом (нитью). Если в EAX значение 0, то это означает, что до выполнения инструкции XCHG критический ресурс был свободен, а теперь он занят нами и мы внутри критической секции. Сигнал #lock обеспечивает работоспособность способа на многопроцессорных вычислительных системах.

(Для некоторых аппаратных платформ, в частности, для платформы ARM, захват и освобождение критического ресурса требуют существенно большего количества ассемблерных инструкций, связанных в том числе, с барьерами кэширования данных. Платформа Intel в данном случае является наиболее наглядной и простой для понимания организации критических секций.)

Рассмотрим, почему нельзя заменять инструкцию xchg другими инструкциями. Для этих целей рассмотрим два варианта, которые, на первый взгляд, позволяют отказаться от применения инструкции xchg.

Вариант 1

В первом варианте мы просто заменяем инструкцию обмена данными xchg на две инструкции занесения данных mov.

```
mov eax, Status
mov Status, 1
```

В этом случае прерывание смены кванта возникает ровно посередине между двумя применяемыми инструкциями и приводит к краху. Например, две нити заносят в регистр eax предыдущее значение переменной Status, а затем одна из нитей засыпает, в то время как вторая изменяет это значение и входит в критическую секцию. Первая нить затем просыпается и анализирует значение в регистре eax, не подозревая, что на текущий момент оно не является актуальным, и тоже входит в критическую секцию, которая с этого момента перестала быть таковой.

Вариант 2

В этом варианте мы учтём опыт из предыдущего варианта и запретим обработку прерываний процессором с помощью инструкции cli и, тем самым, сделаем невозможным переключение кванта между двумя инструкциями mov.

```
cli
mov eax, Status
mov Status, 1
sti
```

Поскольку прерывания запрещены, то смены кванта не возникает, но другие процессоры всё ещё могут поменять значение переменной Status во время выполнения инструкции mov eax, Status и это снова приведет к краху. Именно поэтому использование специальной инструкции, (в нашем случае, xchg), является непременным условием программно-аппаратного способа организации критических секций.

Приведённый организации критических секций называется активной блокировкой (spinlock). Мы видим, что процессор постоянно опрашивает значение переменной Status. Это отрицательно влияет на производительность, поэтому такой способ блокировки используется только в ядрах операционных систем. В прикладных процессах используются более сложные объекты синхронизации, созданные как надстройка к активной блокировке.

Необходимо отметить, что в современных процессорах Intel существуют и другие инструкции, позволяющие организовать активную блокирвку. Это инструкции xadd, bts, btr, btc. При их использовании сигнал #lock на шине управления автоматически не выставляется и нужно

обязательно предварять их префиксом lock. (Результат bt_ инструкций заносится во флаг CF).

В операционной системе Windows имеется специальная функция, которая обеспечивает механизм создания критических секций через активные блокировки. Это функция IntelockedExchange [2].

```
LONG InterlockedExchange(
  LONG volatile *Target,
  LONG          Value);
```

Первый аргумент функции – адрес переменной, значение которой будет изменяться, второй аргумент – новое значение. Функция возвращает предыдущее значение переменной. Пример её использования:

Определение и инициализация:

```
long volatile Status = 0;
```

Использование:

```
while(InterlockedExchange(&Status, 1));
// Работает КС
Status = 0;
// Завершение КС
```

Однако такой способ в процессе ожидания будет потреблять все процессорное время одного процессорного ядра. Произведём модификацию способа:

```
while(InterlockedExchange(&Status, 1))
{
      Sleep(20); // Добровольная потеря кванта
}
// Работает КС
Status = 0;
// Завершение КС
```

Необходимо обратить особое внимание, что в приведённом примере переменная Status объявлена как volatile – изменчивая. Это ключевое слово заставляет компилятор никогда не производить перенос значения переменной в регистр при оптимизации кода.

Приведенный пример можно дополнительно усложнить проверкой нахождения в цикле активного ожидания не дольше заданного времени. В противном случае можно завершать ожидание ошибкой.

Помимо функции InterlockedExchange в операционной системе Windows также имеются функции, выполняющие другие операции с активной блокировкой. Перечислим некоторые из них:

InterlokedCompareExchange – блокированное сравнение и обмен при удовлетворении результатов сравнения.

IntelockedExchangeAdd – блокированный обмен с параллельным сложением.

IntelockedIncrement – блокированное увеличение значения переменной на единицу.

InterlockedDecrement – блокированное уменьшение значения переменной на единицу.

Объекты синхронизации

На основе активной блокировки можно создавать различные объекты синхронизации. В этом случае содержимое объекта синхронизации в ядре ОС представляется как критический ресурс, закрываемый для доступа с помощью критической секции. Сам объект синхронизации может выполнять различные функции. Рассмотрим несколько типов объектов синхронизации применительно к операционной системе Windows.

Каждый из объектов синхронизации в операционной системе Windows может в любой момент времени находиться в одном их двух состояний: сигнальном (SIGNALED) и несигнальном (NONSIGNALED). Когда объект находится в несигнальном состоянии, это означает, что он занят. Если какая-либо нить захочет захватить этот объект, то она вынуждена будет ожидать, пока объект не освободится [2].

Для ожидания любых объектов синхронизации в операционной системе Windows используется специальное множество функций, которое можно объединить общим термином «WaitFor». Это функции, название которых начинается со слов WaitFor: WaitForSingleObject, WaitForMultipleObjects, MsgWaitForMultipleObjects и другие.

Возвращаемое значение этих функций представляет собой беззнаковое целое число. Для него имеется несколько заранее заданных значений. Значение WAIT_OBJECT_0 означает, что нить захватила объект. Если объектов ожидания несколько, и нужно дождаться одного из них, то возвращаемое значение будет WAIT_OBJECT_0 + <номер объекта в массиве ожидания>. Значение WAIT_TIMEOUT возвращается, когда ожидание объекта закончилось тайм-аутом. Значение WAIT_FAILED сигнализирует об ошибке.

События как двоичные семафоры

Понятие семафора было впервые введено Дейкстрой в 1965 году [3,4]. Семафоры бывают двоичные и многозначные. Двоичные семафоры в ОС Windows называются событиями (Event). Многозначные семафоры в ОС Windows называются семафорами.

Рассмотрим объект Event (событие) в операционной системе Windows. Для него возможны два стандартных состояния: сигнальное (SIGNALED) и несигнальное (NONSIGNALED). Сигнальное состояние говорит, что путь

открыт и событие можно захватить нитью. Несигнальное состояние говорит, что путь закрыт и нужно ждать.

События создаются с помощью функции CreateEvent. События бывают с ручным сбросом и автоматические. События с ручным сбросом остаются в сигнальном состоянии даже после того, как нить продолжила свое выполнение (поезд проехал мимо семафора). Автоматические события сбрасываются сразу же после того, как ждущая нить продолжает свое выполнение.

Функции: SetEvent – перевести событие в сигнальное состояние. ResetEvent – перевести событие в несигнальное состояние. По сути событие можно представить, как булевскую переменную, принимающую два значения.

В интерфейсе POSIX операционной системы UNIX имеются объекты, называемые signal. Они представляют собой, так называемые активные сигналы, в противовес пассивным сигналам, реализуемым в Windows с помощью событий. Отличие активных сигналов от пассивных заключается в прерывании стандартной последовательности исполнения программы при возникновении активного сигнала, подобно тому, как стандартную последовательность исполнения команд изменяют прерывания. После срабатывания активного сигнала в POSIX управление передаётся специальному обработчику, сопоставленному с заданным видом сигнала. Некоторые из сигналов в POSIX несут на себе также функцию таймера [5].

Семафоры

Семафоры в отличие от событий могут принимать больше двух значений. Они создаются с помощью функции CreateSemaphore. Их можно представить, как целочисленную беззнаковую переменную. Изначально значение переменной обычно задаётся равным нулю. С помощью функции ReleaseSemaphore можно увеличить значение семафора на заданное число. Семафор считается находящимся в сигнальном состоянии, если значение его переменной больше нуля. Функции группы WaitFor уменьшают значение семафора на 1. Если значение равно 0, то семафор переходит в несигнальное состояние и происходит ожидание.

К недостатку реализации семафора в ОС Windows следует отнести невозможность получения текущего значения семафора без его изменения.

В ОС UNIX семафоры представляют собой более мощный объект ядра. В UNIX можно создать не один семафор, а массив семафоров и задать несколько операций над ними при исполнении одной атомарной функции работы с массивом семафоров [5].

Мьютексы

Впервые были изобретены разработчиками ОС Windows NT. Представляют собой двоичные семафоры с возможностью повторного входа. Любопытно, что изначально этот объект синхронизации назывался «mutant», но затем был переименован для более благозвучного названия [2].

В объекте мьютекса в структуру события добавлено поле «количество захватов» После того, как мьютекс перешел в несигнальное состояние, нить, захватившая мьютекс, может захватывать его еще несколько раз. Мьютекс переходит в сигнальное состояние, когда число захватов его текущей нитью становится равно нулю.

Функция CreateMutex создает мьютекс. Функции WaitFor захватывают мьютекс. Функция ReleaseMutex уменьшает количество захватов на 1.

У мьютексов есть замечательное свойство, которого нет у других объектов. Если нить, захватившая мьютекс, завершается, то мьютекс переходит в сигнальное состояние автоматически. Функции WaitFor у другой нити в этом случае вернут результат WAIT_ABANDONED вместо WAIT_OBJECT_0 при обычном захвате мьютекса.

В стандартном интерфейсе POSIX операционных систем UNIX мьютексов нет, однако они есть в дополнительной библиотеке PTHREADS.

Таймеры ожидания

В ОС Windows представляют собой объекты синхронизации, переходящие в сигнальное состояние, когда время тайм-аута истекло. По сути, являются событиями с привязкой ко времени.

Таймеры могут быть синхронизирующие и уведомительные по аналогии с автоматическими событиями и событиями с ручным сбросом. Таймеры могут быть также периодическими.

Таймеры ожидания создаются функцией CreateWaitableTimer. Время задается функцией SetWaitableTimer. Отменить счетчик времени можно с помощью функции CancelWaitableTimer.

Автоматические события, семафоры, мьютексы и синхронизирующие таймеры ожидания можно объединить в общую группу объектов синхронизации с автоматическим переходом в несигнальное состояние после захвата объекта.

Объект CRITICAL_SECTION

CRITICAL_SECTION – это структура, экземпляр которой создаётся непосредственно в прикладной программе. Этим она отличается от других объектов синхронизации, представленных выше, поскольку они создаются в ядре системы и операции над ними происходят посредством описателей.

Структура CRITICAL_SECTION представляет собой совокупность активной блокировки и события, используемого при длительном ожидании. При захвате объекта первые несколько тысяч циклов ожидания делаются через активную блокировку для ускорения входа в критическую секцию без обращения к ядру операционной системы. Если после выполнения этих циклов захватить объект не удается, то ожидание переводится на событие, что экономит процессорное время.

Функции объекта:

InitializeCriticalSection – инициализация структуры.

InitializeCriticalSectionAndSpinCount – инициализация структуры с одновременным заданием числа циклов ожидания активной блокировки.

EnterCriticalSection – захват объекта критической секции.

LeaveCriticalSection – освобождение объекта.

SetCriticalSectionSpinCount – задание числа циклов ожидания активной блокировки.

TryEnterCriticalSection – попытка мгновенного захвата объекта. В случае неуспешного захвата происходит немедленный возврат в вызывающую функцию.

DeleteCriticalSection – деинициализация структуры.

Процессы и нити

Процессы и нити представляют собой в том числе объекты синхронизации. Они находятся в несигнальном состоянии все время существования в качестве субъекта, то есть пока нить выполняется или у процесса имеется хотя бы одна нить. После своего завершения объекты процессов и нитей переходят в сигнальное состояние.

В операционной системе Windows имеются и другие объекты, которые могут функционировать как объекты синхронизации. Все они, как процессы и нити, по сути, выполняют функционал событий с ручным сбросом.

Ниже на рисунке 13 представлена простая схема иерархии стандартных объектов синхронизации.

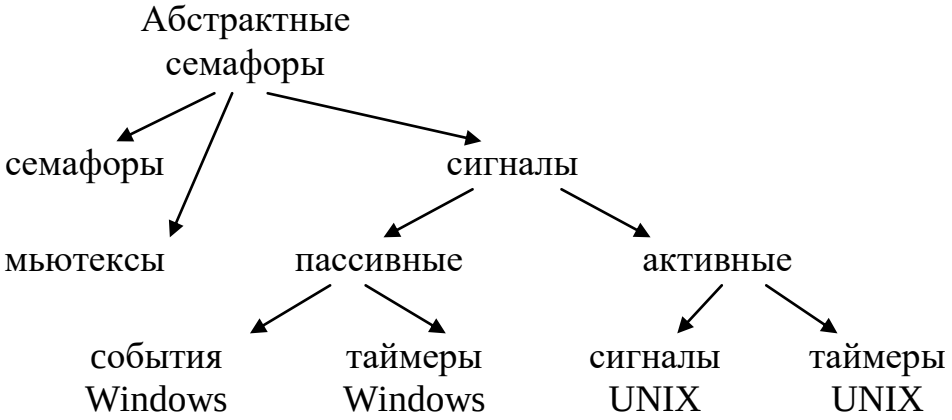

Рис. 13. Иерархия объектов синхронизации в современных операционных системах.

Как видим, все объекты синхронизации берут своё начало от абстрактных семафоров, предложенных Дейкстрой.

Более сложные виды объектов синхронизации

В этом разделе представлены объекты синхронизации, которых нет в системном API операционной системы Windows. Однако, все объекты этих видов можно организовать самостоятельно, используя имеющиеся стандартные объекты синхронизации.

Барьеры

Барьер представляет собой объект, который переходит в сигнальное состояние только в том случае, когда достигнуто определенное число ожидающих его процессов (нитей) [3, 4].

Барьер можно организовать как совокупность семафора и переменной счетчика. Каждый процесс, приходящий к барьеру, увеличивает значение счетчика и переходит к ожиданию семафора. Когда счетчик достигает порогового значения, последний процесс вместо перехода к ожиданию, наоборот переводит семафор в сигнальное состояние, отпуская все ждущие процессы, и обнуляет счетчик. Счетчик можно организовать через Interlocked-функции, либо весь объект барьера представлять в виде критического ресурса.

Мониторы

Представляют собой абстракцию, когда критический ресурс один, а критических секций по его обработке много. Абстракция монитора лучше всего представляется в виде класса в объектном языке программирования с

приватными данными. Работа с монитором происходит только из методов этого класса.

Очень удобно для организации мониторов использовать мьютексы. Пусть некий класс есть критический ресурс. Тогда все методы этого класса должны на входе захватывать мьютекс по доступу к критическому ресурсу, а на выходе освобождать. Благодаря возможности повторного входа тупика не происходит.

Яркий пример использования монитора – классическая задача информатики о спящем парикмахере.

Портфель задач

Этот объект синхронизации позволяет преобразовать рекурсивные вызовы в вызовы, параллельные по данным. Он был предложен (в статье с тремя авторами) в 1986 году. Другое название этого объекта синхронизации: thread pool (пул нитей). Принцип работы портфеля задач следующий.

Когда у процесса возникает необходимость выполнить ту или иную задачу, решение этой задачи оформляется в виде отдельной функции. Вместо того, чтобы вызвать эту функцию напрямую, процесс передает её в портфель задач и продолжает свою работу. В состав портфеля задач входят одна или несколько нитей, находящихся в состоянии ожидания. При поступлении задачи нить просыпается и выполняет работу путем вызова заданной функции из портфеля. После того, как все задачи из портфеля выполнены, нить в составе объекта либо переходит в состояние ожидания, либо завершается в зависимости от реализации.

Этот механизм синхронизации существует в ядре операционной системы Windows и задачи из портфеля называются WorkItem.

Читатели и писатели

Несмотря на то, что этот тип объектов относится к классическим задачам синхронизации, он является весьма востребованным объектом синхронизации, поскольку повышает пропускную способность системы по сравнению с обычным критическим ресурсом. Принцип работы следующий.

Доступ к базе каких-либо данных разделяется на два вида: модификация информации и чтение содержимого. Поэтому выделяются следующие операции над объектом: захватить на запись, захватить на чтение, освободить объект от записи, освободить объект от чтения. Модификацию информации в базе данных в один момент времени может делать только один процесс, а чтение – несколько одновременно. Повышение пропускной способности происходит за счет параллельных операций чтения.

Таким образом, задачи чтения конкурируют с задачами записи, а задачи записи конкурируют еще и между собой. Для реализации объекта нужно, чтобы читатели пропускали писателей вперед в очереди доступа, а писатели захватывали объект только, когда его не использует ни одна из задач.

Объект «читатели и писатели» реализован в ядре ОС Windows. Там он называется RESOURCE.

Работа с системным временем в операционной системе Windows

В операционной системе Windows время рассчитывается в 100 наносекундных интервалах. За нулевую отметку принята дата 1 января 1601 года, полночь [2]. Как известно, это дата начала действия Григорианского календаря. Чаще всего, для хранения такого представления времени в Windows используется структура FILETIME.

```
typedef struct _FILETIME {
  DWORD dwLowDateTime;
  DWORD dwHighDateTime;
} FILETIME, *PFILETIME;
```

Эта структура в принципе идентична структуре LARGE_INTEGER, которая используется для хранения 8-байтных чисел, но намеренно лишена представления, в котором с ней можно оперировать как с единым 8-байтным числом. (Так сделано для учёта аппаратных платформ, в которых байты располагаются в BigEndian последовательности, в отличие от стандартной LittleEndian последовательности байт.) Как следствие, сравнивать между собой две точки во времени необходимо не напрямую, а используя специальную функцию CompareFileTime.

```
LONG CompareFileTime(
  const FILETIME *lpFileTime1,
  const FILETIME *lpFileTime2);
```

Если результат выполнения этой функции больше нуля, то первое время позднее второго, если меньше нуля, то первое время раньше второго; если результат равен нулю, то времена идентичны. Однако, подобное представление времени крайне неудобно для человека, поэтому для использования времени в стандартном виде применяется структура SYSTEMTIME.

```
typedef struct _SYSTEMTIME {
  WORD wYear;
```

```
  WORD wMonth;
  WORD wDayOfWeek;
  WORD wDay;
  WORD wHour;
  WORD wMinute;
  WORD wSecond;
  WORD wMilliseconds;
} SYSTEMTIME, *PSYSTEMTIME;
```

Поля структуры:

wYear – год (от 1601 до 30827).

wMonth – месяц, в котором январь равен 1, соответственно декабрь имеет значение 12.

wDayOfWeek – день недели, начиная с воскресенья, которое имеет значение 0, суббота имеет значение 6.

wDay – день месяца от 1 до 31.

wHour – час от 0 до 23 часов.

wMinute – минута от 0 до 59.

wSecond – секунда от 0 до 59.

wMilliseconds – миллисекунда от 0 до 999.

Как видим, в структуре SYSTEMTIME время округляется до миллисекунд, поскольку более точное значение времени обычно не требуется для представления в понятном человеку виде. Программист может получить текущее системное время в виде структуры SYSTEMTIME напрямую, используя функцию GetSystemTime.

```
void GetSystemTime(
  LPSYSTEMTIME lpSystemTime
);
```

В этом случае возвращаемое время будет временем по Гринвичскому меридиану. Для получения текущего времени для имеющегося часового пояса программисту можно вызвать функцию GetLocalTime.

```
void GetLocalTime(
  LPSYSTEMTIME lpSystemTime
);
```

Можно также получить текущую точку во времени по Гринвичу сразу в виде структуры FILETIME с помощью функции GetSystemTimeAsFileTime.

```
void GetSystemTimeAsFileTime(
  LPFILETIME lpSystemTimeAsFileTime
);
```

Однако, для того, чтобы установить таймер на заданную точку во времени, перечисленных функций недостаточно. Для установки будильника программист должен сначала самостоятельно заполнить содержимое структуры SYSTEMTIME, а затем преобразовать её в структуру FILETIME с помощью функции

```
BOOL SystemTimeToFileTime(
  const SYSTEMTIME *lpSystemTime,
  LPFILETIME       lpFileTime
);
```

Имеется и обратная ей функция FileTimeToSystemTime.

```
BOOL WINAPI FileTimeToSystemTime(
  _In_  const FILETIME     *lpFileTime,
  _Out_       LPSYSTEMTIME lpSystemTime
);
```

После преобразования представления времени из вида SYSTEMTIME в вид FILETIME необходимо произвести ещё одно преобразование, прежде, чем устанавливать таймер. Теперь нужно преобразовать текущее локальное время к его представлению по Гринвичскому меридиану с помощью функции

```
BOOL LocalFileTimeToFileTime(
  const FILETIME *lpLocalFileTime,
  LPFILETIME     lpFileTime
);
```

Вот теперь всё готово для вызова стандартной функции установки таймера ожидания SetWaitableTimer.

```
BOOL WINAPI SetWaitableTimer(
  _In_           HANDLE           hTimer,
  _In_     const LARGE_INTEGER    *pDueTime,
  _In_           LONG             lPeriod,
  _In_opt_       PTIMERAPCROUTINE pfnCompletionRoutine,
  _In_opt_       LPVOID           lpArgToCompletionRoutine,
  _In_           BOOL             fResume
);
```

Вторым аргументом функции необходимо передать результат всех преобразований точки во времени, но уже (что довольно странно) в виде адреса стандартной структуры LARGE_INTEGER. Поля этой структуры нужно при этом заполнять по 4 байта, отдельно младшую и старшую часть. Сам таймер должен быть предварительно создан с помощью функции CreateWaitableTimer и его описатель нужно передать как значение первого аргумента функции SetWaitableTimer.

Третий аргумент функции SetWaitableTimer – это период, который можно задать равным нулю для однократного срабатывания таймера, либо указать в нём необходимое значение периода. Причём, значение периода задаётся не 100 наносекундных интервалах, а в миллисекундах.

Для задания точки во времени не относительно 1601 года, а относительно текущего значения времени, преобразования намного проще. Для того, чтобы таймер срабатывал подобным образом, время срабатывания нужно указывать отрицательное. Например, для задания времени ожидания в две минуты, необходимо структуру LARGE_INTEGER заполнить значением -10*1000*1000*60*2. (2 минуты * 60 секунд * 1000 миллисекунд * 1000 микросекунд * 10 стонаносекундных интервалов).

У функции SetWaitableTimer имеется ещё одна интересная особенность. По срабатыванию таймера ожидания как объекта синхронизации можно автоматически поставить на вызов специальную асинхронную APC функцию. (Аббревиатура от Asynchronous Procedure Call) как 4 параметр SetWaitableTimer. Описание APC функций будет дано позднее, при рассмотрении файловых операций.

Передача данных между процессами

С точки зрения передачи данных между процессами каждый из механизмов синхронизации процессов, рассмотренных в предыдущем разделе, является механизмом передачи одного бита информации. За передачу информации большего объема ответственны другие механизмы.

Все многообразие механизмов передачи данных между процессами можно условно свести к двум типам: прямая передача данных и опосредованная передача данных на основе промежуточных механизмов хранения данных, предоставляемых ядром операционной системы.

Прямая передача данных

К механизмам прямой передачи данных относятся: разделяемая память (и разделяемая секция в файле), а также прямое чтение-запись памяти процесса.

Разделяемая память

Механизм разделяемой памяти является самым быстрым механизмом передачи данных между процессами (рис. 14). При его использовании скорость передачи может составить многие гигабайты в секунду.

Организуется в системе Windows посредством вызова последовательности функций:

CreateFileMapping – создание области памяти либо на базе имеющегося файла, либо на базе памяти в своппинге.

MapViewOfFile – подключение созданной области памяти к адресному пространству процесса.

Отключение процесса от разделяемой памяти организуется посредством вызова последовательности функций:

UnmapViewOfFile – отключение разделяемой памяти от адресного пространства процесса.

CloseHandle – стандартная функция закрытия описателя, в данном случае описателя на разделяемую память.

По сути, разделяемая память представляет собой набор страниц физической памяти, отображенных одновременно в два и более процесса в системе. Набор страниц может отображаться в режимах READONLY, READWRITE и WRITECOPY. Режим READONLY применяется для отображения данных только для чтения, режим READWRITE применяется для чтения и записи данных. Наконец, режим WRITECOPY позволяет свободно читать данные, но при попытке их записи система сразу же создаёт

новую копию объекта разделяемой памяти, и дальнейшая работа будет производиться процессом в отдельной копии разделяемой памяти.

Для целей передачи данных между процессами наиболее часто используется режим READWRITE, позволяющий читать и писать данные.

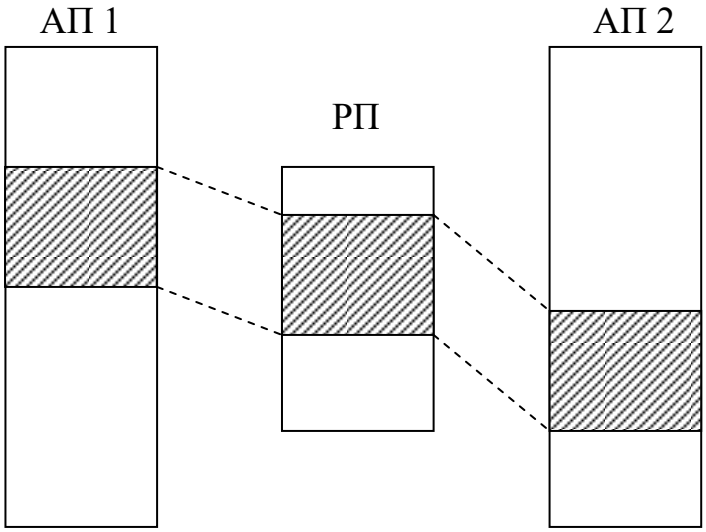

Рис. 14. Принцип работы разделяемой памяти.

