Федеральное агентство по образованию

Государственное образовательное учреждение высшего профессионального образования

СЕВЕРО-ЗАПАДНЫЙ ГОСУДАРСТВЕННЫЙ ЗАОЧНЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

Б.И. Илюшкин

Операционные системы

Процессы и потоки

Учебное пособие

Санкт-Петербург
2005

Утверждено редакционно-издательским советом университета

УДК 681.3.07

Илюшкин Б.И. Операционные системы. Процессы и потоки: Учеб. пособие. – СПб.: СЗТУ, 2005, - 103 с.

Пособие разработано в соответствии с государственными образовательными стандартами высшего профессионального образования по направлению подготовки дипломированного специалиста: 654600 – «Информатика и вычислительная техника»(специальность 220100 – «Вычислительные машины, комплексы, системы и сети») и направлению подготовки бакалавра 552800 – «Информатика и вычислительная техника».

В пособии рассмотрены классификация и архитектура современных операционных систем, концепции управления процессами и потоками, а также механизмы управления виртуальной памятью.

Учебное пособие предназначено для студентов, изучающих дисциплину «Операционные системы» в шестом семестре.

Рецензенты: **М.В.Копейкин**, канд. техн. наук, доц. кафедры компьютерных технологий и программного обеспечения СЗТУ,
О.В.Мотыгин, канд. физ.-мат. наук, старший научный сотрудник института проблем машиноведения РАН

Предисловие

Дисциплина «Операционные системы» относится к циклу обязательных общепрофессиональных дисциплин учебного плана по направлению «Информатика и вычислительная техника» и рассчитана на изучение во 2-м семестре 3-го курса студентами факультета информационных технологий и систем управления.

Согласно Государственному образовательному стандарту, по которому ведется обучение студентов в рамках дисциплины «Операционные системы», студент должен изучить следующие специальные разделы: «Назначение и функции операционных систем (ОС). Мультипрограммирование. Режим разделения времени. Многопользовательский режим работы. Режим работы и ОС реального времени. Универсальные операционные системы и ОС специального назначения. Классификация операционных систем. Модульная структура построения ОС и их переносимость. Управление процессором. Понятие процесса и ядра. Сегментация виртуального адресного пространства процесса. Структура контекста процесса. Идентификатор и дескриптор процесса. Иерархия процессов. Диспетчеризация и синхронизация процессов. Понятия приоритета и очереди процессов. Средства обработки сигналов. Понятие событийного программирования. Средства коммуникации процессов. Способы реализации мультипрограммирования. Понятие прерывания. Многопроцессорный режим работы. Управление памятью. Совместное использование памяти. Защита памяти. Механизм реализации виртуальной памяти. Стратегия подкачки страниц. Принципы построения и защита от сбоев и несанкционированного доступа».

В соответствии с вышесказанным в данном пособии излагаются основные понятия ОС, их классификация и архитектура, а также концепции и механизмы управления процессами, потоками и виртуаль-

ной памятью. Представленный материал посвящен в основном изложению общетеоретических вопросов. Конкретные реализации операционных систем оставлены для самостоятельной проработки студентами при выполнении заданий контрольной работы [10]. Для лучшего усвоения материала рекомендуется анализировать работу приведенных в пособии примеров программ.

Дисциплина «Операционные системы» тесно связана с параллельно изучаемой дисциплиной «Системное программное обеспечение». Во избежание дублирования часть материала, связанного с понятием операционная система, в частности подсистемы ввода/вывода и управления файлами, стратегии диспетчеризации в данном пособии не рассматриваются, поскольку излагаются в курсе «Системное программное обеспечение»[11]. Поэтому для лучшего усвоения материала рекомендуется использовать оба пособия.

При написании пособия автор сосредоточил свои усилия на отборе необходимого материала и изложении его в