Синхронизацию доступа к разделяемой памяти можно создать на основе простейшего механизма IntelockedExchange, используя в качестве статусной переменной любые 4 байта в составе самой разделяемой памяти. В этом случае все процессы, участвующие во взаимодействии, смогут получить доступ к переменной статуса критической секции. Также синхронизацию доступа к разделяемой памяти можно организовать посредством любого подходящего объекта механизмов синхронизации.

Для организации обмена между процессами посредством дейтаграмм (сообщений данных) можно еще 4 байта разделяемой памяти отвести под номер текущего сообщения. Можно также в составе разделяемой памяти предусмотреть поля назначения (от какого процесса какому процессу или процессам).

Достоинства метода:

- Высокая скорость передачи данных.
- Простота организации.

Недостатки:

- Структурирование информации ложится на программиста.

Прямое чтение-запись памяти процесса

Передачу данных посредством прямого чтения и записи памяти процесса (рис. 15) можно организовать только в том случае, если оба

процесса имеют полномочия для выполнения таких операций, например, запущены от лица одного и того же пользователя.

Для осуществления операции передачи данных вполне достаточно использования только одной из функций: чтение памяти процесса, запись памяти процесса. Как именно будет производиться обмен данными, определяется алгоритмом работы для конкретной реализации метода.

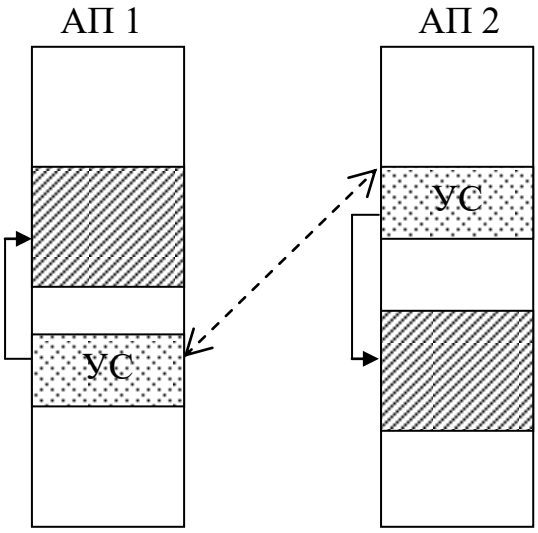

Рис. 15. Прямое чтение и запись памяти процесса.

Адресное пространство каждого из процессов, участвующих в обменен данными, включает в себя управляющую структуру и поля данных. Процессы перед передачей информацией обмениваются адресами управляющих структур в адресном пространстве друг друга.

В состав управляющей структуры входят указатели на поля данных, которые передаются тому или иному процессу. Другой процесс сначала считывает управляющую структуру, а затем при необходимости поля данных.

Чтение и запись памяти в операционной системе Windows организуются функциями:

ReadProcessMemory – прямое чтение памяти процесса.

WriteProcessMemory – прямая запись памяти процесса.

Достоинства метода:

- Высокая скорость передачи данных, но меньшая, чем у разделяемой памяти
- Очень высокий уровень безопасности передачи данных за счёт отсутствия у ядра операционной системы и других процессов в системе информации о структуре данных.

Недостатки:

- Структурирование информации ложится на программиста.

Опосредованная передача данных

При опосредованной передаче данных последняя происходит при участии механизмов ядра операционной системы. В общем случае передаваемые данные буферизируются операционной системой. Опосредованная передача данных включает в себя следующие механизмы: почтовые ящики, файлы, конвейеры.

Почтовые ящики

Почтовые ящики – один из наиболее распространенных механизмов передачи информации между процессами [1, 4]. К почтовым ящикам относятся очереди сообщений, в том числе оконные сообщения Windows и msg в ОС UNIX, дейтаграммы при передаче данных через сеть, LPC каналы в ОС Windows.

Почтовые ящики обеспечивают обмен сообщениями. Часто реализуются как один из механизмов передачи данных в ядре операционной системы. Почтовые ящики включают в себя набор буферов сообщений и служебную информацию о почтовом ящике.

Почтовые ящики можно разделить на классы с точки зрения доступа к почтовому ящику [1]:

- Привязка к процессу-получателю;
- Привязка к процессу-отправителю;
- Привязка к паре взаимодействующих процессов;
- Привязка к имени.

В простейшем случае почтовый ящик работает только в одном направлении. Отправитель отсылает сообщение и продолжает свою работу, не дожидаясь доставки сообщения получателю. Получатель время от времени заглядывает в почтовый ящик и забирает сообщения. Возможны варианты, когда получатель входит в ожидание, когда почтовый ящик пуст, а отправитель входит в ожидание, когда почтовый ящик переполнен.

Возможен вариант организации почтового ящика с использованием уведомления о доставке.

Существует 4 стандартных операции с почтовым ящиком;

1. SEND_MESSAGE Отправка сообщения получателю.
2. WAIT_MESSAGE Получение ответа получателем.
3. SEND_ANSWER Отправка уведомления о доставке от получателя отправителю.
4. WAIT_ANSWER Получение уведомлении о доставке со стороны отправителя.

В том случае, если используется первые 2 операции, то почтовый ящик является асинхронным. При использовании всех 4 операций почтовый ящик становится синхронным.

В общем случае для почтового ящика можно дополнительно задать требуемую дисциплину обслуживания сообщений: FIFO, LIFO, приоритетный, произвольный доступ.

Достоинства метода:

- Процессу не нужно знать о существовании других процессов, пока он не получит от них сообщение;
- Два процесса могут за один раз обменяться более чем одним сообщением;
- Операционная система может гарантировать приватность сообщений. Высокий уровень безопасности достигается особенно при использовании привязки почтового ящика к паре взаимодействующих процессов.
- Очередь буферов позволяет отправителю продолжать работу, не ожидая получателя (возможность асинхронной передачи данных).

Недостатки:

- Возможность переполнения буферов сообщений.

Файлы

Информацию между параллельными процессами можно передавать с помощью файлов. При этом информация может передаваться как через содержимое файла, так и через атрибуты файла, а также через факт доступности файла на запись. В последнем случае файл становится объектом синхронизации.

Когда файл открывается на запись, механизмы операционной системы позволяют блокировать доступ к файлу со стороны других процессов. Посредством файлов можно передавать огромные массивы данных.

Достоинства метода:

- Большой объем передаваемой информации;
- Сохранность информации при внезапной потере питания;
- Доступ к файлу контролируется операционной системой, в том числе с точки зрения прав пользователя.

Недостатки:

- Относительно медленная скорость передачи данных;
- Необходимость очистки данных (обнуления размера файла).

Конвейеры

Конвейеры решают классическую задачу производители-потребители. В операционной системе UNIX существует специальный объект передачи данных, создаваемый с помощью функции pipe, который представляет собой классический конвейер. Каждому процессу, работающему с конвейером, выдается по 2 описателя: на чтение и на запись информации. Чтение и запись информации происходят с побайтной гранулярностью с помощью стандартных функций чтения и записи read и write, которые используются для файлов.

Конвейер всегда однонаправленный (рис. 16). Он организован в виде кольцевого буфера. В ОС UNIX для каналов, созданных функцией pipe размер буфера равен 64 кб по историческим причинам. Организация конвейера очень простая: имеется два указателя, на голову (head) и на хвост (tail) данных. Запись данных всегда происходит по указателю на голову, а чтение осуществляется по указателю на хвост. Когда какой-либо из указателей доходит до конца буфера, хранящего данные, то указатель перемещается на начало буфера и операция продолжается.

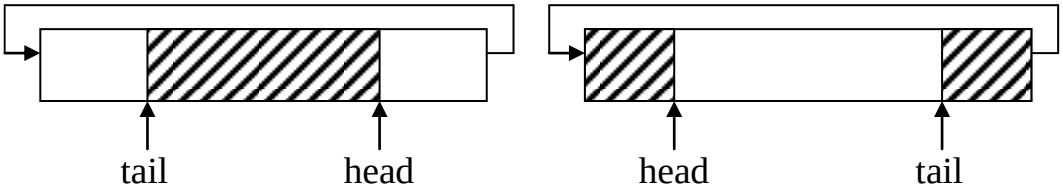

Рис. 16. Организация конвейера.

Если голова и хвост данных конвейера указывают на один адрес, то в конвейере данных нет. Конвейер также может быть переполнен, в этом случае можно либо завершить операцию записи с ошибкой, либо дождаться освобождения конвейера.

Достоинства метода:

- Кольцевой буфер, позволяющий не заботиться об очистке данных;

Недостатки:

- Низкая скорость;
- Небольшой размер буфера.

Менеджер безопасности

Этот раздел призван продемонстрировать основные принципы организации менеджера безопасности на примере операционной системы Windows. Информация в этом разделе отнюдь не является полной, – её основное предназначение заключается в освещении методов контроля над данными и реализации прав доступа.

В операционной системе Windows реализована, так называемая, дискреционная схема контроля доступа. Основные принципы этой схемы довольно просты. При осуществлении доступа к информации всегда имеется субъект, который получает доступ и объект, хранящий информацию. Операционной системе необходимо где-то хранить права доступа субъекта к объекту. Здесь имеется два способа: привязывать права доступа к субъекту, либо привязывать их к объекту. В операционной системе Windows (так же, как и в UNIX) выбран второй способ: привязка прав доступа к объекту. Правами доступа субъектов к объектам управляют владельцы объектов. Системный администратор имеет возможность изменять как права доступа, так и владельца объекта.

Понятие описателя

С точки зрения менеджера безопасности нельзя предоставлять прикладным процессам адрес системного объекта, потому что в этом случае система не сможет контролировать, какой из процессов производит доступ. Поэтому вместо адреса объекта система предоставляет процессам, так называемый, описатель – HANDLE [6].

Описатель HANDLE с точки зрения процесса представляет собой абстрактное число, используемое для идентификации открытого объекта системы. В переводе с английского слово HANDLE означает «ручка». По числовому значению описателя невозможно узнать, какой объект он описывает.

С точки зрения операционной системы описатель представляет собой сложный индекс в системе массивов (рис. 17). Актуальный размер описателя составляет 24 бита. Описатель содержит в себе индекс в массивах верхнего, среднего и низкого уровня как показано на рисунке.

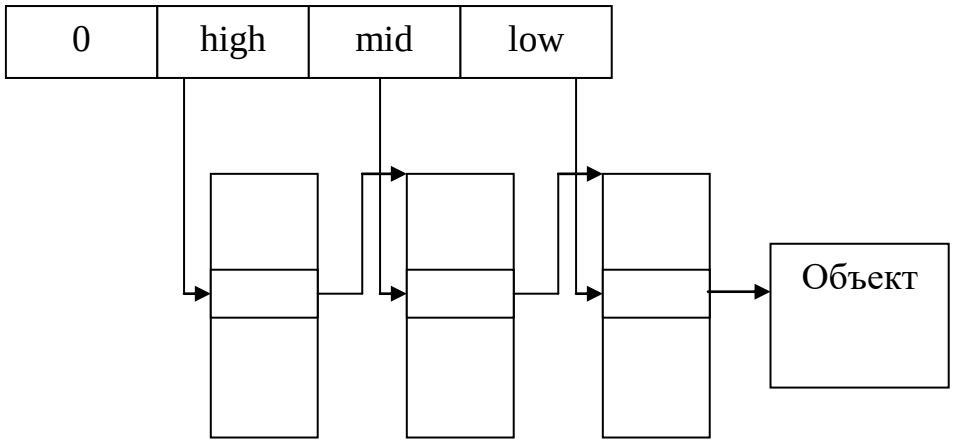

Рис. 17. Принцип работы описателя.

Помимо указателя на объект структура описателя хранит в себе еще и права доступа к объекту, разрешенные по данному описателю. При каждом обращении к объекту права доступа по описателю сравниваются с требуемым правом согласно операции над объектом. Если право отсутствует, то операция отклоняется. Например, если процесс попытается произвести запись в файл, открытый только на чтение, то право на запись будет отсутствовать в информации, связанной с описателем и операция записи будет отклонена менеджером безопасности в самом начале обработки запроса от процесса.

Атрибуты безопасности

Структура SECURITY_ATTRIBUTES содержит в себе описатель безопасности (security descriptor) объекта и определяет, будет ли наследуемым описатель, являющийся результатом работы функции. Эта структура используется во многих функциях создания объектов системы, таких как CreateFile, CreatePipe, CreateProcess, RegCreateKeyEx, RegSaveKeyEx. Во всех этих функциях один из аргументов, это указатель на экземпляр структуры, создаваемый программистом.

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

Члены структуры:

nLength – размер структуры в байтах.

lpSecurityDescriptor – указатель на описатель безопасности. Может быть NULL, тогда объект получит описатель безопасности по умолчанию для объектов текущего процесса.

bInheritHandle – параметр показывает, будет ли наследуемым вновь созданный описатель. При создании нового процесса наследуемые описатели старого процесса унаследуются новым.

Описатель безопасности SecurityDescriptor содержит в себе, помимо прочего, список дискреционного контроля доступа DACL (Discretionary Access Control List). Этот список определяет права других пользователей и групп пользователей на доступ к объекту. Когда программа открывает объект, она запрашивает права на открываемый объект. DACL содержит в себе перечень правил по доступу к объекту других пользователей и групп пользователей. Каждое такое правило называется ACE (Access Control Entry). Правило может быть разрешающим или запрещающим. Оно включает в себя универсальный идентификатор SID и набор прав, которые разрешаются или запрещаются. Просмотр правил происходит в порядке очереди.

Описатель безопасности хранит в себе еще и список системного контроля доступа SACL, в котором содержатся требования по аудиту операций над объектом.

После того, как объект создан, другой субъект может получить к нему доступ. Для этого используются специальные функции получения доступа к объекту. Рассмотрим, например, функцию получения доступа к объекту события OpenEvent.

```
HANDLE OpenEvent(
  DWORD dwDesiredAccess,
  BOOL bInheritHandle,
  LPCTSTR lpName);
```

Первый параметр функции – запрашиваемые права доступа, второй – флаг наследования описателя, а третий – имя события, к которому субъект пытается получить доступ. Здесь наибольший интерес представляет первый параметр: запрашиваемые права доступа. Это 4-байтовое число, информацию о запрашиваемых правах доступа в котором несут отдельные биты этого числа.

Запрашиваемые права доступа может быть унифицированными или точно определенными. Точно определенные виды доступа для каждого вида объекта определены по-разному. Например, для процессов определены такие права:

PROCESS_TERMINATE (0x0001) – право на завершение процесса.

PROCESS_CREATE_THREAD (0x0002) – право на создание нити.

PROCESS_SET_SESSIONID (0x0004) – право на установку идентификатора сессии.

PROCESS_VM_OPERATION (0x0008) – право на управление памятью процесса.

PROCESS_VM_READ (0x0010) – право на чтение из памяти процесса.

PROCESS_VM_WRITE (0x0020) – право на запись в память другого процесса.

PROCESS_DUP_HANDLE (0x0040) – право на дублирование описателей.

PROCESS_CREATE_PROCESS (0x0080) – право на создание нового процесса.

PROCESS_SET_QUOTA (0x0100) – право на установку квот.

PROCESS_SET_INFORMATION (0x0200) – право на установку информации о процессе.

PROCESS_QUERY_INFORMATION (0x0400) – право на запрос информации о процессе.

PROCESS_SUSPEND_RESUME (0x0800) – право на паузу и возобновление работы процесса.

К точно определённым правам также относятся несколько дополнительных прав, которые могут быть применимы к большому количеству типов объектов в системе. Это следующие права:

DELETE (0x00010000L) – право на удаление объекта.

READ_CONTROL (0x00020000L) – право на чтение и управление объектом.

WRITE_DAC (0x00040000L) – право на запись данных безопасности.

WRITE_OWNER (0x00080000L) – право на запись нового владельца.

SYNCHRONIZE (0x00100000L) – право на возможность дождаться объект синхронизации, (если объект является объектом синхронизации).

Унифицированных прав доступа 4: GENERIC_READ, GENERIC_WRITE, GENERIC_EXECUTE и GENERIC_ALL, который обычно объединяет в себе три предыдущих. Унифицированные права доступа помогают упростить получение доступа к объекту, чтобы у программиста не возникало необходимости всякий раз перечислять для каждого объекта требуемые точно определённые права. (Следует отметить, что в операционной системе UNIX схема дискреционного контроля доступа использует только унифицированные права доступа.)

Для трансформации точно определенных прав в унифицированные для каждого типа объектов определена специальная структура: GENERIC_MAPPING.

```
typedef struct _GENERIC_MAPPING {
  ACCESS_MASK GenericRead;
  ACCESS_MASK GenericWrite;
  ACCESS_MASK GenericExecute;
  ACCESS_MASK GenericAll;
} GENERIC_MAPPING;
```

Поле GenericRead содержит в себе набор точно определённых прав, требуемых для операций чтения объекта. Поле GenericWrite содержит в себе набор точно определённых прав, требуемых для операций записи объекта. Поле GenericExecute используется только в точ случае, если объект исполняемый. Поле GenericAll содержит в себе все точно определённые права доступа к объекту.

На основе описателя безопасности, запрашиваемых прав и маркера пользователя операционная система либо запрещает, либо разрешает доступ к объекту. Если по итогам проверки хотя бы одно из запрашиваемых прав доступа не разрешено, то доступ к объекту запрещается.

Маркер пользователя

У каждого пользователя в системе есть структура, описывающая его права доступа. Она называется маркером пользователя или просто маркером [6]. Каждый процесс, запущенный пользователем, функционирует от лица пользователя и связан со своим экземпляром маркера пользователя. Маркер пользователя можно открыть по описателю на процесс с помощью функции OpenProcessToken. Результатом работы этой функции является создание описателя уже на маркер. По этому описателю можно запросить различную информацию с помощью функции GetTokenInformation.

К информации, возвращаемой с помощью функции GetTokenInformation относятся в том числе: имя пользователя, группы, в которые входит пользователь, привилегии пользователя, идентификатор текущей сессии пользователя и др.

Маркер пользователя создается при входе пользователя в систему. Когда пользователь запускает новый процесс, для нового процесса создается новый маркер методом дублирования маркера процесса-родителя.

Каждого пользователя в системе также можно идентифицировать по универсальному идентификатору SID (Security IDentifier). Однако, понятие SID шире, поскольку с помощью этого идентификатора можно указывать также группу пользователя, его псевдонимы, а также, так называемые, широко известные идентификаторы.

Примеры SID приведены ниже:

S-1-0-0 Никто.

S-1-1-0 Все

S-1-2-1 Группа, включающая пользователей, вошедших в физическую консоль.

S-1-5-80-0 Группа, в которую входят все процессы служб, настроенных в системе.

S-1-5 Администратор

S-1-5-2 Группа, в которую входят все пользователи, вошедшие в систему с использованием сетевого подключения.

S-1-5-7 Группа, в которую входят все пользователи, вошедшие в систему анонимно.

S-1-5-18 Ядро локальной операционной системы.

Понятие олицетворения

Все нити процесса функционируют от лица пользователя процесса. Но иногда в серверных системах нужно, чтобы какая-либо нить функционировала от лица другого пользователя и использовала его права. В этом случае применяется механизм, называемый олицетворением или перевоплощением. Для этого используются функции:

ImpersonateLoggedOnUser – олицетвориться в другого пользователя, вошедшего в систему.

ImpersonateAnonymousToken – олицетвориться в анонимного пользователя.

ImpersonateNamedPipeClient – олицетвориться в пользователя, с процессом которого установлена связь через именованный канал.

Для простого уменьшения полномочий нити применяется функция ImpersonateSelf. Когда нити нужно обратно вернуться к правам процесса, она вызывает функцию RevertToSelf.

Олицетворение бывает 4-х видов, описываемых перечислением.

```
typedef enum _SECURITY_IMPERSONATION_LEVEL
{
  SecurityAnonymous,
  SecurityIdentification,
  SecurityImpersonation,
  SecurityDelegation
} SECURITY_IMPERSONATION_LEVEL;
```

Наименьшими правами обладает пользователь, олицетворённый как SecurityAnonymous. В этом случае после олицетворения нельзя получить какую-либо идентификационную информацию пользователя. SecurityIdentification позволяет получать эту информацию, но не позволяет производить вложенное олицетворение. SecurityImpersonation позволяет

производить вложенное олицетворение на локальном компьютере, а SecurityDelegation позволяет производить вложенное олицетворение на удалённом сервере.

Общая схема получения доступа к объекту

На рисунке 18 представлена общая схема получения доступа к объекту на примере вызова функции OpenEvent. Для работы этой функции требуется наличие самого объекта с его описателем безопасности. На основе списка дискреционных прав DACL и текущего маркера пользователя система производит проверку прав на доступ к объекту.

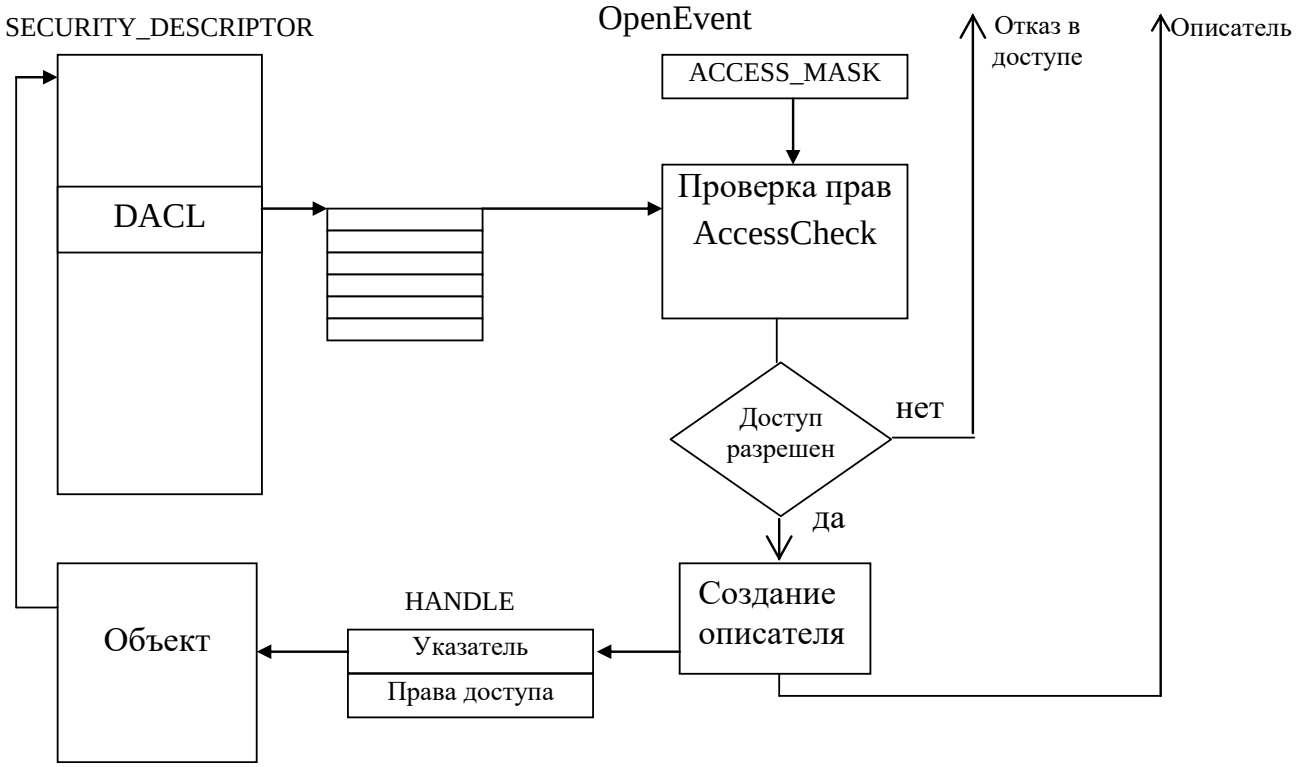

Рис. 18. Общая схема предоставления доступа к объекту при его открытии.

Если после выполнения всех проверок доступ к объекту разрешается, то функция возвращает вновь созданный описатель на объект. В противном случае происходит отказ в доступе.

Управление процессами и нитями

Интерфейс между прикладными процессами и ядром в ОС Windows называется Win32 API. Он состоит из большого количества разнообразных функций. С каждой новой версией ОС добавляются новые функции. Еще одной особенностью Win32 является большое количество параметров функций. В ОС UNIX параметров у системных функций значительно меньше.

Многие функции, описанные в Win32, на самом деле не существуют, поскольку являются макросами, которые компилятор заменяет на нужную функцию. Подобный подход к функциям стал необходим из-за использования символов в формате UNICODE параллельно символам в обычном ASCII формате.

Понятие UNICODE

UNICODE – это форматы алфавитно-цифровых символов. Существуют несколько таких форматов, утвержденных международным институтом стандартизации ISO. К ним, в том числе, относятся UTF-8, используемый в Linux, UTF-16, используемый в Windows и UTF-32 [7]. В формате UTF-16 каждый символ кодируется двумя байтами. Первый байт символа идентифицирует страницу, а второй – сам символ. Так латинские символы расположены на странице с номером 0x00, кириллические символы на странице 0x04. В том случае, если двух байт для кодирования символа не хватает, может быть использовано еще 2 байта. Такое случается в языках с иероглифическим письмом.

Операционная система Windows работает с форматом UTF-16 так же, как и с ASCIIZ строками. В зависимости от того, какой набор символов используется в настройках проекта, компилятор подставляет вместо исходного макроса вызов правильной функции ядра. Названия таких функций отличаются от названия исходного макроса подстановкой символа «A» или «W» в качестве суффикса в название функции. Так функции CreateProcess на самом деле не существует, а вместо нее существуют две функции CreateProcessA и CreateProcessW. Это происходит в том случае, если на вход функции поступает один или несколько строковых параметров. В этом случае они задаются типами LPSTR или LPWSTR. Возможно еще добавление буквы C, означающей константную строку, например, LPCSTR.

Функции, которые принимают на вход ASCIIZ символы, являются обычными ретрансляторами, которые преобразуют не UNICODE строки в UNICODE. Ядро операционной системы Windows работает с UNICODE строками.

Какую конкретно функцию нужно использовать задается директивой предкомпиляции UNICODE. Можно в основном заголовочном файле проекта сделать запись #define UNICODE или определить эту директиву в настройках проекта.

Для задания строк в формате UNICODE используется специальный префикс L.

Например, текст “Hello, world!” будет воспринят компилятором, как строка ASCII символов, а текст L”Hello, world!” как строка в формате UNICODE.

Чтобы упростить переход между компиляцией UNICODE и не UNICODE программами рекомендуется не использовать префикс L, а использовать макрос _T. Этот макрос разворачивается либо как L, либо как пустота в зависимости от наличия директивы UNICODE.

Создание и завершение процессов и нитей

Процессы в ОС Windows делятся на оконные и консольные. Возможен вариант, когда одно приложение является одновременно и оконным, и консольным. В зависимости от типа процесса и использования UNICODE основная функция программы называется по-разному:

Таблица 1.

Названия основных функций программы в зависимости от вида процесса.

	Оконный процесс	Консольный процесс
ASCII	WinMain	main
UNICODE	wWinMain	wmain

Для создания процесса используется функция CreateProcess, которая создает новый объект процесса, новое адресное пространство и одну нить процесса, которая начинает выполнение с точки входа процесса, указанной в исполняемом файле. После выполнения предварительных действий (инициализация переменных, вызов конструкторов) нить приходит в стандартную функцию MainCrtStartup, из которой вызывается соответствующая типу процесса основная функция.

Для создания новой нити используется функция CreateThread.

Нить завершает свое выполнение в следующих случаях:

- По выходу из основной функции нити;
- По вызову из этой нити функции ExitThread;
- По вызову из какой-либо нити функции TerminateThread.

Рекомендованный способ и хороший стиль завершения нити – выход из основной функции нити. В этом случае правильно отработают различные вспомогательные функции библиотек, по очистке различных структур.

Процесс завершает свою работу в следующих случаях:

- Если главная нить процесса выйдет из функции main;
- Если одна из нитей процесса вызовет функцию ExitProcess;
- Если одна из нитей другого процесса вызовет функцию TerminateProcess;
- Если все нити процесса завершат свою работу.

Рекомендованный способ завершения процесса следующий: главная нить процесса должна дождаться завершения всех остальных нитей процесса, а затем выйти из основной функции процесса. Только в этом случае правильно отработают деструкторы глобальных переменных программы (в случае использования языка C++).

По завершении процесса операционная система автоматически закрывает все объекты ядра, связанные с процессом. Однако хорошим стилем считается самостоятельное закрытие процессом описателей на объекты ядра.

Запуск нитей процесса можно приостановить, с помощью флага CREATE_SUSPENDED. Этому флагу соответствует функция SuspendThread для уже запущенной нити. С помощью функции ResumeThread работа нити может быть возобновлена. Число «замораживаний» нити накапливается. Это означает, что сколько раз была вызвана функция SuspendThread, столько же раз должна быть вызвана функция ResumeThread.

Перечислим некоторые другие стандартные функции работы с процессами:

GetCommandLine – функция возвращает указатель на командную строку, причем это единственный способ получить командную строку в формате UNICODE.

GetCurrentProcess – функция возвращает псевдоописатель процесса.

GetCurrentProcessId – функция возвращает идентификатор процесса – PID.

GetCurrentThreadId – функция возвращает идентификатор текущей нити – TID.

OpenProcess – функция получает описатель на другой процесс по его идентификатору PID.

OpenThread – функция получает описатель на другую нить, в том числе и нить другого процесса по её идентификатору TID.

Структурная обработка исключений

Структурная обработка исключений (английский термин Structured Exceptions Handling, SEH), это механизм, позволяющий при возникновении ошибки в программе, обработать эту ошибку на этапе исполнения и продолжить выполнение программы. Язык программирования C++ позволяет проводить такую обработку с помощью блоков try, catch. Это весьма гибкий механизм обработки исключительных ситуаций. Его описание можно найти в любой книге по C++.

Здесь мы рассмотрим основные принципы этого механизма, на примере языка C и компилятора этого языка от компании Microsoft. Компания Microsoft заложила в свой компилятор специальный механизм обработки завершения и исключений. Этот механизм входит в состав расширений языка C, которые носят общее название Microsoft-Specific [2].

В состав механизма структурной обработки исключений входят пары блоков __try/_finally и __try/__except. Любое количество блоков __try может быть вложено один в другой. Однако каждый должен завершаться либо блоком __finally, либо блоком __except. Ниже показан пример пары блоков __try/__finally.

```
__try
{
      // защищенный блок
      if (bNeedToOut)
          __leave;
      // другие действия.
}
__finally
{
    // обработчик завершения
    if (AbnormalTermination()) {
        // выход по ошибке
    } else {
        // выход по завершению блока __try
    }
}
```

Блок __finally позволяет элегантно решить проблему «подчистки мусора» – освобождения памяти, закрытия описателей и т. д. Он выполняется всегда, даже если из блока __try был вызван оператор возврата return. Макрос AbnormalTermination() возвращает значение FALSE только в том случае, когда блок __try отработал полностью. Если же в нём был вызван, например return, то макрос AbnormalTermination() вернёт TRUE. Если из блока __try выходить через ключевое слово __leave, то такое завершение считается нормальным, и AbnormalTermination() вернёт FALSE.

Необходимо ещё раз отметить, что *блок __finally выполняется всегда*, даже если в программе произошло исключение. После обработки фильтром исключения, которые рассматриваются дальше, будут обязательно вызваны все блоки __finally расположенные в стеке исключений.

Блок __try, __except призван обработать возникающие ошибки, и собственно, и является элементом структурной обработки исключений. Выглядит следующим образом:

```
__try
{
// защищенный блок
 }
__except(<фильтр исключений>)
{
// обработчик исключений
}
```

Фильтр исключений представляет собой одно значение. Это значение можно установить явно, а можно, например, вернуть как результат работы функции.

Таблица 2.

Виды возвращаемых значений фильтров исключений

Идентификатор	Значение
EXCEPTION_EXECUTE_HANDLER	1
EXCEPTION_CONTINUE_SEARCH	0
EXCEPTION_CONTINUE_EXECUTION	-1

EXCEPTION_EXECUTE_HANDLER – Ошибка распознана. Последствия ошибки будут ликвидированы в блоке __except. Затем программа перейдет на следующий оператор за блоком __except.

EXCEPTION_CONTINUE_EXECUTION – Ошибка распознана и устранена. Необходимо вернуться на ту же самую машинную команду, которая вызвала ошибку.

EXCEPTION_CONTINUE_SEARCH – Ошибка не распознана. Необходимо перейти на следующий вверх блок _try, __except или, если его нет вызвать стандартный обработчик исключений ОС, который аварийно завершит программу.

Рассмотрим, как выглядит реализация структурной обработки исключений для 32-разрядных программ, работающих под управлением операционной системы Windows.

Стек обработчиков исключений индивидуальный для каждой нити в составе процесса. Адрес текущего обработчика исключений нити всегда находится по адресу fs:0. Дело в том, что сегментный регистр fs в операционной системе Windows всегда указывает указывает на специальную структуру, которая называется Thread Environment Block, (сокращенно, TEB). Первое поле этой структуры и есть адрес текущего обработчика исключений. При входе в каждый следующий блок __try текущий обработчик исключения копируется в стек, а на его место по адресу fs:0 копируется новый обработчик исключения. При выходе из функции происходит обратная операция. Данная схема применяется только в 32-битных приложениях Windows, в 64-битных приложениях используется другая, более сложная, но и более защищённая схема, рассмотрение которой выходит за рамки этого курса.

При работе с исключениями посмотреть параметры текущего исключения можно внутри фильтра исключений с помощью функции: GetExceptionInformation.

```
LPEXCEPTION_POINTERS GetExceptionInformation(void);
```

Функция возвращает адрес структуры: EXCEPTION_POINTERS, которая содержит в себе расширенную информацию об исключении в виде двух указателей на другие структуры:

```
typedef struct _EXCEPTION_POINTERS {
  PEXCEPTION_RECORD ExceptionRecord;
  PCONTEXT          ContextRecord;
} EXCEPTION_POINTERS, *PEXCEPTION_POINTERS;
```

Структура EXCEPTION_RECORD предоставляет информацию об условиях возникновения исключения, а структура CONTEXT различается в зависимости от вида аппаратной платформы, поскольку содержит в себе параметры контекста нити в виде регистров аппаратного процессора. В частности, поле ExceptionAddress содержит в себе адрес возникновения исключения в адресном пространстве текущего процесса, а поле ExceptionCode указывает на причину возникновения исключения. Необходимо отметить две наиболее частые причины возникновения исключения EXCEPTION_ACCESS_VIOLATION и EXCEPTION_IN_PAGE_ERROR, которые возникают при ошибках работы с памятью процесса, в частности, при обращении по нулевому указателю.

```
typedef struct _EXCEPTION_RECORD {
  DWORD                    ExceptionCode;
  DWORD                    ExceptionFlags;
```

```
  struct _EXCEPTION_RECORD *ExceptionRecord;
  PVOID                    ExceptionAddress;
  DWORD                    NumberParameters;
  ULONG_PTR
ExceptionInformation[EXCEPTION_MAXIMUM_PARAMETERS];
} EXCEPTION_RECORD;
```

Исключение можно вызвать намеренно с заданными параметрами. Это делается с помощью функции RaiseException. При наиболее простом использовании этой функции аргументы и флаги исключения можно не указывать, достаточно указать код исключения.

```
void WINAPI RaiseException(
  _In_       DWORD     dwExceptionCode,
  _In_       DWORD     dwExceptionFlags,
  _In_       DWORD     nNumberOfArguments,
  _In_ const ULONG_PTR *lpArguments);
```

Приведем пример кода, выполняющего обработку исключения при попытке выполнения операции деления на ноль. В этом случае будем присваивать результату деления также значение ноль.

```
int a = 10, b = 0;
__try
{
    a = a/b;
 }
__except(EXCEPTION_EXECUTE_HANDLER)
{
    a = 0;
}
```

Приведём пример последовательности выполнения операторов языка С при вложенной структурной обработке исключений. В круглых скобках даны числа, указывающие на последовательность выполнения операторов.

```
__try {
      __try{
            __try {
                  a=a/0; (1)
            }
            __except(0 (2)) {}
      __finally { (4) }
__except(filter() (3), 1)
{ (5) }
(6)
```

Работа с файлами

Основные понятия

*Под **файлом** обычно понимают обычно понимают набор данных, организованных в виде совокупности записей одинаковой структуры.* Для управления этими данными создаются соответствующие **системы управления файлами**. Возможность иметь дело с логическим уровнем структуры данных и операций, выполняемых в процессе обработки данных, предоставляет система управления файлами [1]. Логическое представление файлов также оперирует понятием **каталог**, которое представляет собой именованное объединение файлов в группы с возможностью организации иерархической структуры каталогов.

Система управления файлами активно взаимодействует с подсистемой безопасности в составе операционной системы в части контроля доступа к файлам и каталогам. Система управления файлами также взаимодействует с файловыми системами. ***Файловая система** представляет собой набор спецификаций по хранению, организации и контролю доступа, и обработке данных.* Кроме того, файловой системой по традиции называют также драйвер, реализующий данный набор спецификаций.

Необходимо различать понятия «система управления файлами» и «файловая система», поскольку система управления файлами находится в составе операционной системы в единственном экземпляре, а файловых систем может быть много. Все современные операционные системы имеют в своём составе систему управления файлами. Система управления файлами предоставляет возможности:

- создание, удаление, переименование файлов и др. функции с использованием файлового API;
- работа с не дисковыми периферийными устройствами как с файлами;
- взаимный обмен данными между файлами и устройствами;
- защита файлов от несанкционированного доступа.

Система управления файлами часто очень сильно взаимосвязана с конкретной файловой системой и, поэтому, зачастую рассматривается в контексте той или иной файловой системы.

На рисунке 19 представлена последовательность действий программы и ядра операционной системы при работе с файлами.

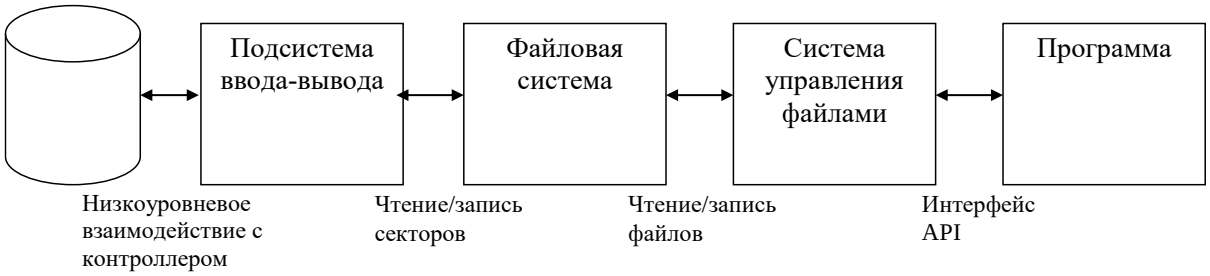

Рис. 19. Последовательность действий при доступе к файлу.

На представленном рисунке для каждого уровня иерархии выполняется действие с соответствующим уровню логическим представлением данных.

Система управления файлами в операционных системах UNIX

Этот раздел один из немногих, описывающих внутреннее устройство ядра операционной системы UNIX, поскольку система управления файлами в операционной системе UNIX является основой построения всей операционной системы. Верно, в том числе, утверждение, что в операционной системе UNIX всё есть файл [5, 7].

В операционной системе UNIX имеется 6 типов файлов: обычный файл, каталог, специальный файл устройства, FIFO, связь (link), сокет [8, 9]. В среде программирования UNIX существует два основных интерфейса для файлового ввода/вывода:

1. интерфейс системных вызовов, включающий системные функции низкого уровня, непосредственно взаимодействующие с ядром операционной системы;
2. стандартная библиотека ввода/вывода включающая функции буферизированного ввода/вывода Функции начинаются на букву «f».

Второй интерфейс является надстройкой над интерфейсом системных вызовов.

Ниже в таблице 3 перечислены функции обоих интерфейсов и дана их краткая аннотация. Функции буферизированного ввода-вывода помечены символом «б».

Таблица 3.

Функции операционной системы UNIX по работе с файлами

Флаг	Имя функции	Описание

б	open fopen	Функция открывает файл и возвращает дескриптор файла (файловый указатель). Если файл не существует, он может быть создан.
	creat	Создает файл
б	close fclose	Закрывает файловый дескриптор (поток)
	dup dup2	Создает дубликат файлового дескриптора.
б	lseek fseek	Устанавливает файловый указатель на определенное место в файле.
б б б б	read readv fread getc gets scanf	Считывает данные из файла.
б б б б б	write writev fwrite putc puts printf	Записывает данные в файл.
	pipe	Создает конвейер.
	fcntl	Обеспечивает интерфейс управления открытым файлом.
б	fileno	Возвращает дескриптор по файловому указателю.
	link	Создает жесткую связь
	unlink	Удаляет жесткую связь
	symlink	Создает символическую ссылку
	readlink	Чтение содержимого символической ссылки
	mmap	Отображение файла в память
	chown fchown	Изменение владения файлом
	lchown	Изменение владения символической ссылкой
	chmod fchmod	Устанавливает права доступа к файлу
	chroot fchroot	Изменить корневой каталог для текущего процесса
	chdir fchdir	Изменить текущий каталог
	stat	Получить метаданные файла (атрибуты и т. д.)

	lstat fstat	

Рассмотрим подробнее метаданные файла. Каждый файл помимо собственно данных содержит метаданные, описывающие его характеристики [5]. Все характеристики файла хранятся в специальной структуре, номер экземпляра которой уникален в пределах заданного экземпляра жесткого диска. Экземпляр этой структуры на диске называется inode (индексный дескриптор). Некоторые параметры этой структуры могут быть получены с помощью функции stat

```
int stat(const char* filename, struct stat *buf);

struct stat {
    dev_t         st_dev;      /* device */
    ino_t         st_ino;      /* inode */
    mode_t        st_mode;     /* protection */
    nlink_t       st_nlink;    /* number of hard links */
    uid_t         st_uid;      /* user ID of owner */
    gid_t         st_gid;      /* group ID of owner */
    dev_t         st_rdev;     /* device type (if inode device)
*/
    off_t         st_size;     /* total size, in bytes */
    blksize_t     st_blksize;  /* blocksize for filesystem I/O
*/
    blkcnt_t      st_blocks;   /* number of blocks allocated */
    time_t        st_atime;    /* time of last access */
    time_t        st_mtime;    /* time of last modification */
    time_t        st_ctime;    /* time of last change */
};
```

Как видно из описания структуры stat, она содержит в себе, в том числе, отдельно время изменения файла и время изменения информации о файле, размер файла и другие параметры.

Система управления файлами в ОС UNIX называется виртуальной файловой системой. Файловые системы различных типов встраиваются в виртуальную файловую систему, которая обеспечивает унифицированный интерфейс. Интерфейс работы с файлами называется vnode. (виртуальный индексный дескриптор). Структура данных vnode одинакова для файлов любых файловых систем.

Прежде чем может состояться работа с файлами, соответствующая файловая система должна быть встроена в дерево каталогов. Эта операция называется монтированием файловой системы. Каждая подключенная файловая система представляется в виде экземпляра структуры vfs. Все экземпляры этой структуры организованы в виде односвязного списка,

первым экземпляром которого является корневая файловая система. Весь этот список имеет общее название – таблица монтирования.

Монтирование файловой системы осуществляется системным вызовом mount, а также утилитой mount. Утилита mount с ключом –l даст список всех смонтированных файловых систем.

Лучший способ работы с файловыми системами в операционной системе Linux – прописать их в специальный файл /etc/fstab. Записи в этом файле можно настроить таким образом, чтобы монтирование файловой системы происходило автоматически при старте операционной системы. Если точка монтирования описана в файле fstab, то для ее монтирования достаточно запустить утилиту mount и указать каталог монтирования файловой системы. Все остальные параметры будут взяты из файла fstab. В противном случае придется все параметры файловой системы перечислять в командной строке.

Для размонтирования файловой системы используется утилита umount и функция umount. Чтобы размонтировать файловую систему, необходимо в качестве параметра указать каталог монтирования.

Структура файла fstab следующая. Каждая строка в этом файле описывает одну точку монтирования файловой системы (каталог). В строке содержится 6 полей, разделенных табуляцией или пробелом.

Первое поле (fs_spec) указывает на файл монтируемого устройства, например, “/dev/cdrom”.

Второе поле (fs_file) описывает точку монтирования, например, “/cdrom”.

Третье поле (fs_vfstype) описывает тип файловой системы, например, “iso9660”. Если вместо типа указать ключевое слово “auto”, то тип будет определен автоматически.

Четвертое поле (fs_mntops) описывает опции, ассоциированные с файловой системой, разделенные запятыми. Опция “noauto” заставляет не монтировать эту файловую систему в течение загрузки ОС. Опция “user” разрешает монтирование всем пользователям ОС, а не только суперпользователю. Есть и другие опции.

Пятое поле (fs_freq) управляет дампом файловой системы во время краха и может отсутствовать или быть равным нулю.

Шестое поле (fs_passno) определяет порядок проверки целостности файловых систем при перезагрузке. Может отсутствовать или быть равным нулю и это означает, что эту файловую систему проверять не нужно.

Система управления файлами в операционной системе Windows

Работа с файлами в Windows организована иначе, чем в UNIX. Основная функция работы с файлами здесь – CreateFile [2]. Эта функция открывает и создает файл. Однако этим ее возможности не исчерпываются. Эта функция открывает:

3. Файлы
4. Консольные потоки (CONIN$ и CONOUT$)
5. Коммуникационные ресурсы (порты компьютера (COM, LPT))
6. Каталоги
7. Дисковые устройства
8. Драйверы
9. Майлслоты
10. Каналы

Все вышеперечисленные объекты системы закрываются стандартной функцией CloseHandle. Таким образом, функция CreateFile является универсальной функцией работы с устройствами компьютера для хранения и вывода информации. Однако, после получения дескриптора на объект, дальнейшая работа с объектом ведется с помощью различных функций, зависящих от типа объекта. В этом разница от операционной системы UNIX, в которой любой объект системы есть файл. В операционной системе Windows файл – это лишь один из многих типов объектов.

Функция CreateFile задает, будет ли производится буферизация при работе с файлом, а также дисциплину буферизации: для последовательного или случайного доступа внутри файла. Таким образом буферизация делается ядром ОС, а не прикладной библиотекой, как в UNIX. После получения дескриптора дальнейшая работа с объектом идет по одному из двух направлений.

1. Файлы, консольные потоки, коммуникационные ресурсы, майлслоты, каналы
2. Каталоги, дисковые устройства, драйверы

Первое направление использует для обмена данными функции ReadFile, ReadFileEx для чтения и WriteFile, WriteFileEx для записи. Второе направление использует функцию DeviceIoControl. Кроме того, имеются функции, специфичные только для конкретного объекта.

Основное отличие функций ReadFile и WriteFile от функций read и write операционной системы UNIX заключается в том, что в Windows возможно и синхронное и асинхронное выполнение файловых операций.

Рассмотрим сказанное на примере функции ReadFile, предназначенной для чтения файла.

```
BOOL ReadFile(
  HANDLE       hFile,
  LPVOID       lpBuffer,
  DWORD        nNumberOfBytesToRead,
  LPDWORD      lpNumberOfBytesRead,
  LPOVERLAPPED lpOverlapped
);
```

Первый аргумент функции, это описатель на предварительно открытый файл с помощью функции CreateFile. Второй аргумент, – это указатель на предварительно выделенный буфер, размер которого задаётся третьим аргументом. Четвёртый аргумент есть адрес переменной, в которую запишется количество прочитанных байт при успешном завершении синхронной операции чтения файла. Последний, пятый аргумент функции, необходимо рассмотреть подробнее, поскольку именно с его помощью выполняется асинхронное чтение файла. При синхронном чтении пятый параметр может указывать на NULL.

Для организации асинхронного чтения файла необходимо, чтобы описатель на файл был открыт с флагом FILE_FLAG_OVERLAPPED, переданным в функцию CreateFile. Затем, при вызове функции ReadFile необходимо передать в качестве пятого аргумента функции адрес экземпляра структуры OVERLAPPED. В этой структуре необходимо заполнить поле hEvent описателем на событие, созданным с помощью функции CreateEvent. Поля Internal и InternalHigh необходимо обнулить, а поля Offset и OffsetHigh представляют собой смещение в файле, относительно которого необходимо выполнить операцию чтения.

```
typedef struct _OVERLAPPED {
  ULONG_PTR Internal;
  ULONG_PTR InternalHigh;
  union {
    struct {
      DWORD Offset;
      DWORD OffsetHigh;
    } DUMMYSTRUCTNAME;
    PVOID Pointer;
  } DUMMYUNIONNAME;
  HANDLE    hEvent;
} OVERLAPPED, *LPOVERLAPPED;
```

Физический смысл асинхронной операции следующий: возврат из функции происходит сразу же после начала операции (в данном случае –

чтения), а не после ее окончания. Событие hEvent при этом переходит в сброшенное состояние. Программа может продолжать свое выполнение, но при этом параллельно с ней будет выполняться операция работы с файлом. Как только работа с файлом закончится, событие hEvent перейдёт в сигнальное состояние. Нить программы может периодически в цикле опрашивать состояние этого события, при этом выполняя какие-либо другие операции. Такой приём позволяет повысить быстродействие системы в целом.

Как до, так и после перехода события в сигнальное состояние, программа может вызвать функцию GetOverlappedResult и получить результат выполнения операции чтения файла.

```
BOOL GetOverlappedResult(
  HANDLE       hFile,
  LPOVERLAPPED lpOverlapped,
  LPDWORD      lpNumberOfBytesTransferred,
  BOOL         bWait);
```

Функция GetOverlappedResult позволяет узнать число прочитанных байт. Её последний аргумент – флаг bWait в значении TRUE заставляет переключить операцию в синхронный режим и дождаться её завершения в том случае, если на текущий момент времени операция чтения ещё не была завершена.

Еще один вариант запуска асинхронной операции чтения файла – функция ReadFileEx. Она отличается от функции ReadFile наличием дополнительного аргумента lpCompletionRoutine, который указывает на специальную функцию, созданную самим программистом.

```
BOOL ReadFileEx(
  HANDLE                          hFile,
  LPVOID                          lpBuffer,
  DWORD                           nNumberOfBytesToRead,
  LPOVERLAPPED                    lpOverlapped,
  LPOVERLAPPED_COMPLETION_ROUTINE lpCompletionRoutine);
```

У функции ReadFileEx параметр hEvent структуры OVERLAPPED игнорируется. Программист может использовать его по своему усмотрению. По окончании файловой операции вызывается функция из параметра lpCompletionRoutine.

```
VOID CALLBACK FileIOCompletionRoutine(
  _In_    DWORD        dwErrorCode,
  _In_    DWORD        dwNumberOfBytesTransfered,
  _Inout_ LPOVERLAPPED lpOverlapped);
```

Эта функция является APC вызовом. С понятием APC мы уже сталкивались при рассмотрении функции SetWaitableTimer. APC вызов производится самой операционной системой, при выполнении определённых условий. Перечень условий:

1. Должна завершиться асинхронная операция (для функции ReadFileEx это завершение операции чтения файла, а для функции SetWaitableTimer это истечение времени таймера);
2. Нить должна находиться в специальном **тревожном** (alertable) состоянии.

Функция APC в операционной системе Windows будет вызываться только в том случае, если нить находится тревожном состоянии. Нить находится в этом состоянии только во время выполнения некоторых функций с расширением Ex. Это такие функции, как SleepEx, WaitForSingleObjectEx, WaitForMultipleObjectsEx и др.

Необходимо отметить, что асинхронный режим работы с файлами для системы управления файлами является базовым, поскольку асинхронный режим работы можно всегда перевести в синхронный, дополнив операцию командой ожидания. Синхронный режим работы перевести в асинхронный невозможно.

Перечислим кратко некоторые другие функции работы с файлами в операционной системе Windows.

Таблица 4

Перечень функций операционной системы Windows по работе с файлами

CopyFile CopyFileEx	Копирование файлов
MoveFile MoveFileEx	Переименование/перемещение файлов
GetDriveType	Получение типа носителя информации
GetFile Size GetFileSizeEx	Получение размера файла (в 8-ми байтовом виде)
GetFullPathName	Получение полного пути до файла по имени файла
SearchPath	Поиск пути для исполняемого файла
GetFileAttributes GetFileAttributesEx	Получение атрибутов файла
FindFirstFile FindNextFile FindClose	Поиск файла в каталоге

FindFirstChangeNotification FindNextChangeNotification FindCloseChangeNotification	Мониторинг изменения атрибутов файлов в заданном каталоге (в том числе запись в файл и изменение его размера)
GetCurrentDirectory	Получение текущего каталога
SetCurrentDirectory	Установка текущего каталога
DeleteFile	Удаление файла
RemoveDirectory	Удаление каталога

Подсистема ввода-вывода

Основные понятия

Компонент подсистемы ввода-вывода в составе операционной системы отвечает за взаимодействие с аппаратными устройствами компьютера. Подсистема ввода-вывода должна решать ряд задач, наиболее важные из которых приведены ниже:

1. организация параллельной работы устройств ввода-вывода и процессора;
2. согласование скоростей обмена и кэширование данных;
3. разделение устройств и данных между процессами;
4. обеспечение интерфейса между устройствами и остальной частью операционной системы;
5. поддержка широкого спектра драйверов;
6. динамическая загрузка и выгрузка драйверов;
7. поддержка синхронных и асинхронных операций ввода-вывода.

Исходя из представленного перечня можно определить, что подсистема ввода вывода выполняет большое количество разнообразных операций по управлению разнотипными устройствами ввода-вывода в составе компьютера. Именно поэтому подсистема ввода-вывода не является монолитным компонентом операционной системы, а включает в себя дополнительные части, называемые драйверами устройств, или просто драйверами.

*Программа, управляющая конкретной моделью устройства, называется **драйвером** устройства.*

Каждое устройство ввода-вывода вычислительной системы снабжено специализированным блоком управления, называемым **контроллером**.

Контроллер взаимодействует с драйвером устройства. После каждого обмена информацией с драйвером контроллер может управлять устройством автономно, тем самым, разгружая центральный процессор. Работой драйверов управляет подсистема ввода вывода и диспетчер задач операционной системы.

Устройства ввода-вывода могут предоставляться процессам как в монопольное, так и в совместное использование. В последнем случае необходимо разделять управляющую информацию и данные разных процессов друг от друга. Для этого подсистема ввода вывода создаёт логическую структуру, которая называется **заданием ввода-вывода**. Задания ввода-вывода объединяются в списки и обрабатываются драйверами по очереди. В операционной системе Windows задания ввода-вывода называются Io Request Packet или, сокращённо, IRP [6].

Каждое задание ввода-вывода в очереди содержит в себе в том числе PID процесса, создавшего задание, адрес данных, которые необходимо обработать и код операции обработки.

В качестве универсального интерфейса взаимодействия с драйверами операционные системы семейства UNIX используют файловую модель периферийных устройств [5]. Это позволяет использовать унифицированные системные вызовы (read и write) для различных устройств. Такой интерфейс называется базовым. Помимо базового интерфейса подсистема ввода-вывода предоставляет возможность разработки специфического интерфейса, например, для вывода графической информации на экран и т. д.

В операционной системе Windows интерфейс взаимодействия с устройствами в составе компьютера был разработан заново и учитывает различные особенности управления устройствами, в частности асинхронную модель передачи данных.

В простейшем случае подсистема ввода-вывода предоставляет два интерфейса для драйверов: один – для взаимодействия с ядром операционной системы (Driver Kernel Interface, DKI) и второй – для взаимодействия с контроллерами периферийных устройств (Driver Device Interface, DDI), как показано на рисунке 20. Примером DDI интерфейса может служить интерфейс стандарта NDIS (Network Driver Interface Specification) для драйверов сетевых адаптеров.

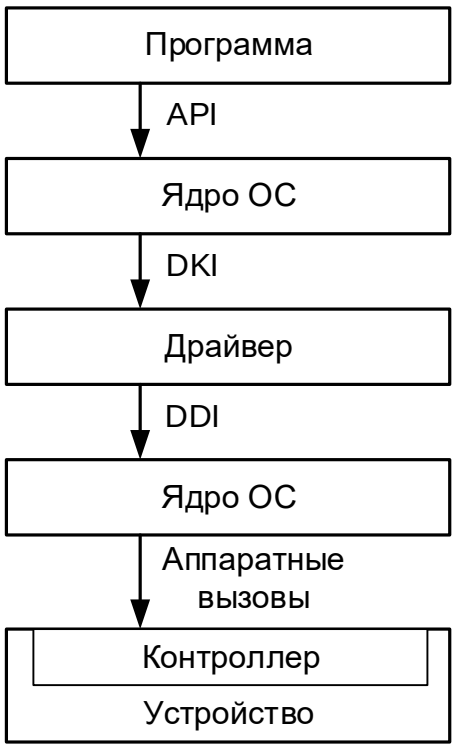

Рис. 20. Последовательность доступа к периферийному устройству.

В первых операционных системах драйверы компилировались вместе с ядром системы, что упрощает структуру операционной системы, но требует высокой квалификации пользователя и необходимости перезагрузки компьютера при старте нового драйвера. Современные операционные системы дают возможность автоматического нахождения, установки и выгрузки требуемых драйверов.

Такая возможность впервые появилась в компьютерах вместе с внедрением аппаратной шины PCI. Устройства на этой шине имеют возможность уведомлять операционную систему о себе, отправляя специальные идентификаторы Hardware Identifier (HID) и Compatible Identifier (CID), содержащие код производителя аппаратуры и код самого устройства. В ответ операционная система выполняет поиск наиболее подходящего драйвера для обслуживания устройства и производит установку и инициализацию драйвера. Принцип подобного взаимодействия операционной системы, устройства и драйвера называется Plug-And-Play (сокращённо PNP). Наибольшее развитие он получил в реализации универсальной последовательной шины USB.

Операции ввода-вывода могут выполняться в синхронном и асинхронном режимах. Синхронный режим означает, что программный модуль приостанавливает свою работу до тех пор, пока операция ввода-вывода не будет завершена. В асинхронном режиме программный модуль инициирует операцию ввода-вывода и продолжает выполняться параллельно с операцией ввода-вывода. Асинхронный режим работы является базовым, поскольку на его основе можно построить синхронный, создав процедуру ожидания ввода-вывода.

Многослойная модель подсистемы ввода-вывода

Многослойная модель подсистемы ввода-вывода (рис. 21) является развитием стандартной модели взаимодействия операционной системы и драйверов устройств. Она используется в современных операционных системах и предоставляет драйверам возможность многократной последовательной обработки одного задания ввода-вывода в виде, так называемых, слоёв. Количество возможных слоёв обработки информации довольно велико, так в операционной системе Windows оно может достигать 255. Традиционные драйверы, обеспечивающие работу с внешними устройствами, стали называться аппаратными или низкоуровневыми драйверами, в противовес высокоуровневым драйверам, обеспечивающим более высокий уровень абстракции при обработке информации.

В качестве примера необходимости многослойной модели можно привести обработку информации подсистемой ввода-вывода при

выполнении записи данных на диск. Для выполнения этой операции в состав подсистемы ввода-вывода должны входить, как минимум, следующие драйверы: драйвер управления контроллером жесткого диска, драйвер контроллера интерфейса SATA, драйвер монтирования разделов накопителя и драйвер файловой системы (на практике количество слоёв обработки для такой операции значительно больше).

При использовании многослойной модели подсистемы ввода-вывода дополнительно повышается гибкость и расширяемость функций по управлению устройством. Так драйвер файловой системы может обрабатывать запросы для различных видов накопителей, например, подключенных по интерфейсам SATA и USB.

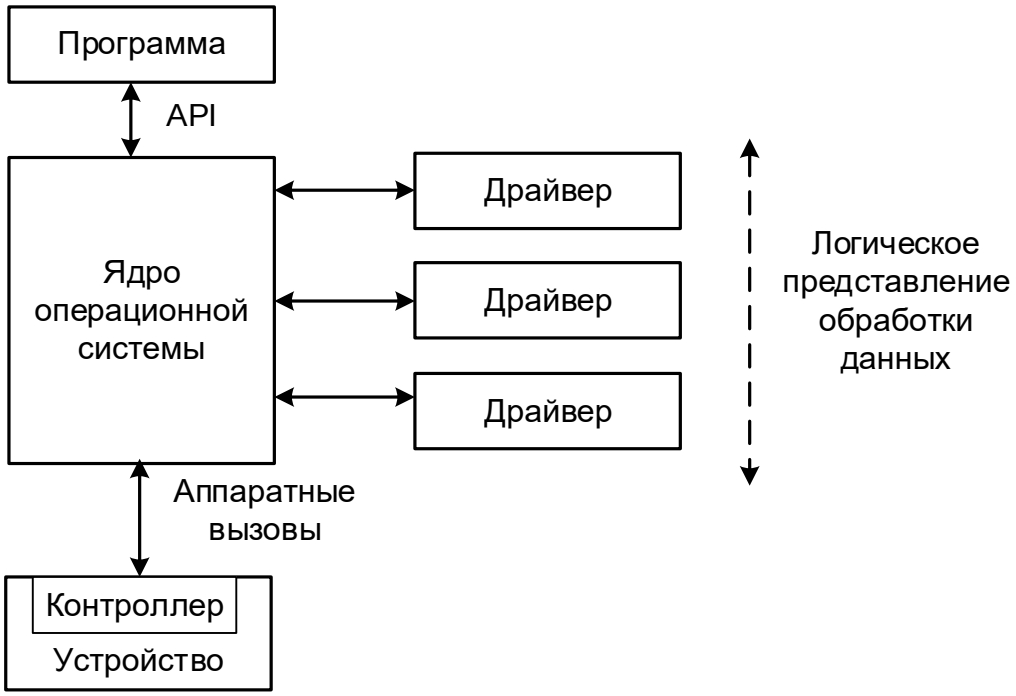

Рис. 21. Многослойная структура подсистемы ввода-вывода.

Драйверы файловых систем

Драйверы файловых систем являются яркими представителями высокоуровневых драйверов, обеспечивающих логическое представление информации на накопителях в виде файлов и каталогов. Как уже упоминалось, драйверы файловых систем тесно взаимодействуют с системой управления файлами для организации и управления доступом к файлам [4].

Все файловые системы можно условно разделить на две группы: файловые системы для накопителей с последовательным доступом к данным

(например, оптические накопители) и файловые системы для накопителей с произвольным доступом к данным (накопители на жёстких дисках и флеш-накопители). Для устройств накопителей с последовательным доступом к данным наибольшее распространение получили файловые системы CDFS и UDF, для накопителей с произвольным доступом к данным наибольшее распространение получили файловые системы FAT, NTFS и Ext. Далее рассмотрим некоторые из них.

Файловая система FAT

Эта файловая система предусматривает наличие на диске специальных областей, выделенных для организации пространства раздела в процессе его форматирования – запись загрузки, таблицу размещения файлов и корневой каталог. Своё название файловая система FAT получила именно по наличию в ней таблицы размещения файлов – File Allocation Table.

На физическом уровне пространство диска разбивается на 512-байтовые области, называемые секторами. Но для оперирования данными на уровне операционной системы такая размерность очень мала, поэтому в системе FAT место для файлов выделяется блоками, которые состоят из целого числа секторов и именуются кластерами.

Число секторов в кластере должно быть кратно степени двойки. Обычно размер кластера можно определить, поделив объем памяти диска на 64 Кбайт (65 536 байт) и округлив результат до ближайшего большего числа, кратного степени двойки. Так, размер кластеров 1,2-Гбайт диска составляет 32 Кбайт – если 1,2 Гбайт (1 258 291,2 Кбайт) поделить на 65 536, получим 19,2 Кбайт, а после округления – 32 Кбайт.

Структура типичного **логического тома** выглядит следующим образом (рис. 22). Рассмотрим последовательно компоненты этой структуры.

Рис. 22. Структура логического тома файловой системы FAT.

Запись загрузки (сектор загрузчика), расположенная в логическом секторе 1 содержит всю важную информацию о характеристиках диска. Структура этого сектора, соответствующая гибкому магнитному диску или логическому тому жесткого диска, выглядит, как показано на следующем рисунке.

Блок параметров BIOS (BPB) описывает физические характеристики диска и позволяет драйверу устройства вычислять правильные физические адреса.

File Allocation Table (FAT). При создании или расширении файла ОС назначает ему кластеры из области файлов. FAT разделена на поля, однозначно соответствующие кластерам диска, предназначенным для размещения файлов. Эти поля могут составлять 12 бит (для гибких дисков), 16 бит (FAT16) или 32 бита (FAT32).

Первые два поля FAT всегда зарезервированы. Остальные записи FAT описывают использование соответствующих им кластеров диска. Пример содержимого полей FAT для системы FAT16 представлен ниже:

(0)000 – кластер свободен;
(F)FF0–(F)FF6 – зарезервированный кластер;
(F)FF7 – дефектный кластер, если не часть цепочки;
(F)FF8–(F)FFF – последний кластер файла;
(X)XXX – следующий кластер файла.

Каждая запись в каталоге, соответствующая тому или иному файлу, включает номер первого кластера, назначенного данному файлу, который используется как точка входа в FAT. Начиная с этой точки, каждая запись FAT содержит номер следующего кластера файла вплоть до метки конца файла.

Операционная система поддерживает две идентичные копии FAT на каждом томе. Каждый раз при расширении файла или модификации каталога операционная система одновременно изменяет обе копии таблицы. Если при обращении к сектору FAT фиксируется ошибка чтения, операционная система пытается читать другие копии до успешного выполнения операции чтения или исчерпания всех копий.

Корневой каталог в файловой системе FAT16 имеет фиксированное положение на диске и фиксированный размер (256 файлов). В файловой системе FAT32 это ограничение устранено, и корневой каталог может содержать любое количество файлов, как и все остальные каталоги логического диска. Каждый каталог содержит 32-байтовые записи, которые имеют следующую структуру (табл. 4).

Таблица 4

Структура элемента каталога файловой системы FAT

Длина в байтах	Содержание
8	Имя файла
3	Расширение
1	Атрибуты файла
2	Расширение начального кластера (только FAT32)
8	Зарезервировано
2	Время создания или последней модификации
2	Дата создания или последней модификации
2	Начальный кластер
4	Размер файла

Если файл имеет длинное имя, то под него отводится несколько записей, идущих подряд. Каждая запись, кроме первой содержит в первой позиции число 0xE5 (файл удален) для совместимости со старыми версиями операционных систем.

Байт атрибутов файла определяется по битам следующим образом: 0 – "только для чтения"; 1 – скрытый файл; 2 – системный файл; 3 – метка тома; 4 – каталог; 5 – бит архива; 6, 7 – зарезервированы.

В FAT16 корневой каталог имеет особенность, отличающую его от обычных каталогов, - его размер и положение фиксированы, что определяется программой форматирования в процессе инициализации диска.

Область файлов. Операционная система рассматривает эту область как совокупность кластеров, каждый из которых в зависимости от формата диска содержит один или более секторов. Каждому кластеру соответствует запись в FAT, описывающая его текущее состояние: свободен, зарезервирован, назначен файлу или не подлежит использованию (из-за повреждения). Так как первые два поля FAT зарезервированы, то первому кластеру в области файлов присвоен номер 2. При расширении файла операционная система начинает просмотр FAT от кластера, который был назначен последним, что уменьшает фрагментацию файлов и сокращает общее время доступа к ним. Файл называется фрагментированным, когда отдельные фрагменты этого файла размещаются на диске в кластерах, не смежных между собой.

Все каталоги, кроме корневого, представляют собой просто файлы специального типа. Пространство для них выделяется из области файлов, а их содержимое составляют 32-байтовые записи, совпадающие по формату с записями корневого каталога, и описывающие файлы, и другие каталоги. Записи каталога, описывающие другие каталоги, характеризуются байтом атрибутов с установленным битом 4 и нулем в поле длины файла.

Все каталоги, кроме корневого, содержат дополнительно две специальные записи с именами "." и "..". Операционная система размещает эти записи при создании каталога и их нельзя удалить. Запись с точкой является синонимом текущего каталога. Запись с двумя точками является синонимом родительского каталога.

Файловая система NTFS

Как и любая другая система, NTFS делит все полезное место на кластеры – неделимые блоки данных. NTFS поддерживает почти любые размеры кластеров – от 512 байт до 64 Кбайт, неким стандартом же считается кластер размером 4 Кбайт.

Рассмотрим структуру логического тома системы NTFS, показанную на рисунке 23.

Диск NTFS условно делится на две части. Первые 12% диска отводятся под так называемую MFT зону (Master File Table) – пространство, в которое расширяется метафайл MFT. Запись каких-либо данных в эту область невозможна. MFT-зона всегда держится пустой – это делается для того, чтобы самый главный, служебный файл (MFT) не фрагментировался при своем росте. Остальные 88% диска представляют собой обычное пространство для хранения файлов.

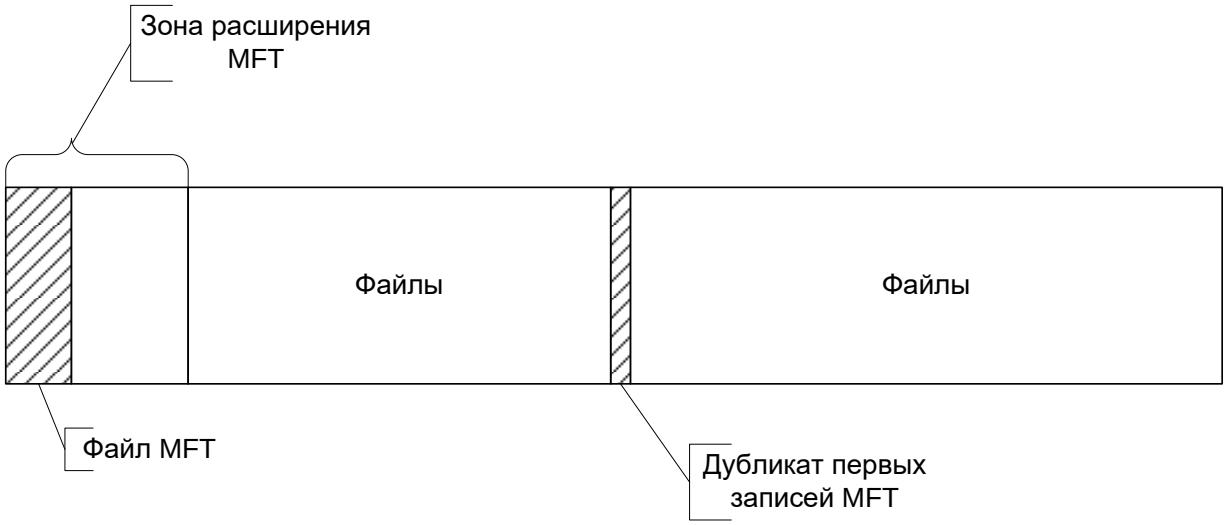

Рис. 23. Структура логического тома файловой системы NTFS.

Свободное место диска, однако, включает в себя всё физически свободное место – незаполненные куски MFT-зоны туда тоже включаются. Механизм использования MFT-зоны таков: когда файлы уже нельзя записывать в обычное пространство, MFT-зона просто сокращается (в версии 4.0 операционной системы Windows NT ровно в два раза), освобождая, таким образом, место для записи файлов. При освобождении места в обычной области MFT зона может снова расширится. При этом не исключена ситуация, когда в этой зоне остались и обычные файлы, это допустимо. Поэтому метафайл MFT все-таки может фрагментироваться, хоть это и нежелательно

В файловой системе NTFS каждый элемент системы представляет собой файл – даже служебная информация. Самый главный файл на NTFS называется MFT или Master File Table – общая таблица файлов. Он размещается в MFT зоне и представляет собой централизованный каталог всех остальных файлов диска, и себя самого. MFT поделен на записи фиксированного размера (обычно 1 Кбайт), и каждая запись соответствует какому-либо файлу.

Первые 16 файлов носят служебный характер и недоступны для использования – они носят название метафайлов. Самый первый метафайл это сам MFT. Эти первые 16 элементов MFT – единственная часть диска, имеющая фиксированное положение. Вторая копия первых трех записей хранится ровно посередине диска, поскольку они очень важны для обеспечения живучести файловой системы. Остальной MFT-файл может располагаться, как и любой другой файл, в произвольных местах диска – восстановить его положение можно с помощью его самого, использовав для этого первый элемент MFT.

Метафайлы находятся в корневом каталоге NTFS диска – они начинаются с символа имени "$". В таблице 5 приведены 10 стандартных метафайлов и их назначение. Остальные 6 элементов зарезервированы для будущего использования.

Таблица 5

Стандартные метафайлы файловой системы NTFS

Название	Значение
$MFT	Собственно файл MFT
$MFTmirr	Копия первых 16 записей MFT, размещенная посередине диска
$LogFile	Файл поддержки журналирования.
$Volume	Служебная информация – метка тома, версия файловой системы, и т.д.
$AttrDef	Список стандартных атрибутов файлов, хранящихся на томе
$.	Корневой каталог
$Bitmap	Карта свободного места тома
$Boot	Загрузочный сектор, (в том случае, если раздел загрузочный)
$Quota	Файл, в котором записаны права пользователей на использование дискового пространства
$Upcase	Файл, содержащий таблицу соответствия заглавных и прописных букв в именах файлов на текущем томе для применяемых языков в формате UTF-16. Используется для ускорения поиска файлов, который может быть задан без учёта прописных символов.

У файлов в NTFS отсутствует понятие хранимых данных, поскольку они разделены на потоки (streams). Один из потоков – собственно данные файла. Большинство атрибутов файла – тоже потоки. Таким образом, базовая сущность у файла только одна – номер в MFT, все остальные компоненты опциональны и могут отсутствовать. Дополнительные потоки файла скрыты и не видны стандартными средствами: наблюдаемый размер файла, – это лишь размер основного потока, который содержит традиционные данные. Имя файла может содержать любые символы, включая полный набор национальных алфавитов, так как данные представлены в Unicode, в формате UTF-16. Максимальная длина имени файла – 255 символов.

Файловая система NTFS отличается от файловой системы FAT значительно более высокой живучестью за счёт транзакционной модели изменения данных и поддержки журналирования. При возникновении сбоев

в работе, например, при внезапной потере питания, данные на файловой системе NTFS могут быть с высокой вероятностью автоматически восстановлены при следующем запуске системы.

Одной из основных особенностей файловой системы NTFS является поддержка атрибутов безопасности применительно к файлам и каталогам. NTFS позволяет хранить в себе полноценный описатель безопасности для каждого файлового объекта. Операционная система при открытии сравнивает хранящийся описатель безопасности на файл (каталог) и права пользователя. Доступ к содержимому файла разрешается только при разрешении заданного вида доступа при проверке.

Файловая система Ext

Файловая система Ext изначально разрабатывалась для операционной системы Linux. За многие годы эта файловая система прошла большой путь своего развития. В настоящее время актуальной является четвёртая версия этой системы, которая поддерживает размер тома до 1 экзабайта. Размер одного файла может достигать 16 Тб.

Начиная с версии Ext3, эта файловая система поддерживает журналирование, что повышает надёжность хранения данных и гарантирует их целостность. В файловой системе поддерживается 3 вида журналирования, это позволяет находить баланс между производительностью и требуемой надёжностью хранения данных.

Операционная системаWindows файловую систему Ext не поддерживает.

Управление памятью

Отображения памяти, виртуальное адресное пространство

Когда мы говорим о памяти в вычислительных системах, то подразумеваем 4 уровня представления памяти: в виде символьных переменных в программе, в виде относительных адресов памяти, которые получаются после компиляции исходного файла программы, в виде ячеек виртуальной памяти, которые получаются после компоновки программы и, наконец, в виде ячеек физической памяти, которые, собственно, и содержат данные [1]. Задача системного программного обеспечения, включая ядро операционной системы, компилятор и компоновщик - связать каждое символьное имя в программе с физической ячейкой памяти (рис. 24).

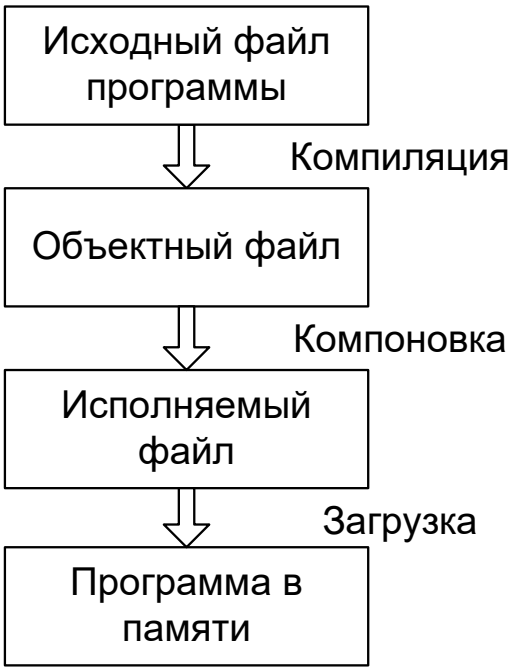

Рис. 24. Последовательность преобразований над кодом программы.

В результате переменная в программе, написанной на языке высокого уровня, подвергается преобразованиям в адресации, которые позволяют на этапе исполнения программы рассматривать её как ячейку физической памяти, которая может адресоваться с помощью машинных инструкций (рис. 25).

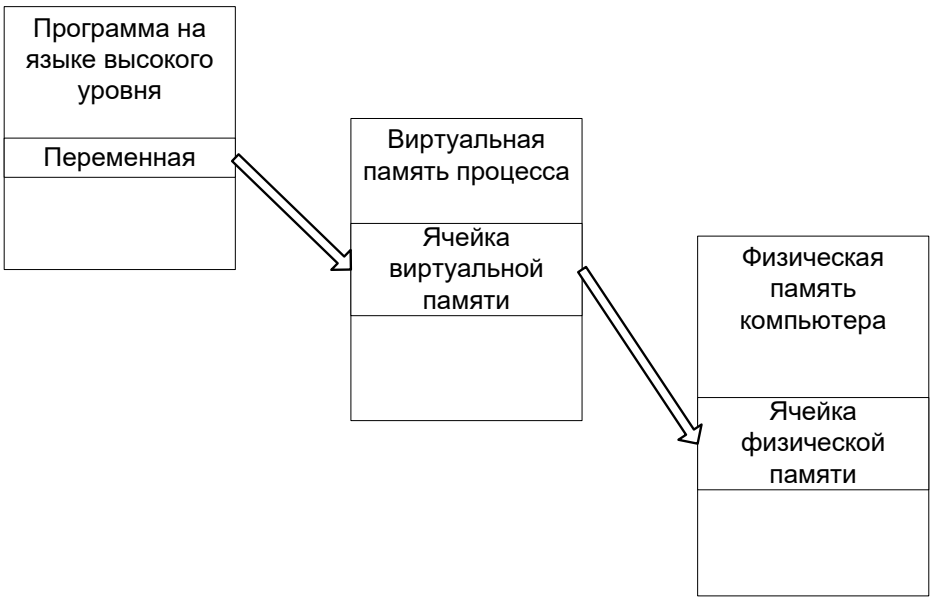

Рис. 25. Преобразование адресации данных.

Таким образом, компиляцию, компоновку программы и загрузку скомпилированной программы в память можно представить в виде единого процесса, задача которого – получить исполняемый код программы в памяти компьютера. Этот процесс искусственно разделяется на две части, одна из которых включает в себя компиляцию и компоновку программы, а другая – загрузку (рис. 26). Левая часть объединяет в себе все предварительные действия по созданию исполняемого кода программы, а правая – весь набор действий по загрузке программы в память.

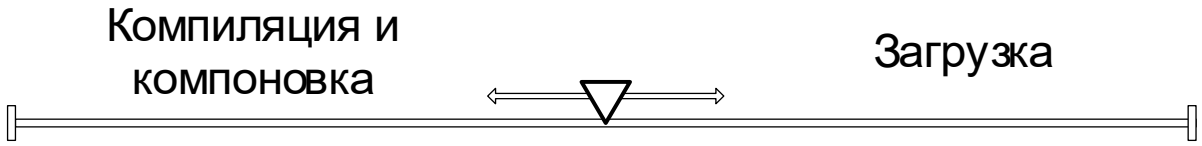

Рис. 26. Представление компиляции, компоновки и загрузки в виде единого процесса.

Для каждой информационной системы необходимо определить оптимальное положение разделителя на шкале, представленной на рисунке. Так, если максимально сместить разделитель вправо, то вся программа будет состоять из одного исполняемого файла, выполняющего весь код программы, включая обращения к системным вызовам. В результате загрузка программы максимально упрощается, поскольку все действия выполняются на предварительном этапе. Однако такая программа будет требовать от операционной системы использования максимального объема памяти.

Наоборот, если разделитель сместить влево, то конечная программа будет состоять из множества мелких исполняемых файлов, включая также различные вспомогательные библиотеки. Это значительно усложняет и замедляет загрузку программы в память, особенно, если на этапе загрузки будет выяснено, что два или более исполняемых файла претендуют на одинаковое местоположение в виртуальном адресном пространстве. В этом случае загрузчик будет вынужден произвести передислокацию одного из исполняемых файлов на новое местоположение в виртуальной памяти. Однако такой подход значительно экономит оперативную память за счет появления возможности использования одного и того же исполняемого файла библиотек в нескольких программах одновременно.

Основываясь на исторической перспективе, оказывается, что одно из важнейших влияний на структуру программного обеспечения оказывает соотношение размера виртуальной памяти процесса V_v и физической памяти компьютера V_p.

Виртуальная память может быть меньше физической $V_v < V_p$, равна физической $V_v = V_p$, и больше физической $V_v > V_p$. Оказывается, что в своём развитии вычислительные устройства последовательно проходят через три перечисленных стадии этого соотношения. Сначала в каждом новом поколении вычислительной техники виртуальная память значительно меньше физической, затем она становится равна физической памяти и уже затем начинает превышать размер физической памяти.

Применительно к персональным компьютерам первая стадия – это реальный режим микропроцессора 8086 и первых версий системы MS-DOS. Размер программы не мог превышать 64 Кб памяти. Затем размер виртуальной и физической памяти сравнялся и стал равен 640 Кб. Далее настал черед 32-битных операционных систем, глее виртуальная память в 4 Гб значительно превышала размер физической памяти.

Процесс изменения соотношения виртуальной и физической памяти всегда развивается по спирали с одновременной миниатюризацией вычислительных устройств. За персональными компьютерами пришла очередь смартфонов, которые также развивались по подобному сценарию. Следующий виток спирали, вероятно, будет пройден для IoT устройств (интернет вещей).

Для понимания принципов построения технологий управления памятью необходимо проанализировать каждый из возможных вариантов такого управления. Мы будем рассматривать эти технологии по порядку, начиная с самых простых систем распределения памяти.

Методы управления памятью и способы её распределения

Простое непрерывное распределение

Простое непрерывное распределение (рис. 27), это исторически самое первое распределение памяти. Вся физическая память делится на три части: область, занимаемая операционной системой, область программы и свободная область памяти.

При использовании простого непрерывного распределения операционная система не поддерживает мультизадачность и, поэтому, не возникает проблема распределения памяти между несколькими задачами. Область памяти получается непрерывной, что облегчает работу компилятору. Виртуальные адреса совпадают с физическими, поэтому привязка виртуальных адресов к физическому адресному пространству фактически не требуется.

Рис. 27. Простое непрерывное распределение памяти.

Достоинства: недорогая и легко программируемая реализация.

Недостатки: однозадачный режим работы, неэффективное использование памяти, поскольку далеко не каждая программа использует память целиком.

Пример: с натяжкой первые версии MS-DOS.

Оверлейное распределение

Оверлейное распределение памяти или, иначе, распределение с перекрытием (рис. 28) возникло как развитие идеи простого непрерывного распределения. Если адресное пространство программы оказывается больше чем размер оперативной памяти, то программу можно разбить на части. Каждая программа имеет одну главную часть и несколько, так называемых, оверлеев – сегментов которые могут перекрываться по адресному пространству. В памяти всегда находится главная часть и один или несколько оверлеев, которые не перекрываются между собой. Остальные оверлеи располагаются на диске. Как только возникает необходимость исполнения выполнить оверлея, которых находится на накопителе, он загружается в оперативную память, перекрывая, возможно, другие оверлеи. Такая реализация программы может быть сделана с помощью системы программирования на этапе компиляции программы.

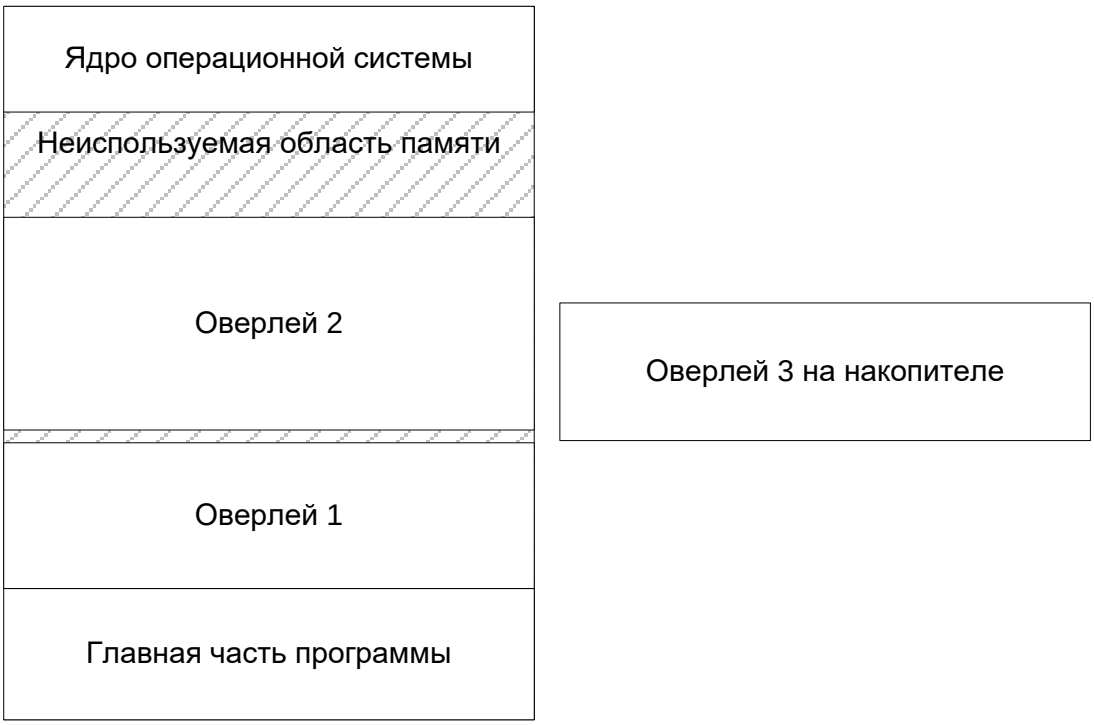

Рис. 28. Оверлейное распределение памяти.

Пример: среда программирования Borland C 3.1 для MS-DOS.

Достоинства: размер программы может превышать размер оперативной памяти.

Недостатки: накладные расходы на загрузку и выгрузку оверлеев.

Разделы с фиксированными границами

Развитие вычислительной техники привело к осознанию необходимости размещения в памяти одновременно нескольких программ. Самая простая схема распределения памяти заключается в следующем: память, не занятая ядром, может быть разбита на несколько непрерывных частей. Части могут быть одного размера или их размеры могут различаться. В последнем случае составляется специальная таблица разделов, в которой указывается имя раздела, его тип, адрес начала раздела и его длина. Если все разделы одной длины, либо размеры разделов не изменяются со временем, то алгоритмы управления памятью проще. Если же границы разделов могут сдвигаться, то алгоритмы управления памятью становятся сложнее, однако при этом достигается большая эффективность использования памяти.

На рисунке 29 представлен вариант организации распределения памяти разделами с фиксированными границами, где все разделы имеют одинаковые размеры.

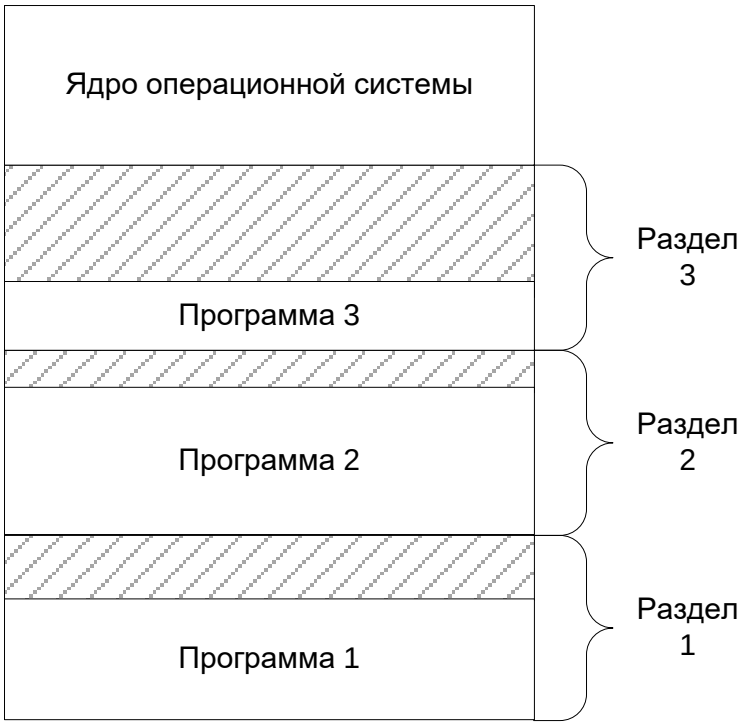

Рис. 29. Разделы с фиксированными границами равного размера.

В каждом разделе в один момент времени располагается по одной программе, поэтому для каждого раздела используются те же методы, что и для однопрограммных систем. Причем, эта схема допускает возможность мультипрограммного режима работы операционной системы. Первые мультипрограммные операционные системы строились именно так. При небольшом объеме памяти увеличить количество одновременно

выполняемых приложений можно за счет выгрузки разделов на диск и загрузки при необходимости обратно. Этот процесс называется **своппингом** (swapping) [1,4].

Достоинства: Возможность мультипрограммного режима работы.

Недостатки: Подобная схема реализации распределения памяти содержит два больших недостатка:

1. Отсутствие защиты одной программы от другой и защиты ядра операционной системы от программ.
2. Наличие большого объема неиспользуемой памяти, поскольку неиспользуемая память есть в каждом разделе, а разделов несколько. Такие потери памяти стали называть фрагментацией памяти.

Чтобы избавится от недостатков предложенной схемы, были предложены следующие решения:

1. Для каждой программы выделять раздел ровно такого объема, который нужен для решения текущей задачи. Такой метод называется «разделы с подвижными границами». Он призван решить вопросы фрагментации памяти.
2. Размещать задачу не в одной непрерывной области, а нескольких областях – это называется «сегментная организация памяти». Такой метод реализует уменьшение фрагментации памяти, а также обеспечивает защиту памяти программ друг от друга, а также защиту ядра операционной системы.

Рассмотрим перечисленные методы подробнее.

Разделы с подвижными границами

При реализации метода распределения памяти разделами с подвижными границами (рис. 30) память выделяется либо побайтно, либо некоторыми дискретными единицами, кратными степени двойки. Например, память может выделяться с градацией 4 Кб. Поскольку по всему пространству памяти может быть множество неиспользуемых областей, то для выделения раздела необходимо осуществлять поиск таких областей. Некоторые свободные области могут быть меньше по размеру, чем требуется, и поэтому их необходимо исключить из рассмотрения; некоторые области могут быть больше по размеру. Оптимальным вариантом является нахождение области, которая в точности совпадала бы с запрашиваемым размером раздела.

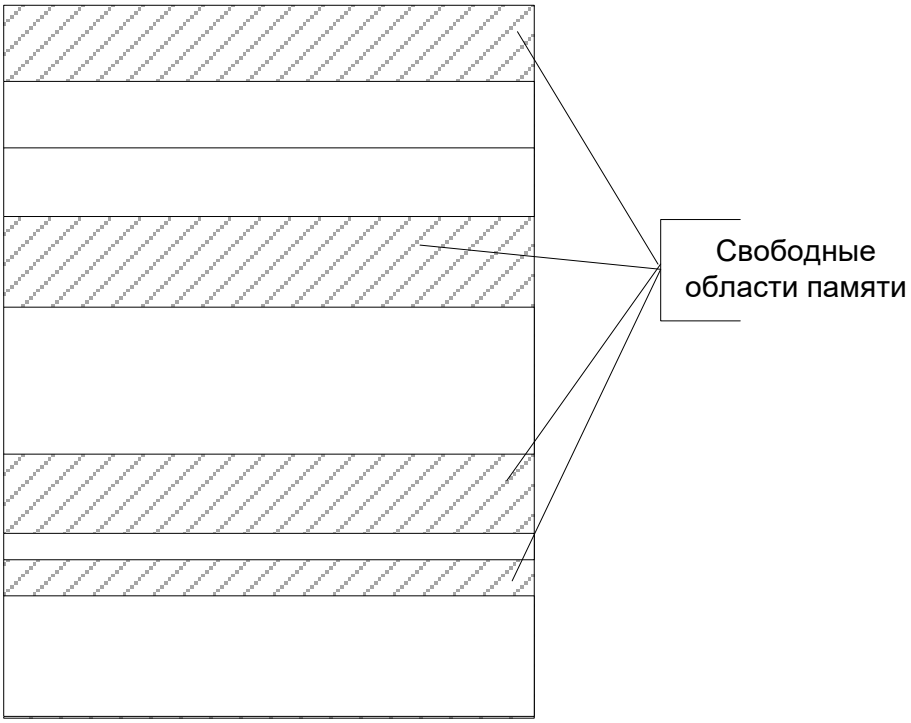

Рис. 30. Разделы с подвижными границами.

При выделении ищется такая область свободного адресного пространства, которая бы наиболее подходила под запрос памяти. Здесь возможно три способа поиска: первый подходящий по размеру участок, самый подходящий по размеру участок, самый большой участок. При первом способе поиск идет достаточно быстро, но фрагментация остается. Второй способ дольше, но и при его использовании остается фрагментация, причем размер остающихся областей настолько мал, что туда уже ничего не удаётся поместить. Относительно эффективным является последний способ. Если из самого большого участка выделить память, то существует вероятность, что из него можно будет выделить память еще раз. К сожалению, применение третьего способа приводит к постоянным выделениям памяти из хвоста списка выделенных областей и, как следствие появлению фрагментации по всему объёму памяти.

При использовании любого из трёх перечисленных способов фрагментация памяти со временем накапливается. Это приводило к необходимости решения проблемы фрагментации. Кардинальным методом решения проблемы фрагментации являлась перезагрузка операционной системы, но, конечно, такой способ нельзя признать допустимым. Поэтому к решению проблемы были предприняты другие подходы, которые необходимо рассмотреть.

Достоинства: Меньшая фрагментация по сравнению с разделами с фиксированными границами.

Недостатки: фрагментация остается и накапливается со временем.

Применение таблиц градаций размеров для решения проблемы фрагментации памяти

Основная проблема фрагментации памяти заключается в неупорядоченном и во многом хаотичном расположении областей фрагментов свободной памяти. Если задать таким фрагментам структуру, то фрагментацию памяти удаётся свести к минимальным значениям, и, что самое главное, фрагментация памяти перестаёт накапливаться с течением времени. Принцип такой организации памяти заключается в создании нескольких компонентов – таблиц памяти, внутри каждой из которых разделы имеют фиксированные границы (рис. 31). Минимизации фрагментации памяти удаётся при этом добиться наличием множества подобных таблиц. Лучше всего принцип иллюстрируется рисунком.

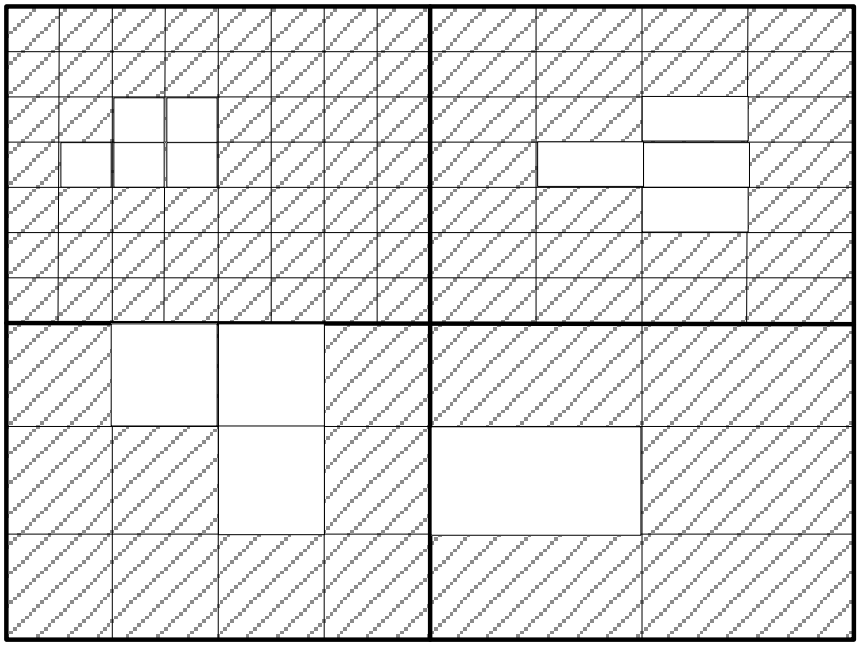

Рис. 31. Таблица градаций размеров областей памяти.

На рисунке схематично представлены 4 таблицы градаций размеров для выделения памяти. При необходимости выделения памяти будет выбрана та таблица, которая наиболее подходит по размеру под запрос. Впоследствии, когда каждая из выделенных областей памяти будет освобождаться, это не приведёт к увеличению фрагментации памяти, поскольку фрагменты не будут выходить за границы ячеек таблицы градации размеров.

Достоинства: Отсутствие накопления фрагментации памяти со временем.

Недостатки: Увеличение потребления памяти из-за выравнивания адресов до границ ячеек таблицы.

Применение ссылок для решения проблемы фрагментации памяти

Еще один метод решения задачи устранения фрагментации памяти заключается в использовании ссылок для адресации объектов (рис. 32). Ссылка представляет собой указатель на объект. Когда в программе организуется работа над объектом, то применяется указатель не на сам объект, а указатель на ссылку. Ссылка располагается по определённому адресу, поэтому указатель на ссылку может использоваться без ограничений. Сам объект, на который указывает ссылка, при этом может перемещаться в памяти. Необходимо только лишь обеспечить отсутствие доступа к объекту в то время, пока перемещение имеет место.

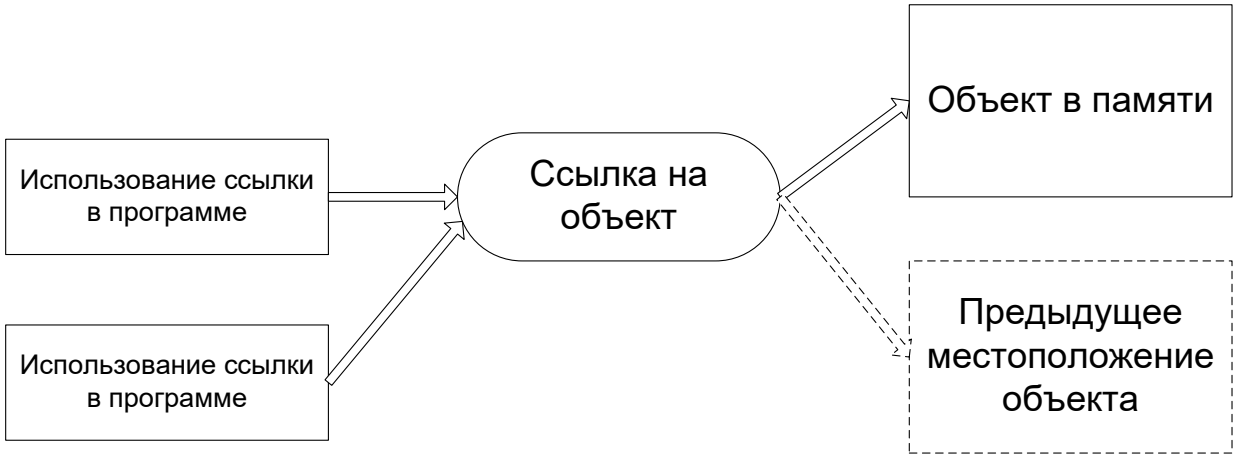

Рис. 32. Применение ссылок на данные в коде программы.

На принципе применения ссылок на объект возможно построение систем, в которых местоположением объектов в памяти управляет не программист, а специальный алгоритм в составе исполняющей машины. Исполняющая машина ведёт учёт ссылок на объект. Когда количество ссылок на объект становится равным нулю, объект удаляется из памяти. Также при необходимости возможно осуществление операции уплотнения памяти, которая смещает все объекты к началу области памяти. Подобный алгоритм носит название **сборщик мусора** (garbage collector).

Достоинства: учёт количества объектов в памяти, возможность проведения операции уплотнения памяти.

Недостатки: фрагментация накапливается со временем, что и диктует необходимость проведения операции уплотнения памяти.

Пример: На базе этого метода работают исполняющие машины JVM и .NET.

Сегментный способ организации виртуальной памяти

Сегментная организация памяти, это исторически первый метод разрывного распределения памяти. Идея выделять память задаче не одной сплошной областью, а фрагментами требует для своей реализации соответствующей аппаратной поддержки в виде относительной адресации. При использовании такой адресации эффективный адрес может быть вычислен на основе таблицы, в которой хранятся адреса всех фрагментов. Адрес объекта в программе при этом разбивается на две части: номер сегмента и смещение. Номер сегмента адресует строку в таблице, в которой описывается указанный фрагмент (сегмент) памяти, а смещение задаёт адрес внутри сегмента.

Каждая строка таблицы сегментов называется дескриптором сегмента и содержит в себе описание адреса и размера блока памяти и прав доступа к нему. Также возможно создание нескольких дескрипторов в таблице сегментов, различающихся по правам доступа к сегменту, но при этом указывающих на одну и ту же область физической памяти.

Если за использование таблицы сегментов отвечает аппаратура процессора, тогда такой метод становится безопасным для использования в многозадачных операционных системах, поскольку обеспечивает распределение доступа к данным и коду программ в памяти. Современные процессоры архитектур x86, ARM различных версий обеспечивают такую организацию памяти.

Для использования сегментной адресации при написании программ код программы необходимо разбивать на части и уже каждой такой части выделять физическую память. Единственный логичный способ такого разбиения – разбиение на логические сегменты, например, каждый модуль (файл) помещать в отдельный сегмент. Каждый сегмент при этом является самостоятельной единицей. Однако увеличение размера таблицы сегментов приводит к существенному замедлению выполнения программы, поскольку количество дескрипторов сегментов, кэшируемых процессором, ограничено. Поэтому возможен вариант, когда весь код программы объединяется в один сегмент кода, равно как и все данные программы объединяются в единый сегмент данных.

Ниже представлен упрощённый рисунок сегментного способа организации виртуальной памяти (рис. 33).

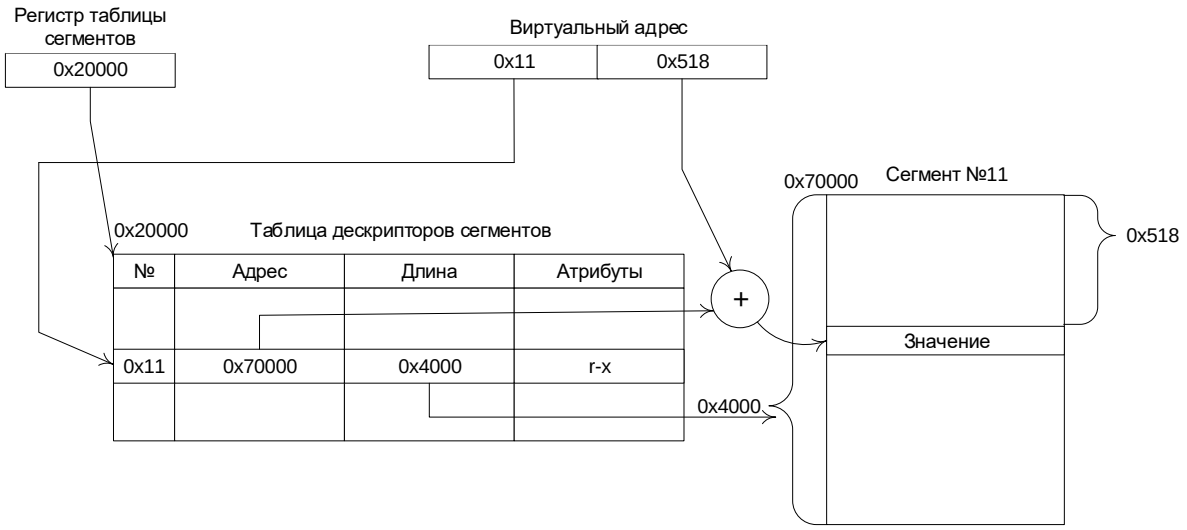

Рис. 33. Сегментный способ организации виртуальной памяти.

Операционная система строит для системы и для каждого процесса таблицу дескрипторов сегментов. Если сегмент находится в оперативной памяти, то об этом делается пометка в дескрипторе, так называемый «бит присутствия». Иначе сегмент находится на диске. В дескрипторе имеется также поле «длина сегмента», которое позволяет проконтролировать обращение программы за пределы сегмента (нарушение адресации) и генерировать сигналы прерывания. Кроме этого в дескрипторе содержится информация о типе сегмента (код или данные), о виде доступа (только на чтение, чтение/запись).

Для вычисления сегмента – кандидата на свопинг идеально было бы хранить в дескрипторе сегмента данные о последнем времени его использования. Однако процессоры предоставляют для этих целей только два бита: один бит для фиксации любого доступа к сегменту и второй бит для фиксации обращения к сегменту на запись. Поэтому операционная система вынуждена определять кандидата на свопинг, основываясь на применении различных косвенных признаков.

Сама таблица дескрипторов представляет собой такой же сегмент данных.

Этот способ позволяет организовать мультипрограммный и мультизадачный режимы работы операционной системы. Однако следует помнить, что при превышении суммарного размера часто использующихся сегментов физического размера памяти очень много времени начинает тратиться на своппинг. Такое явление называется пробуксовкой.

Важнейшим преимуществом применение сегментной организации памяти является защита памяти. Чтобы процессы не могли испортить код и данные ядра операционной системы необходимо, чтобы доступ к коду и данным операционной системы (в том числе к таблице дескрипторов) имела

только сама операционная система. Для распределения полномочий между процессами и системой применяется привилегированный режим работы кода операционной системы, который реализуется аппаратно. Кроме того, каждая программа должна иметь возможность обращаться только к своим собственным сегментам памяти.

Достоинства: 1) Если код программы разбит на несколько сегментов, то возможно при загрузке программы размещать ее код в памяти не целиком, а по мере необходимости. 2) Код некоторых программных модулей может использоваться несколькими программами одновременно. Например, код прикладных библиотек.

Недостатки: Сложность организации метода. Фрагментация памяти хоть и уменьшается, но остается. При использовании метода большие потери времени приходятся на размещение и обработку дескрипторных таблиц.

Пример: Исторически: MS Windows 3.1, OS/2 версия 2.0. В настоящее время в чистом виде не используется.

Страничная организация памяти

При этом способе все фрагменты программы, на которые она разбивается (за исключением последней части), получаются одинаковыми. Соответственно и единицы памяти для фрагментов программы получаются одинаковыми. Эти части называются страницами. Говорят, что оперативная память разбивается на физические страницы, а программа и данные – на виртуальные страницы. Виртуальная страница в один момент времени находится либо в оперативной памяти, либо во внешней памяти, на диске в специальном файле, который называется файлом подкачки (своппинга). В некоторых операционных системах (Linux) это не файл, а дополнительный дисковый раздел.

Величина каждой страницы выбирается кратной степени двойки. У процессоров Intel размер страницы в минимальном варианте равен 4 Кб, у процессоров MIPS – 2 Кб, у процессоров ARM – 1 Кб. Также возможны варианты, когда размер страницы значительно больше, например, у процессоров Intel также возможно задавать страницы размерами 2 Мб и 1 Гб, которые будут выделяться рядом со страницами размером в 4 Кб.

Задача операционной системы во время исполнения программы сводится к отображению затребованной виртуальной страницы на физическую. При этом суммарный размер виртуальных страниц, как уже отмечалось, может быть больше размера оперативной памяти.

Чтобы правильно отобразить страницы, нужно иметь таблицу соответствия виртуальных страниц физическим. Такая таблица называется

таблицей страниц. Дескриптор страницы проще дескриптора сегмента – не нужно поле длины, поскольку все страницы одинакового размера. Трансляция виртуальных страниц на физические идет через бит присутствия, как и в сегментном способе. Если в дескрипторе страницы бит присутствия сброшен, то при обращении к ней сработает прерывание (исключение, в терминах Intel), и управление будет передано менеджеру памяти операционной системы, который извлечёт страницу из файла подкачки. У каждой страницы есть свой код доступа. Если программа требует доступ, отличный от имеющегося, то работа программы прерывается.

На рисунке 34 представлен способ страничной организации памяти в существенно упрощённом виде. В реальных процессорах Intel для получения конечного физического адреса страницы при самом простом режиме 32-битной страничной адресации используется не одна, а две вложенных таблицы дескрипторов страниц (PT и PD). В режиме x64 процессоров Intel применяется уже дерево из 4 вложенных таблиц дескрипторов страниц (PT, PD, PDPT, PML4).

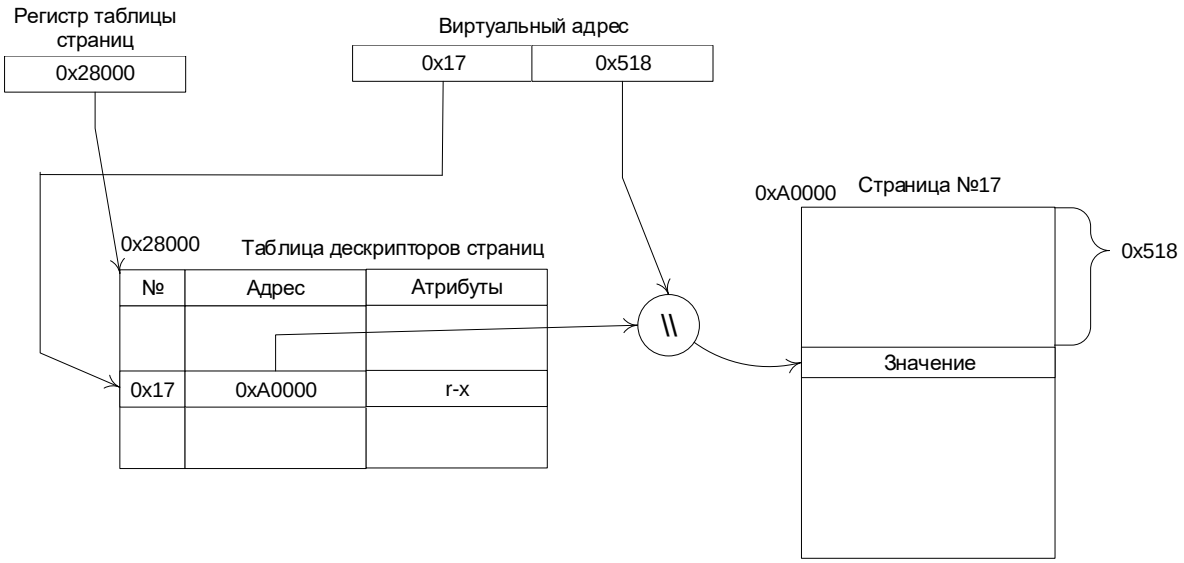

Рис. 34. Страничный способ организации виртуальной памяти.

Необходимо обратить внимание, что для получения конечного физического адреса ячейки памяти не требуется производить сложение адреса начала страницы со смещением внутри страницы; для получения адреса достаточно выполнить простейшую конкатенацию адресных линий.

Если при первом обращении к странице её не оказывается в памяти, то возникает прерывание и управление передается менеджеру памяти, который отдает процессу первую же свободную физическую страницу. Если таковой

нет, то происходит замена страниц. Алгоритм выбирает для замещения ту страницу, к которой не было обращений дольше всех.

Поскольку в дескрипторе страницы нет поля, в котором можно было бы хранить время последнего доступа к странице, то вычисление кандидата на свопинг происходит по относительно сложному алгоритму. В случае аппаратной поддержки страничного способа организации памяти процессор устанавливает бит обращения в дескрипторе страницы автоматически при любом обращении к странице. Сбрасывается этот бит программно. Менеджер памяти осуществляет поиск кандидата на свопинг на основе исследования дескрипторов множества страниц.

Как и в случае с сегментным способом организации памяти, страничный механизм без аппаратной поддержки существенно замедляет работу памяти. Наиболее эффективный способ ускорения работы – создание аппаратного страничного кэша для страничных дескрипторов. Начиная с процессора 80386 в процессорах Intel был страничный кэш на 32 вхождения, который в последующих моделях только увеличивался.

Пример: метод применяется в операционных системах Windows и Linux.

Достоинства: минимально возможная фрагментация физической памяти.

Недостатки: существенные накладные расходы, разбиение на страницы без учета логических взаимосвязей в программе, поэтому межстраничные переходы осуществляются чаще, чем межсегментные.

Чтобы избавится от второго недостатка был предложен сегментно-страничный способ организации памяти.

Сегментно-страничный способ организации памяти

Сегментно-страничный способ организации памяти представляет собой сочетание сегментного и страничного способов управления памятью. При его использовании сегменты оперируют адресами виртуальной памяти, которые затем преобразовываются в адреса физической памяти с помощью страничного преобразования.

Адрес ячейки памяти, по факту, состоит из трёх частей: номера сегмента, номера виртуальной страницы и смещения в странице. При обращении по заданному адресу запрос сначала проходит проверку на основе дескриптора сегмента. Если проверка пройдена, то на выходе получается виртуальный адрес ячейки памяти. Этот виртуальный адрес затем проходит проверку на основе дескрипторов страницы. Если все проверки пройдены, то происходит обращение к физической ячейке памяти.

Ниже представлен рисунок сегментно-страничного способа организации виртуальной памяти (рис. 35). Также, как и два предыдущих рисунка, он представлен в сильно упрощённом виде.

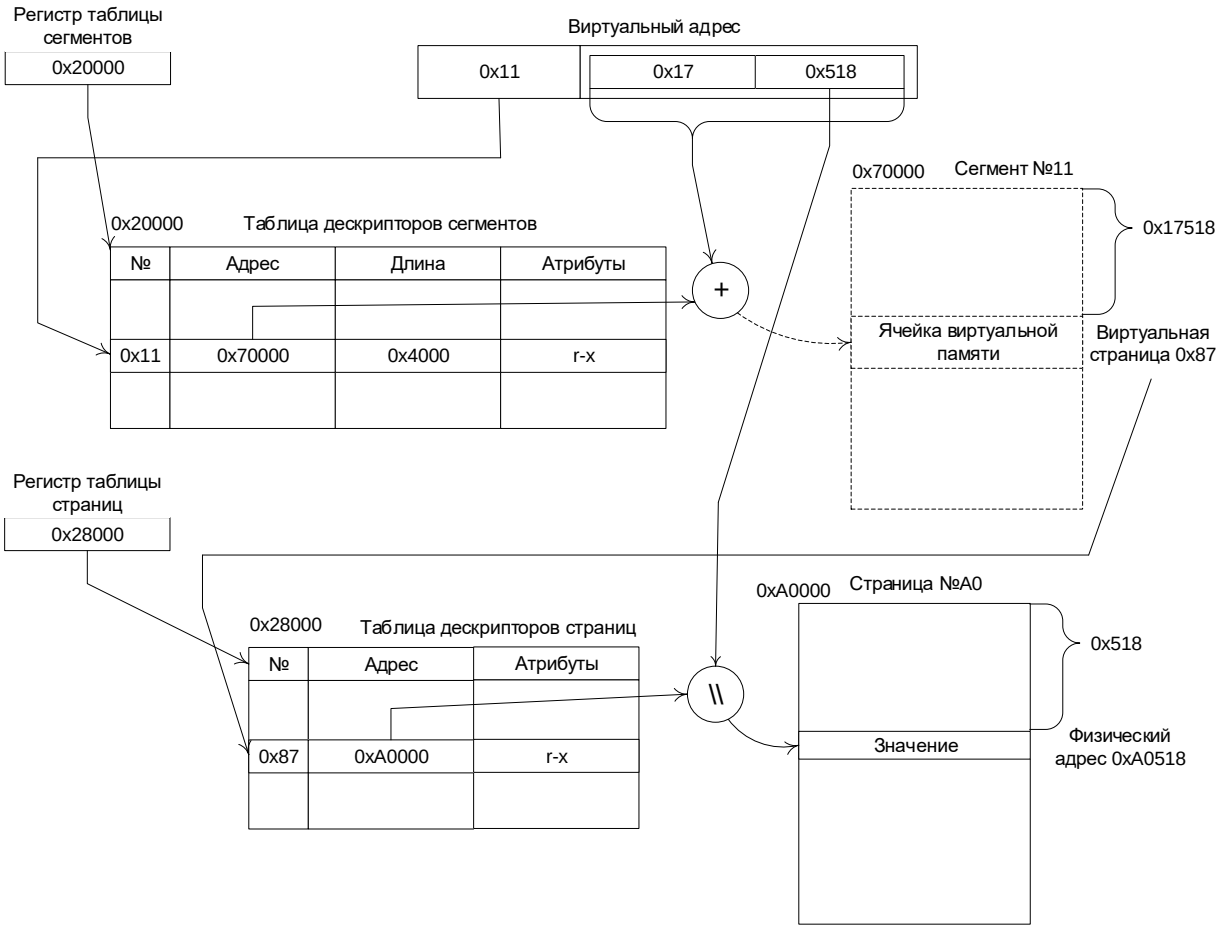

Рис. 35. Сегментно-страничный способ организации виртуальной памяти.

Из рисунка видно, насколько усложняется, и как следствие, замедляется обращение к памяти. Сначала нужно вычислить и получить дескриптор сегмента, затем вычислить адрес дескриптора страницы для этого сегмента и получить его из таблицы страниц, и только после этого приписать в конец адреса индекс ячейки памяти. Задержка доступа в три раза больше чем при простой прямой адресации. Единственный способ решения – аппаратный. Наиболее эффективным аппаратным способом является ассоциативный кеш. В качестве тега берут номер сегмента и страницы, а результат функции – адрес физической страницы.

Необходимо также отметить, что рисунок, иллюстрирующий сегментно-страничный способ организации памяти, также представлен со значительными упрощениями в части страничного преобразования адреса по сравнению с реальными таблицами дескрипторов страниц.

Достоинства: Этот способ соединяет в себе несколько достоинств. Разбиение программы на сегменты позволяет размещать сегменты в памяти целиком. Сегменты разбиваются на страницы, и редко используемый код удаляется из памяти. Поскольку программа разбита на сегменты согласно внутренней логике, число межсегментных переходов минимально. Страницы сегмента расположены в памяти, но не подряд, а россыпью, и можно не заботиться о фрагментации памяти. Наличие сегментов дает эффективную защиту одной программы от другой, а также защищает ядро операционной системы от программ. Возможна динамическая компоновка адресного пространства процесса.

Пример: в настоящее время способ практически не используется из-за чрезмерной сложности управления памятью со стороны операционной системы.

Управление физической памятью

Распределение первого мегабайта оперативной памяти

Первый мегабайт оперативной памяти персональных компьютеров необходимо рассматривать отдельно от всей остальной физической памяти по историческим причинам. В первом мегабайте располагалось всё доступное адресное пространство персонального компьютера на процессорах Intel 8086/8088 – исторически первых успешных в маркетинговом плане процессоров для персональных компьютеров. В первом мегабайте оперативной памяти располагались все отображения аппаратных устройств, базовая система ввода-вывода и оперативная память компьютера. Такой компьютер управлялся операционной системой MS-DOS.

MS-DOS – однозадачная операционная система. В один момент времени в ней исполняется одна программа, которая может быть запущена из другой программы, находящейся в памяти, но в данный момент не используемой. Поэтому можно говорить, что в MS-DOS использовалось распределение памяти с фиксированными границами.

Из-за особенностей организации микропроцессора 8086 при двух используемых совместно 16 разрядных адресных регистрах можно адресоваться только к 1 Мб оперативной памяти. Это происходит потому, что конечный физический адрес получался сложением этих двух 16 разрядных регистров с перекрытием 12 бит, как показано на рисунке 36.

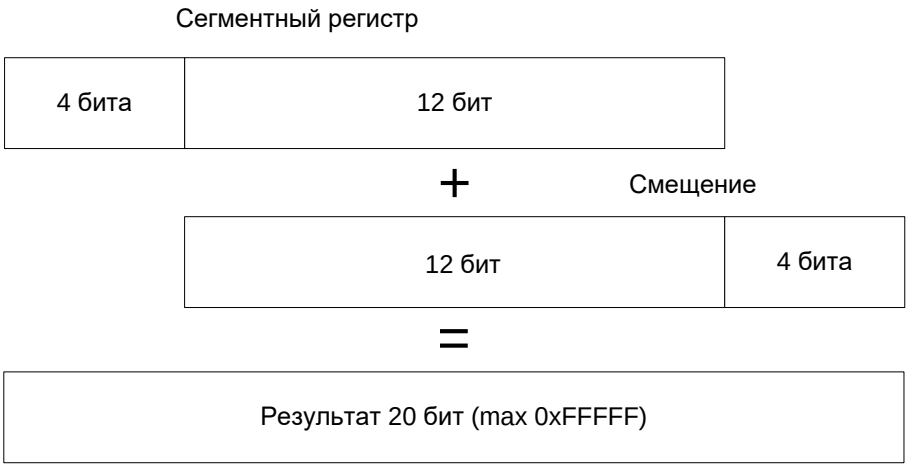

Рис. 36. Образование адреса в реальном режиме работы процессора Intel 8086.

Вся память при этом разбивается на псевдосегменты с гранулярностью 16 байт, которую обеспечивают младших 4 бита смещения. Для выхода за пределы 64 Кбайт требуются межсегментные переходы.

Программно память разбивается на три части (рис. 37). В самых младших адресах (1024 байта) находится таблица векторов прерываний процессора 8086. Вторая часть памяти отводится для размещения программных модулей самой MS-DOS и программ пользователя. Эта область памяти называется Conventional memory. Её максимальный размер равен 640 Кб. Наконец, третья часть адресного пространства отведена для постоянных запоминающих устройств и функционирования некоторых устройств ввода-вывода. Эта область памяти получила название Upper memory.

В области Upper memory располагается базовая система ввода-вывода компьютера. Она расположена по адресу 0xE0000 и занимает 256 Кб памяти до границы первого мегабайта. Диапазон адресов от 0xA0000 до 0xC0000 отводится для видеопамяти, которая может работать в разных видеорежимах.

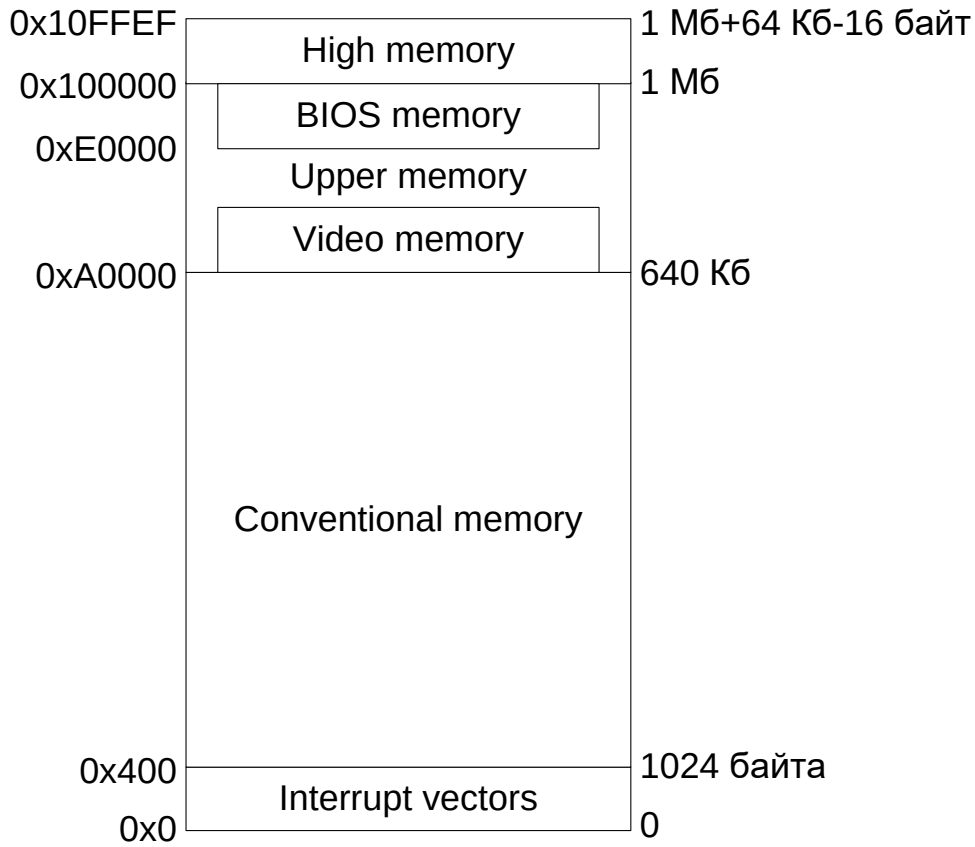

Рис. 37 Организация физической памяти с точки зрения операционной системы MS-DOS.

В младших адресах основной памяти размещается то, что можно назвать ядром этой операционной системы – системные переменные, основные программные модули, блоки данных для буферирования операций ввода-вывода. Для управления устройствами, драйверы которых не входят в BIOS, загружаются инсталлируемые драйверы. Перечень инсталлируемых драйверов определяется специальным конфигурационным файлом CONFIG.SYS. После загрузки расширения BIOS – файла IO.SYS, этот файл загружает файл MSDOS.SYS – собственно операционную систему, считывает в память файл CONFIG.SYS и в соответствии с ним загружает в память требуемые драйверы.

Остальные модули MS-DOS оформлены как программы. Командный процессор COMMAND.COM делится на две части: резидентную и транзитную. Последняя загружается в область для программ по старшим адресам и восстанавливается при необходимости, если какая-либо программа во время работы использовала эту область памяти.

Необходимо также обратить внимание, что при сложении с перекрытием двух чисел 0xFFFF, соответствующих сегментному регистру и

смещению получается число не 0xFFFFF, а число 0x10FFEF, то есть 1 Мб+64 Кб-16 байт. В первых процессорах старший разряд полученного числа отбрасывался и получался адрес 0xFFEF, то есть адрес «заворачивался» в начало первого мегабайта.

В последующих моделях процессоров появилась возможность адресоваться к памяти, размером больше 1 Мб, однако для совместимости с первыми моделями процессоров адресное пространство также «заворачивалось» в начало первого мегабайта. Это достигалось сбросом в ноль двадцатой линии по шине адреса (так называемая, линия A20). Чтобы получить возможность адресоваться к памяти за пределами 1 Мб была создана специальная команда, которую необходимо было передать почему-то в контроллер клавиатуры. В результате появилась возможность адресовать ещё почти 64 кб памяти, которую стали называть High memory.

При использовании старших моделей микропроцессоров и драйвера HIMEM.SYS модули IO.SYS и MSDOS.SYS могут быть размещены за пределами первого мегабайта в области High Memory. Память за пределами области High memory стали называть Extended memory. Для её использования понадобилось вводить новый режим работы процессора, который был назван Protected mode в противовес режиму Real mode, использующему только первый мегабайт памяти. В режиме Protected mode появилась возможность адресовать до 4 Гб физической памяти, используя тридцать две адресные линии.

ACPI, UEFI и распределение физической памяти за пределами первого мегабайта

ACPI – это аббревиатура, которая расшифровывается как Advanced Configuration and Power Interface [10]. Это подсистема в составе BIOS современных персональных компьютеров, позволяющая управлять настройками и энергопотреблением аппаратуры. Операционная система взаимодействует с ACPI, для управления устройствами компьютера.

Для получения информации из BIOS операционная система следует специальному алгоритму. BIOS содержит в своём составе набор специальных таблиц, которые после преобразования можно развернуть в, так называемое, дерево устройств. Для любого из устройств может быть предоставлен набор функций по его конфигурированию и управлению энергопотреблением. Эти функции написаны на специальном языке в виде байт-кода, называемого AML или Acpi Machine Language. Благодаря применению байт-кода операционная система может сама исполнять функции для устройств ACPI. Для этого ей нужно иметь в своём составе интерпретатор команд AML. Примером AML инструкции может служить

требование записи определенной константы в определённый порт ввода-вывода.

Байт-код языка AML получается путём компиляции функций, представленных в текстовом виде на другом языке, называемом ASL (Acpi Source Language). Язык ASL более понятен для человека, поэтому на нём пишутся функции управления устройствами ACPI.

Функции языка AML при старте компьютера загружаются BIOS или экстендерами периферийных устройств в оперативную память, либо изначально находятся в постоянной памяти компьютера. В любом случае они располагаются в адресном пространстве физической памяти.

Операционная система при старте должна получать точную картину распределения областей оперативной памяти по физическому адресному пространству. Для этих целей служит специальный вызов BIOS прерывания int 15h – функция 0e820h. Результатом работы этой функции является таблица, каждая строка которой описывает одну область физического адресного пространства.

Существует несколько типов таких областей адресного пространства. Только адресное пространство типа 1 содержит в себе оперативную память, которую может использовать операционная система. Остальные области считаются зарезервированными.

Таким образом, оперативная память в современных компьютерах может располагаться прерывистыми областями. Если какая-либо часть оперативной памяти располагается выше области в 4 Гб, то операционная система сможет работать с такой памятью только если она работает в 64-разрядном режиме работы процессора, или поддерживает режим PAE (Physical Address Extension) в 32-разрядном режиме работы.

Развитием идеи BIOS в персональных компьютерах является модель и подсистема UEFI [11]. UEFI (Unified Extensible Firmware Interface) – универсальный расширяемый интерфейс производителя, определяет новую модель взаимодействия между операционной системой и разработчиком материнской платы (firmware). Иногда его пишут без первой буквы аббревиатуры: EFI.

Этот интерфейс включает в себя набор таблиц и перечень вызовов, в отличие от классического BIOS, работающих в защищённом режиме работы процессора или даже в 64-битном режиме. Модель UEFI включает в себя понятие драйвера и приложения, одним из которых является загрузчик операционной системы. Таким образом, при использовании модели UEFI некоторые действия компьютер может выполнять даже без наличия работающей операционной системы.

В процессе своей работы операционная система иногда обращается к BIOS. В случае наличия UEFI такие обращения выполнять гораздо проще, то есть модель UEFI в скором будущем вытеснит классические BIOS материнских плат.

Таблица физической памяти в составе операционной системы

В предыдущем разделе было показано, что оперативная память в составе современного персонального компьютера располагается не непрерывно, а участками. Операционная система в процессе своей работы обязана учитывать все страницы физической памяти. Для этих целей в составе операционной системы имеется специальная таблица, каждая строка которой описывает одну страницу физической памяти. В операционной системе Windows эта таблица называется PFN-table [6].

Как нетрудно убедиться, такая таблица должна быть весьма большого размера. Например, если одна строка таблицы будет равна 24 байтам, то при размере оперативной памяти в 16 Гб вся таблица будет занимать 96 Мб. Причём вся эта таблица обязана постоянно находиться в памяти – её нельзя выгружать в файл свопинга, поскольку именно по этой таблице координируется сам процесс свопинга.

Доступ к таблице физической памяти может иметь только ядро операционной системы. Это диктуется соображениями безопасности.

В строке PFN-table обязательно располагается поле флагов. Приведём примеры флагов, которые обязательно должны содержаться в этом поле: страница свободна или занята, страница содержит нули и может быть выделена процессу или страница содержит данные и должна пройти процедуру обнуления, страница содержит данные, которые необходимо передать в свопинг и др. Ещё одно поле в строке таблицы содержит в себе идентификатор процесса, которому принадлежит страница памяти.

Поскольку таблица физической памяти всегда целиком находится в памяти, то её становится удобно рассматривать как массив структур с заданными полями. Индекс в этом массиве однозначно идентифицирует физическую страницу памяти. В операционной системе Windows он называется PFN (Page Frame Number). Роль этого индекса в операциях с памятью очень высока.

Если в процессе работы возникает необходимость передать какому-либо устройству данные, то эти данные будут расположены в виде буфера, находящегося в виртуальной памяти процесса. Однако физическое устройство не имеет доступа к виртуальной памяти процесса – всё, что ему доступно – физическая память компьютера. Данные в составе буфера виртуальной памяти с точки зрения физической организации памяти

представляют собой набор физических страниц, которые хаотично располагаются в оперативной памяти. Для описания этих данных используется специальная структура, которая называется Memory Descriptor List или, сокращённо, MDL.

Некоторые устройства в составе современных компьютеров способны самостоятельно выполнять разбор MDL. К таким устройствам относятся, например, контроллеры Serial ATA, управляющие накопителями на жёстких дисках. Рассмотрим, из каких полей состоит эта структура.

В состав структуры MDL входят индексы всех физических страниц памяти, в которых находятся данные, поэтому структура MDL не имеет фиксированной длины. Порядок следования индексов физических страниц обязан повторять порядок следования соответствующих им виртуальных страниц памяти в составе буфера данных. Кроме того, в состав структуры MDL должны входить поле размера буфера и поле смещения первого байта буфера от начала страницы памяти. В состав структуры MDL входят и другие поля, но для полного описания данных достаточно уже перечисленных полей.

Работа с виртуальной памятью

В операционной системе Windows используется, так называемая, плоская модель памяти (flat – простыня) [2]. Суть её заключается в следующем. Каждому 32-разрядному процессу в системе отводится один сегмент равный 4 гигабайтам виртуальной памяти. В этих 4 Гб помещается всё видимое пространство памяти. Сегментные регистры кода, данных, стека указывают на это одно и то же виртуальное пространство. Кроме того, в этом же виртуальном пространстве находится и 2 Гб ядро операционной системы (в старших адресах). Внутри flat-модели работа с памятью осуществляется с помощью страничной организации памяти. Размер страницы – 4 Кб. Отсюда и название плоская, (т.е двумерная – (P, i)). Но её можно также трактовать как линейную.

В младших адресах процесса располагается, так называемая, NULL область. Её размер 64 Кб. Затем идет защищенная область программы до 2 Гб без 64 Кб. Заблокированная область в 64 Кб сделана для простоты защиты ядра операционной системы. В область программы грузятся и файлы, проецируемые в память.

Выше 2 Гб находится ядро операционной системы, либо ядро работает в своем собственном адресном пространстве, недоступном прикладным программам и вызывается из них через специальный шлюз INT 2Eh. Код функции задается содержимым регистров процессора. Необходимость выделения кода ядра операционной системы в отдельное адресное

пространство возникла после появления атаки на данные процессоров от компании Intel, получившей название Meltdown. Такая организация адресного пространства ядра, несомненно уменьшает производительность операционной системы.

Все сказанное, в целом, верно и для 64-битных процессов Windows, с той лишь разницей, что размер области адресного пространства, выделенной под прикладной процесс, равно 1 Тб.

Управление памятью в ОС Windows

Вся виртуальная память в ОС Windows делится на три класса: зарезервированная, выделенная и доступная. Каждая страница может принадлежать в один момент времени к одному классу.

Доступная память – это вся память процесса (4 Гб), то есть та память, которую он может адресовать.

Зарезервированная память – это область адресного пространства, затребованная процессом, но не используемая для хранения данных. При резервировании памяти вы не выделяете память, а выделяете участок адресного пространства как единицу, с которой вы намерены в дальнейшем работать. Физическая память при этом не выделяется. Эта память может быть затребована в любой момент времени.

Наконец, выделенная память, – это память, находящаяся в работе. Она может быть на диске или в ОЗУ, но она хранит в себе какие-либо данные.

Резервирование и выделение памяти делаются с помощью функции VirtualAlloc. При выделении программист может указать любой размер выделяемой области, но ядро операционной системы обязательно выровняет его до границы следующих 64 кб.

```
LPVOID WINAPI VirtualAlloc(
  _In_opt_ LPVOID lpAddress,
  _In_     SIZE_T dwSize,
  _In_     DWORD  flAllocationType,
  _In_     DWORD  flProtect);
```

Освобождение памяти делается с помощью функции VirtualFree.

```
BOOL WINAPI VirtualFree(
  _In_ LPVOID lpAddress,
  _In_ SIZE_T dwSize,
  _In_ DWORD  dwFreeType);
```

Память под исполняемый код программ резервируется в самих файлах программ, а не в свопинге. При отсутствии свободной памяти код

программы в ОЗУ просто уничтожается, а затем, при необходимости, загружается из файла программы.

Пример резервирования памяти, (довольно надуманный) приведён ниже. Например, необходимо выделить массив данных равный 4 Мб памяти. Но заранее известно, что на самом деле программе потребуется лишь небольшая часть от этого массива. Эту память можно не выделять, а зарезервировать. Код, работающий с массивом помещается внутрь структурного обработчика исключений. При обращении к ячейке памяти внутри массива возникает исключение и с помощью функции VirtualAlloc происходит выделение участка памяти по адресу обращения.

```
typedef int Data[100][100][100];

int main(int argc, char* argv[]) {

      Data *pData;
      pData = (Data*)VirtualAlloc(0, sizeof(Data), MEM_RESERVE,
PAGE_READWRITE);
      int *pSomeData = &(*pData)[50][50][50];

      __try {
            *pSomeData = 1;
      }
      __except(VirtualAlloc(pSomeData, sizeof(int), MEM_COMMIT,
PAGE_READWRITE), -1)  {}

      printf("SomeData == %d", *pSomeData);

      VirtualFree(pData, sizeof(Data), MEM_DECOMMIT);
      return 0;
}
```

При освобождении памяти освобождаются все 64 Кб блоки, выделенные внутри массива зарезервированной памяти.

Кучи

Вся выделенная память в операционной системе Windows, как и в других операционных системах делится на три типа: код, стек и куча. Код создается отображением PE-сегмента кода на адресное пространство процесса. Стек используется для локальных переменных внутри функций и при необходимости растет блоками по 64 кб. Выделение памяти для стека происходит в контексте ядра операционной системы и для процесса происходит незаметно. В куче процесса находится динамически выделяемая память.

При создании процесса система создает в его адресном пространстве стандартную кучу процесса. Её зарезервированный размер – 1 Мб, а размер предварительно выделенной области равен 64 кб. Если выделенная память под кучу заканчивается, то к ней добавляется еще нужное количество блоков размером по 64 Кб. Получить описатель кучи можно с помощью функции GetProcessHeap:

```
HANDLE GetProcessHeap();
```

При компиляции программы необходимо указать, будет ли использоваться доступ к куче одной или несколькими нитями. Для этого служит ключ _MT. При использовании этого ключа компилятор организует вызовы создания основной кучи процесса, выделения и освобождения памяти через критическую секцию.

Память выделяется из кучи с использованием вызовов HeapAlloc, HeapReAlloc, а освобождается с помощью функции HeapFree. Алгоритм работы функции HeapRealloc подобен алгоритму работы функции realloc из состава стандартной библиотеки POSIX, то есть производится поиск области памяти большего размера, данные копируются в эту область памяти, а затем исходная область памяти освобождается.

```
LPVOID HeapAlloc(
  HANDLE hHeap,
  DWORD  dwFlags,
  SIZE_T dwBytes);

LPVOID HeapReAlloc(
  HANDLE  hHeap,
  DWORD   dwFlags,
  LPVOID  lpMem,
  SIZE_T  dwBytes);

BOOL HeapFree(
  HANDLE  hHeap,
  DWORD   dwFlags,
  LPVOID  lpMem
);
```

Флаг HEAP_NO_SERIALIZE во всех перечисленных функциях отключает критическую секцию, ускоряя работу приложения и перекладывая ответственность за синхронизацию нитей на программиста.

Помимо основной кучи процесса, возможно создание дополнительных куч с помощью функции HeapCreate:

```
HANDLE HeapCreate(
  DWORD  flOptions,
  SIZE_T dwInitialSize,
  SIZE_T dwMaximumSize);
```

Первый аргумент flOptions функции HeapCreate, это опции создания кучи, а именно: HEAP_GENERATE_EXCEPTIONS – если памяти нет, то для функции HeapAlloc генерировать исключение вместо возврата NULL, HEAP_NO_SERIALIZE – отключает для этой кучи критическую секцию. Второй аргумент содержит в себе начальный размер создаваемой кучи, а третий – максимальный размер. Если указать максимальный размер кучи, то она не сможет расти дальше этого размера.

Удаляется куча через вызов HeapDestroy.

```
BOOL HeapDestroy(
  HANDLE hHeap);
```

Использование нескольких куч помогает упростить и ускорить работу с памятью. Например, если имеется четко определенная задача, после выполнения которой данные, использованные для работы, становятся не нужны, то имеет смысл создать для этой задачи новую кучу, а по окончании работы удалить кучу целиком, не утруждаясь освобождением отдельных элементов данных.

Физическая организация куч следующая: при создании и росте кучи для выделения памяти используется функция VirtualAlloc. Функции HeapAlloc и HeapFree организованы внутри библиотек API и, если памяти достаточно, то обращения к контексту ядра не происходит. Мелкие выделения памяти идут в контексте самого процесса, ускоряя выполнение программы.

Для проверки наличия выделенной памяти по заданному адресу имеется группа функций проверки, которые возвращают результат проверки в виде булевского значения: IsBadCodePtr, IsBadReadPtr, IsBadStringPtr, IsBadWritePtr

Стандартные библиотечные POSIX функции malloc и free в операционной системе Windows реализованы как надстройка к функциям HeapAlloc и HeapFree соответственно.

Работа с кучей в операционной системе UNIX

Функции malloc, calloc, realloc и free являются вызовами, описанными в стандартной библиотеке UNIX для языка C – библиотеке libc [9]. Память из ядра выделяется порциями по размеру страницы. Функция getpagesize

дает размер страницы памяти на текущем процессоре. Как уже упоминалось, для процессоров семейства Intel x86 он равен 4 Кб.

Сегмент данных в операционной системе UNIX не пересекается с сегментом кода. Его максимальный размер задается наряду с другими характеристиками процесса функцией setrlimit. Посмотреть максимальный размер можно с помощью функции getrlimit. Текущий размер сегмента данных не равен максимальному. Его можно задать функцией brk.

Как именно поведет себя система при ошибках, связанных повторным освобождением одной и той же памяти и т. д. определяется переменной среды MALLOC_CHECK_. Если она равна 0, то ошибки по памяти просто игнорируются. Если MALLOC_CHECK_ == 1, то диагностические ошибки выводятся в поток stderr, если она равна 2, то после ошибки немедленно вызывается функция abort, приводящая к уничтожению процесса.

Динамически загружаемые библиотеки

Динамически загружаемые библиотеки *(Dynamic Link Library – DLL) это бинарные модули, предназначенные для предоставления унифицированного интерфейса к данным и обработчикам данных на этапе исполнения программы.*

Динамически загружаемые библиотеки в операционной системе Windows представляют собой бинарные файлы формата PE. Они предназначены для загрузки и выгрузки по требованию в процессе выполнения программы. Библиотеки связываются с программой и между собой с помощью специальных таблиц экспорта и импорта, находящихся в составе файлов формата PE [2].

Просмотреть содержимое таблиц экспорта и импорта можно с помощью утилиты dumpbin.exe из состава MS Visual Studio. Ниже приведены примеры использования этой утилиты для библиотеки kernel32.dll:

```
dumpbin.exe kernel32.dll \exports > kernel32.exports
dumpbin.exe kernel32.dll \imports > kernel32.imports
```

Таблица импорта содержит запрашиваемые ресурсы, таблица экспорта – предоставляемые ресурсы библиотеки.

Другой способ исследования взаимосвязей между библиотеками – использование утилиты Dependency Walker (depends.exe) также из состава MS Visual Studio, которая предоставляет информацию в графическом режиме.

Для создания динамически загружаемой библиотеки необходимо указать при создании проекта его тип – DLL. У динамически загружаемой библиотеки отсутствует функция WinMain. Вместо нее используется функция DllMain

```
BOOL WINAPI DllMain(
  _In_ HINSTANCE hinstDLL,
  _In_ DWORD     fdwReason,
  _In_ LPVOID    lpvReserved);
```

Функция DllMain вызывается операционной системой в четырех случаях:

1. при загрузке библиотеки процессом (отображении в виртуальное адресное пространство процесса) – DLL_PROCESS_ATTACH;
2. при создании новой нити – DLL_THREAD _ATTACH;
3. при завершении созданной нити – DLL_THREAD_DETACH;

4. при завершении процесса или при выгрузке библиотеки – DLL_PROCESS_DETACH.

Аргумент fdwReason содержит в себе причину вызова функции DllMain. Всего в процессе работы программы функции DllMain из загруженных ей библиотек могут вызываться десятки раз при возникновении соответствующей причины вызова.

Аргумент lpvReserved имеет смысл в двух случаях: При DLL_PROCESS_ATTACH параметр равен нулю для динамической загрузки и не ноль для загрузки при старте процесса. При DLL_PROCESS_DETACH параметр равен нулю для динамической выгрузки и не ноль для выгрузки в связи с завершением процесса.

Аргумент hinstDLL представляет собой адрес начала содержимого файла библиотеки в адресном пространстве процесса.

Любую функцию, переменную или класс библиотеки можно сделать экспортируемыми, т.е. подключаемыми извне с помощью таблицы экспорта. Чтобы адрес ресурса был помещен в таблицу экспорта, необходимо указать непосредственно перед определением ресурса ключевые слова `__declspec(dllexport)`. Примеры экспортирования ресурсов:

```
__declspec(dllexport) int i;
__declspec(dllexport) void func();
class __declspec(dllexport) Class;
```

DLL могут загружаться процессом при старте программы на этапе её загрузки или явно с помощью функции LoadLibrary.

```
HMODULE LoadLibrary(
  LPCSTR lpLibFileName
);
```

Единственный аргумент функции LoadLibrary – путь до файла библиотеки.

После компиляции библиотеки компоновщик создает два файла для каждой динамически загружаемой библиотеки – с расширениями .lib и .dll. Файл с расширением .dll собственно и является динамически загружаемой библиотекой.

Файл с расширением .lib необходимо подключить при компоновке программы, использующей связывание с библиотекой на этапе загрузки. Это делается в опциях проекта для компоновщика. Программа может импортировать ресурсы из DLL с помощью ключевых слов __declspec(dllimport). Примеры импортирования ресурсов:

```
__declspec( dllimport ) int i;
```

```
__declspec(dllimport) void func();
class __declspec(dllimport) Class;
```

При запуске программы файл библиотеки с расширением .dll должен находиться в одном каталоге с файлом программы или быть доступен по путям поиска.

Для простоты работы с динамическими библиотеками рекомендуется использовать общие заголовочные файлы, в которых определения даны с использованием директив предкомпиляции, например, как показано ниже:

```
#ifndef DLL
#define DLL_LINK __declspec(dllimport)
#else
#define DLL_LINK __declspec(dllexport)
#endif

DLL_LINK int Var;
```

В свойствах проекта библиотеки должна быть определена директива предкомпиляции DLL. В этом случае переменная Var из примера будет экспортироваться. Если такая директива отсутствует, то переменная Var будет импортироваться.

Как уже упоминалось, второй способ загрузки динамической библиотеки основывается на вызове функции LoadLibrary. Результатом вызова этой функции является загрузка библиотеки в виртуальное адресное пространство процесса. Процессу становится доступен дескриптор (HMODULE) библиотеки. Для получения адреса ресурса, находящегося в библиотеке, необходимо вызвать специальную функцию GetProcAddress.

```
FARPROC GetProcAddress(
  HMODULE hModule,
  LPCSTR  lpProcName
);
```

Параметр lpProcName должен содержать указатель на верное имя ресурса или его номер. Поскольку в библиотеке может находиться не более 65535 ресурсов (от 1 до 65535), то не требуется дополнительной функции получения ресурса по номеру. Функция анализирует адрес, и если он попадает в NULL область, то это номер ресурса, иначе это указатель на строку с именем.

Для выгрузки DLL из виртуального адресного пространства процесса используется функция `FreeLibrary`. Напомним, что перед тем, как библиотека будет выгружена из адресного пространства процесса, произойдёт вызов её функции DllMain с соответствующей причиной вызова.

```
BOOL FreeLibrary(
  HMODULE hLibModule
);
```

В параметре hLibModule следует указать дескриптор библиотеки, полученный с помощью функции LoadLibrary.

Следует отметить, что имя ресурса в таблице экспорта зачастую отличается от имени ресурса, определенного в исходном коде библиотеки. Это следствие, так называемого, **декорирования имен**. Например, функция

```
void GetState();
```

получает название "?GetState@@YAXXZ". Декорирование используется в C++ для того, чтобы компилятор мог сравнить типы аргументов и возвращаемых значений, поскольку язык C++ допускает одинаковые имена функций с различными аргументами. Чтобы указать имя без декорирования, можно имя экспортируемого ресурса писать в языке Си. Например

```
extern “C” void GetState();
```

При такой декларации функции, она будет экспортироваться с именем "_GetState". Еще один способ указания имени экспортируемого ресурса без использования декорирования, – это использование файлов с расширением def. Такой файл можно просто добавить в проект. Примерное содержимое файла для экспорта:

```
IBRARY   MY_LIB
EXPORTS
   GetState   @1
```

Символ @ показывает, что далее будет идти номер ресурса. Это необязательный элемент файла def, его можно и не писать.

Системные перехватчики

Системные перехватчики – это специальные функции, вызываемые операционной системой Windows при возникновении условия перехвата. В основном системные перехватчики работают с оконными сообщениями. Системный перехватчик можно установить на выбранную нить или на все нити текущего рабочего стола. В общем случае системные перехватчики являются небезопасным инструментом, способным нарушить нормальное функционирование оконного процесса в операционной системе Windows, однако их использование предоставляет программисту некоторые возможности по управлению визуальным интерфейсом операционной системы, недоступные для реализации другими способами.

Если системный перехватчик должен перехватывать не только сообщения текущего процесса, но и сообщения других процессов этого рабочего стола, то функция перехвата должна находиться в динамически загружаемой библиотеке. Адрес этой функции и адрес самой библиотеки необходимо получить с помощью динамической загрузки DLL посредством функции LoadLibrary. После установления перехватчика эта библиотека будет внедрена во все процессы, для которых требуется активизировать функцию перехвата, причём эта функция будет вызываться в контексте именно той нити процесса, которая работает с оконным сообщением.

Для создания системного перехватчика используется функция SetWindowsHookEx:

```
HHOOK SetWindowsHookEx(
  int idHook, // тип перехватчика
  HOOKPROC lpfn, // функция перехвата
  HINSTANCE hMod, // DLL, в которой содержится функция
перехвата или NULL если не нужно внедряться в другой процесс
  DWORD dwThreadId //нить или 0 для всех нитей этого рабочего
стола
);
```

Все функции перехвата имеют следующий вид:

```
LRESULT CALLBACK <Имя>(
  int     nCode,
  WPARAM  wParam,
  LPARAM  lParam);
```

Значения параметров функций различаются для разных видов перехватчиков.

Некоторые состояния процесса можно получить несколькими способами с помощью разных видов перехватчиков. Выбор конкретного

перехватчика производится исходя из требований задачи. Виды перехватчиков приведены в таблице 6. Виды перехватчиков указаны символом «T» для перехватчиков, действующих по отношению к текущей нити и символом «D» для перехватчиков, действующих для всего рабочего стола.

Таблица 6

Виды системных перехватчиков в операционной системе Windows

Перехватчик	Вид	Функция	Описание
WH_CALLWNDPROC	TD	CallWndProc	Проверка сообщений перед посылкой оконной процедуре
WH_CALLWNDPROCRET	TD	CallWndRetProc	Проверка сообщений после обработки оконной процедурой
WH_CBT	TD	CBTProc	Проверка состояния окна (создание, уничтожение, получение фокуса, активизация, минимизация и т.д.)
WH_DEBUG	TD	DebugProc	Проверка других перехватчиков
WH_FOREGROUNDIDLE	TD	ForegroundIdleProc	Выполняется, когда компьютер переходит в состояние простоя
WH_GETMESSAGE	TD	GetMsgProc	Проверка всех сообщений в момент вызова функций GetMessage и PeekMessage
WH_JOURNALRECORD	D	JournalRecordProc	Запись всех оконных сообщений
WH_JOURNALPLAYBACK	D	JournalPlaybackProc	Проигрывание записанных оконных сообщений

WH_KEYBOARD	TD	KeyboardProc	Проверка символов, введённых с клавиатуры
WH_KEYBOARD_LL	D	LowLevelKeyboardProc	Проверка нажатий клавиш клавиатуры
WH_MOUSE	TD	MouseProc	Проверка логических действий мышью
WH_MOUSE_LL	D	LowLevelMouseProc	Проверка движений мышью и нажатий клавиш мыши
WH_MSGFILTER	TD	MessageProc	Проверка оконных сообщений для диалоговых окон, меню и выпадающих списков в пользовательской области экрана
WH_SHELL	TD	ShellProc	Проверка действий оболочки (переключение языка, активизация окна, вывод списка задач и т.д.)
WH_SYSMSGFILTER	D	SysMsgProc	Проверка оконных сообщений для диалоговых окон, меню и выпадающих списков в системной области экрана

Для отключения системного перехватчика используется функция UnhookWindowsHookEx:

```
BOOL UnhookWindowsHookEx(
  HHOOK  hhk);
```

Результатом действия функции UnhookWindowsHookEx будет полное отключение работы системного перехватчика, однако динамически загружаемая библиотека, которая была загружена в адресные пространства процессов в то время, пока перехватчик ещё действовал, так и останется загруженной.

Службы в ОС Windows

Общие сведения

Службы *– это специальные программы в операционной системе Windows, предназначенные для выполнения различных дополнительных функций этой операционной системы*. Например, удалённый вызов процедур, сетевая печать, терминальный доступ и т.д.

Работа служб обеспечивается взаимодействием компонентов трёх типов (рис. 38) [12].

- Диспетчер управления службами (Service Control Manager, SCM)
- Приложение-служба
- Программа управления службой.

Диспетчер управления службами при работе операционной системы представляется как программа Services.exe. Он автоматически запускается при запуске системы. Диспетчер управления службами имеет привилегии системы и обеспечивает интерфейс взаимодействия программ управления службами и программ-служб.

Служба – это прикладная программа, вызывающая специальные функции взаимодействия с диспетчером управления службами. По результатам работы этих функций работа службы может быть запущена, остановлена, приостановлена и повторно возобновлена. Помимо этого, служба информирует SCM о своём текущем состоянии.

Программа управления службой обычно имеет интерфейс с пользователем, с помощью которого можно управлять работой службы. Программа управления службой также взаимодействует с диспетчером управления службами. Эта программа может находиться не только на той же самой машине, где работает служба, но и на другой, связанной с первой через локальную сеть. При этом связь с другим компьютером будет производиться средствами операционной системы посредством специальной службы удалённого вызова процедур (RPC).

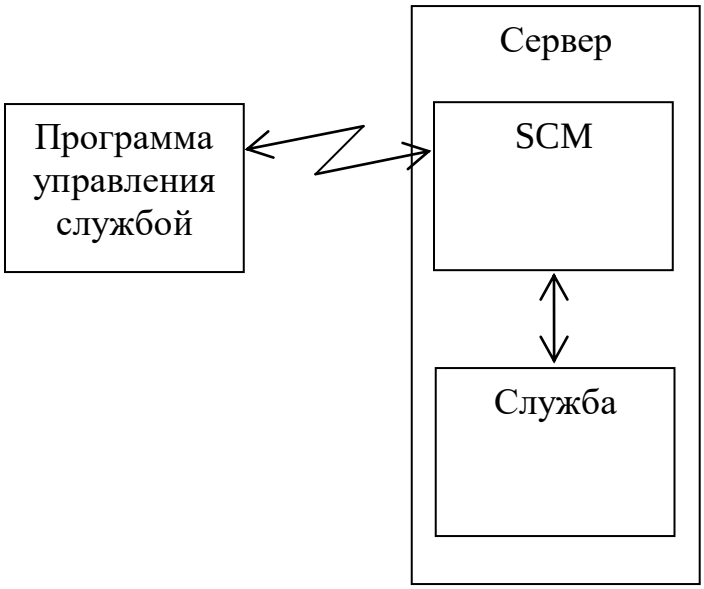

Рис. 38. Взаимодействие компонентов механизма служб.

Служба может работать с правами системы (пользователь SYSTEM) или с правами заданного конкретного пользователя.

У службы может быть 2 вида запуска автоматический или ручной, а также запуск службы может быть запрещён. При автоматическом запуске служба запускается при старте операционной системы Windows. При ручном запуске служба запускается по команде от программы управления службой.

Служба может запускаться довольно длительное время (до нескольких секунд). Поэтому в программном интерфейсе диспетчера управления службами предусмотрено время ожидания запуска службы. Служба может также и не запуститься по какой-либо причине. В стандартной программе управления службами, поставляемой вместе с операционной системой Windows можно выбрать тип реакции системы на сбой при запуске службы. Отдельно оговорены первая попытка запуска, вторая попытка запуска и все последующие попытки запуска службы. Реакция может быть от перезагрузки компьютера, до простого сообщения в системный журнал.

Программа управления службой

Программа управления службой может инсталлировать, деинсталлировать не только службу, но и драйвер. Она может управлять работой службы. Максимальные права такая программа получает, если текущий пользователь – администратор. В другом случае её права существенно уменьшаются.

Для доступа к диспетчеру из программы управления службой используется функция OpenSCManager:

```
SC_HANDLE OpenSCManager(
  LPCSTR lpMachineName,
  LPCSTR lpDatabaseName,
  DWORD  dwDesiredAccess);
```

Для доступа к диспетчеру текущего компьютера первые два аргумента функции могут иметь значение NULL. Третий аргумент – запрашиваемые права доступа. Функция возвращает дескриптор на диспетчера SCM. Когда он станет ненужным, его следует закрыть с помощью функции CloseServiceHandle:

```
BOOL CloseServiceHandle(
  SC_HANDLE hSCObject);
```

С помощью функции CloseServiceHandle следует также закрывать имеющиеся открытые дескрипторы служб.

Для создания новой службы используется функция CreateService:

```
SC_HANDLE CreateService(
  SC_HANDLE hSCManager,
  LPCSTR    lpServiceName,
  LPCSTR    lpDisplayName,
  DWORD     dwDesiredAccess,
  DWORD     dwServiceType,
  DWORD     dwStartType,
  DWORD     dwErrorControl,
  LPCSTR    lpBinaryPathName,
  LPCSTR    lpLoadOrderGroup,
  LPDWORD   lpdwTagId,
  LPCSTR    lpDependencies,
  LPCSTR    lpServiceStartName,
  LPCSTR    lpPassword);
```

У этой функции довольно много аргументов. Рассмотрим, эти аргументы, поскольку они в значительной степени определяют характеристики службы.

hSCManager – это описатель на SCM, полученный с помощью функции OpenSCManager.

lpServiceName – это имя службы, содержащее не более 256 символов.

lpDisplayName – это, так называемое, отображаемое имя службы, которое обычно состоит из нескольких слов, характеризующих предназначение службы, также не более 256 символов.

dwDesiredAccess – права доступа к службе по новому описателю. Максимальные права доступа кодируются константой

SC_MANAGER_ALL_ACCESS. Если нужно только установить и не запускать службу, то в этом аргументе можно передать 0.

dwServiceType – тип службы. Может принимать одно из значений, перечисленных в таблице 7.

Таблица 7

Тип службы операционной системы Windows

Тип	Означает
SERVICE_FILE_SYSTEM_DRIVER	Драйвер файловой системы.
SERVICE_KERNEL_DRIVER	Драйвер.
SERVICE_WIN32_OWN_PROCESS	Одна служба в этом процессе
SERVICE_WIN32_SHARE_PROCESS	Несколько служб в этом процессе

Как видно из таблицы, с помощью функции CreateService можно устанавливать не только службы, но и драйверы операционной системы Windows. Если службе нужно выводить сообщения на экран локального терминала, то нужно в этом аргументе по ИЛИ добавить флаг SERVICE_INTERACTIVE_PROCESS.

dwStartType – тип запуска службы. Возможны следующие типы запуска (табл. 8):

Таблица 8

Тип запуска службы операционной системы Windows

Тип	Означает
SERVICE_AUTO_START	Автоматический запуск службы
SERVICE_BOOT_START	Драйвер, запускающийся на раннем этапе запуска операционной системы
SERVICE_DEMAND_START	Запуск службы вручную
SERVICE_DISABLED	Запуск службы запрещён
SERVICE_SYSTEM_START	Драйвер, запускающийся после запуска операционной системы

dwErrorControl – какова будет реакция системы на ошибку запуска службы. Выбирается одно из значений, приведённых в таблице 9.

Таблица 9

Виды реакции системы на ошибку запуска службы

Значение	Означает
SERVICE_ERROR_IGNORE	Игнорировать ошибки запуска службы.
SERVICE_ERROR_NORMAL	Вывести сообщение и игнорировать ошибку запуска.
SERVICE_ERROR_SEVERE	Игнорировать, если это последняя известная хорошая конфигурация операционной системы.
SERVICE_ERROR_CRITICAL	Перегрузиться в последнюю известную хорошую конфигурацию. Иначе аварийно завершить работу операционной системы.

lpBinaryPathName – путь до бинарного файла службы. Желательно в этом параметре указывать полный путь до файла, во избежание возможных ошибок запуска.

lpLoadOrderGroup – группа запуска службы. Службы могут объединяться в группы запуска. В простейшем случае в этот аргумент функции рекомендуется передавать значение NULL. В этом случае служба будет запущена в самом конце последовательности запуска служб.

lpdwTagId – этот параметр применяется в основном для установки драйверов. Рекомендуется в этот аргумент функции передать NULL.

lpDependencies – зависимости службы. Этот аргумент представляет собой, так называемую, мультистроку. Это массив строк, заканчивающийся пустой строкой. Каждая строка в массиве есть имя службы, которая должна запуститься до этой службы. По умолчанию в этот аргумент можно передать NULL.

lpServiceStartName – имя пользователя, с правами которого должна быть запущена служба. Если в этом аргументе передать значение NULL, то служба будет запущена с правами локальной системы.

lpPassword – пароль пользователя. Этот аргумент необходимо указывать только в том случае, если пользователь службы отличается от локальной системы.

Функция OpenService используется для открытия уже имеющейся службы. Аргументы этой функции идентичны соответствующим аргументам функции CreateService.

```
SC_HANDLE OpenService(
  SC_HANDLE hSCManager,
  LPCSTR    lpServiceName,
  DWORD     dwDesiredAccess);
```

Функции создания и открытия службы возвращают описатель на службу. Этим описателем программа управления службой может воспользоваться, для контроля работы службы. Одно из возможных действий – удаление службы с помощью функции DeleteService.

```
BOOL DeleteService(
  SC_HANDLE hService
);
```

Необходимо отметить, что эта функция только удаляет информацию о службе из операционной системы, но не удаляет бинарный исполняемый файл службы.

Запуск установленной службы производится с помощью функции StartService:

```
BOOL StartService(
  SC_HANDLE hService,
  DWORD     dwNumServiceArgs,
  LPCSTR    *lpServiceArgVectors);
```

Первый аргумент этой функции, – описатель на службу, второй аргумент содержит количество параметров запуска, а третий – это указатель на строки аргументов запуска службы. Параметры командной строки при запуске будут переданы без изменений в основную функцию службы. Поскольку в одном процессе операционной системы Windows может быть несколько служб, параметры командной строки передаются именно таким способом не процессу, а службе внутри процесса.

Управление службой производится с помощью функции ControlService:

```
BOOL ControlService(
  SC_HANDLE        hService,
  DWORD            dwControl,
  LPSERVICE_STATUS lpServiceStatus);
```

Аргумент hService – описатель на службу. Аргумент dwControl представляет собой команду службе. Среди возможных команд службе следует отметить следующие (табл. 10):

Таблица 10

Некоторые виды команд службе операционной системы Windows

Команда	Означает
SERVICE_CONTROL_CONTINUE	Возобновить работу службы
SERVICE_CONTROL_INTERROGATE	Получить текущее состояние службы
SERVICE_CONTROL_PAUSE	Приостановить работу службы
SERVICE_CONTROL_STOP	Остановить службу
128-255	Команды, определённые пользователем

Параметр lpServiceStatus это указатель на экземпляр структуры SERVICE_STATUS, который заполняется самой службой посредством диспетчера управления службами.

Структура SERVICE_STATUS определена следующим образом:

```
typedef struct _SERVICE_STATUS {
  DWORD dwServiceType;
  DWORD dwCurrentState;
  DWORD dwControlsAccepted;
  DWORD dwWin32ExitCode;
  DWORD dwServiceSpecificExitCode;
  DWORD dwCheckPoint;
  DWORD dwWaitHint;
} SERVICE_STATUS, *LPSERVICE_STATUS;
```

dwServiceType – тот же, что и в функции CreateService

dwCurrentState – текущее состояние службы. Служба заполняет это поле одним из значений, определяющих её состояние. Перечень значений приведён в таблице 11.

Таблица 11

Возможные состояния службы операционной системы Windows

Значение	Означает
SERVICE_CONTINUE_PENDING	Идёт процесс возобновления работы службы
SERVICE_PAUSE_PENDING	Идёт процесс приостанова службы
SERVICE_PAUSED	Служба приостановлена

SERVICE_RUNNING	Служба работает
SERVICE_START_PENDING	Идёт процесс старта службы
SERVICE_STOP_PENDING	Идёт процесс останова службы
SERVICE_STOPPED	Служба остановлена

dwControlsAccepted – управляющие команды, распознаваемые службой. С помощью управляющих команд программа управления службой может останавливать службу, и приостанавливать её работу, но служба должна задекларировать поддержку команд управления. Перечень возможных команд, объединяемых по ИЛИ, приведён в таблице 12.

Таблица 12

Управляющие команды, распознаваемые службой операционной системы Windows

Название	Описание
SERVICE_ACCEPT_STOP	Служба способна к обработке команды останова
SERVICE_ACCEPT_PAUSE_CONTINUE	Служба способна к обработке команд приостанова и возобновления работы
SERVICE_ACCEPT_SHUTDOWN	Служба способна к обработке команды завершения работы системы
SERVICE_ACCEPT_PARAMCHANGE	Служба способна к обработке команды изменения параметров
SERVICE_ACCEPT_SESSIONCHANGE	Служба способна к обработке команды изменения сессии пользователя.
SERVICE_ACCEPT_PRESHUTDOWN	Служба способна к обработке команды предварительного оповещения о завершении работы системы.
SERVICE_ACCEPT_TIMECHANGE	Служба способна к обработке команды изменения системного времени.

dwWin32ExitCode – если в процессе старта или останова работы службы произошла ошибка, то служба заполняет это поле значением ошибки. В процессе нормальной работы службы это поле должно быть равно NO_ERROR. Если ошибка работы службы является специфичной только для этой службы, то в этом поле служба устанавливает значение ERROR_SERVICE_SPECIFIC_ERROR, что означает, что код ошибки необходимо получить из следующего поля структуры.

dwServiceSpecificExitCode – код ошибки, специфичный для конкретной службы.

dwCheckPoint – поле проверки работоспособности службы. Во время переходных процессов (идёт останов, приостанов или возобновление работы) служба периодически инкрементирует это значение, чтобы программа управления службой знала, что служба продолжает нормально функционировать.

dwWaitHint – предлагаемое службой время ожидания перед следующим опросом работоспособности службы. Для переходных состояний службы это поле содержит время в миллисекундах, через которое служба планирует обновить своё состояние (инкрементировать поле dwCheckPoint и прописать новое значение предлагаемого времени ожидания dwWaitHint). Если через указанное время таких действий не происходит, то программа управления службой делает вывод, что служба перестала функционировать.

Получить текущее состояние службы можно с помощью функции QueryServiceStatus:

```
BOOL QueryServiceStatus(
  SC_HANDLE        hService,
  LPSERVICE_STATUS lpServiceStatus);
```

Все параметры этой функции уже знакомы. Её нужно вызывать до тех пор, пока служба находится в переходном состоянии (PENDING) или до возникновения ошибки или до тех пор, пока служба не перестанет нормально функционировать (не перестанет инкрементировать поле dwCheckPoint).

Расширенное описание службы можно задать с помощью функции ChangeServiceConfig2:

```
BOOL ChangeServiceConfig2(
  SC_HANDLE hService,
  DWORD     dwInfoLevel,
  LPVOID    lpInfo);
```

Первый аргумент функции – описатель на службу, второй аргумент функции задаёт тип изменяемой информации о службе, а третий аргумент представляет собой адрес буфера с информацией о службе. Структура содержимого буфера различается в зависимости от типа изменяемой информации. Для изменения описания службы необходимо, чтобы второй параметр содержал значение SERVICE_CONFIG_DESCRIPTION, тогда третий параметр должен указывать на структуру SERVICE_DESCRIPTION.

```
typedef struct _SERVICE_DESCRIPTION {
  LPSTR lpDescription;
} SERVICE_DESCRIPTION, *LPSERVICE_DESCRIPTION;
```

Поле lpDescription есть указатель на строку с описанием службы.

Перечислим требуемую последовательность вызова функций для инсталляции и запуска службы: OpenSCManager, CreateService, StartService, CloseServiceHandle (для службы), CloseServiceHandle (для SCM).

Для останова и деинсталляции службы требуется выполнить другую последовательность функций: OpenSCManager, OpenService, ControlService (останов), QueryServiceStatus (проверка останова), DeleteService, CloseServiceHandle (для службы), CloseServiceHandle (для SCM).

Если программа управления службой будет деинсталлировать службу, не останавливая её, то она будет полностью деинсталлирована только после рестарта компьютера. В этом случае для удаления исполняемого файла службы (а также других файлов) можно воспользоваться функцией MoveFileEx, которая позволяет удалять файл после перезагрузки.

```
BOOL MoveFileEx(
  LPCSTR lpExistingFileName,
  LPCSTR lpNewFileName,
  DWORD  dwFlags);
```

Первый аргумент функции при таком её применении должен указывать на удаляемый файл, во второй параметр необходимо передать значение NULL, а третий аргумент должен иметь значение MOVEFILE_DELAY_UNTIL_REBOOT.

Создание процесса-службы

Программу службы необходимо создавать как обычную Win32 программу, главной функцией которой является функция WinMain. В одной такой программе может содержаться несколько служб. Каждая из таких служб будет взаимодействовать с диспетчером управления службами самостоятельно.

Программа службы отличается от обычной прикладной программы операционной системы Windows наличием специальных требований. Перечень требований приведён далее:

1. В функции WinMain необходимо как можно быстрее вызвать функцию регистрации службы StartServiceCtrlDispatcher. Эта функция выполняет предварительную инициализацию служб в текущем процессе, предоставляет SCM перечисление всех служб процесса с помощью таблицы.
2. С каждой строкой таблицы связывается основная функция конкретной службы. Её можно условно назвать ServiceMain. У функции ServiceMain имеются аргументы запуска, которые передаются ей через регистрацию в программе управления службой.
3. Внутри функции ServiceMain необходимо вызвать функцию RegisterServiceCtrlHandler. С помощью этой функции служба регистрирует в системе обработчик запросов от SCM. Назовём его условно HandlerFunction.

Функция StartServiceCtrlDispatcher содержит только один аргумент – указатель на константную структуру типа SERVICE_TABLE_ENTRY, описывающую все службы текущего процесса.

```
BOOL StartServiceCtrlDispatcher(
  const SERVICE_TABLE_ENTRY *lpServiceStartTable);

typedef struct _SERVICE_TABLE_ENTRY {
  LPTSTR                   lpServiceName;
  LPSERVICE_MAIN_FUNCTION  lpServiceProc;
} SERVICE_TABLE_ENTRY, *LPSERVICE_TABLE_ENTRY;
```

Поле lpServiceName содержит в себе имя службы. Поле lpServiceProc содержит в себе указатель на основную функцию службы (условный ServiceMain).

```
VOID WINAPI ServiceMain(
  _In_ DWORD  dwArgc,
  _In_ LPTSTR *lpszArgv);
```

Функция ServiceMain имеет два параметра. Первый параметр – количество аргументов службы, второй параметр – указатель на массив аргументов. Эти аргументы поступают в основную функцию службы непосредственно от программы управления службой как второй и третий параметры функции StartService.

Функция ServiceMain должна содержать в себе сам функционал службы. Но перед эти она обязана вызвать функцию RegisterServiceCtrlHandler.

```
SERVICE_STATUS_HANDLE RegisterServiceCtrlHandlerA(
  LPCSTR             lpServiceName,
  LPHANDLER_FUNCTION lpHandlerProc);
```

Первый аргумент функции – имя службы, второй аргумент – адрес функции обработки команд от диспетчера управления службами HandlerFunction. Взаимодействовать с диспетчером управления службой будет функция HandlerFunction.

```
void HandlerFunction(
  DWORD dwControl)
```

Функция HandlerFunction имеет один аргумент: код команды от диспетчера управления службами. Некоторые из команд перечислены в таблице 13:

Таблица 13

Некоторые виды команд службе от диспетчера управления службами

Название	Описание
SERVICE_CONTROL_STOP	Команда останова службы
SERVICE_CONTROL_PAUSE	Команда приостанова службы
SERVICE_CONTROL_CONTINUE	Команда возобновления работы службы
SERVICE_CONTROL_INTERROGATE	Команда запроса состояния службы
SERVICE_CONTROL_PRESHUTDOWN	Команда оповещения о завершении работы системы
SERVICE_CONTROL_SHUTDOWN	Команда завершения работы системы
SERVICE_CONTROL_PARAMCHANGE	Команда изменения параметров службы
SERVICE_CONTROL_TIMECHANGE	Команда изменения системного времени
SERVICE_CONTROL_SESSIONCHANGE	Команда смены сессии пользователя

Если служба не обрабатывает какой-либо тип команды от диспетчера, то она может эти команды просто игнорировать.

В ответ на команду со стороны SCM служба при необходимости меняет своё состояние и выставляет изменение своего статуса с помощью функции SetServiceStatus. На вход этой функции поступает адрес переменной структуры SERVICE_STATUS.

Таким образом служба реагирует на команды со стороны SCM. Обработка команд от SCM происходит в специальной нити, связанной напрямую с текущей службой.

Служба имеет возможность информировать менеджер служб о командах, которые она может обрабатывать, как уже отмечалось в разделе, посвященном программе управления службами. Для этого в соответствующей переменной структуры SERVICE_STATUS необходимо выставить соответствующие биты, информирующие о принимаемых службой командах.

Как правило, основная нить службы выполняет действия, предусмотренные алгоритмом работы службы. Изменения в работе службы происходят после обработки команд со стороны менеджера служб в отдельной нити. Взаимодействие между нитями можно организовывать любыми возможными средствами, начиная от глобальных переменных и заканчивая объектами синхронизации ядра операционной системы.

Литература

1. Гордеев А.В., Молчанов А.Ю. Системное программное обеспечение. – СПб.: Питер, 2001. – 736 с.
2. Дж. Рихтер. Windows для профессионалов. Создание эффективных WIN32-приложений с учетом специфики 64-разрядной версии Windows. Изд-ва: Питер, Русская Редакция, 2001 г., 752 стр. ISBN 5-272-00384-5, 1-57231-996-8
3. Эндрюс Г.Р. Основы многопоточного, параллельного и распределенного программирования.: Пер. с англ.— М.: Издательский дом «Вильямс», 2003. — 512 с.
4. Таненбаум Э., Бос Х. Современные операционные системы. 4-е изд. — СПб.: Питер, 2015. — 1120 с.
5. Робачевский А.М. Операционная система UNIX®. – СПб.: БХВ–Санкт-Петербург, 1999. – 528 с.
6. Соломон Д., Руссинович М. Внутреннее устройство Microsoft Windows 2000. Мастер-класс. / Пер. с англ.– СПб.: Питер; Издательско-торговый дом «Русская редакция», 2001. – 752 с.
7. The Unicode® Standard. Version 11.0 – Core Specification. [Электронный ресурс] 2018 URL: http://www.unicode.org/versions/Unicode11.0.0/UnicodeStandard-11.0.pdf (Дата обращения 20.01.2019)
8. Уорд Б. Внутреннее устройство Linux. — СПб.: Питер, 2016. — 384 с.
9. Лав Р. Linux. Системное программирование. 2-е изд. — СПб.: Питер, 2014. — 448 с.
10. Advanced Configuration and Power Interface (ACPI) Specification. Version 6.2 [Электронный ресурс] 2017. URL: http://www.uefi.org/sites/default/files/resources/ACPI%206_2_A_Sept29.pdf (дата обращения: 11.01.2019)
11. Unified Extensible Firmware Interface (UEFI) Specification Version 2.7 Errata A [Электронный ресурс] 2017 URL: http://www.uefi.org/sites/default/files/resources/UEFI%20Spec%202_7_A%20Sept%206.pdf (дата обращения: 11.01.2019)
12. Дж. Рихтер, Дж. Кларк. Программирование серверных приложений для Microsoft Windows 2000. Изд-ва: Питер, Русская Редакция, 2001 г., 592 стр.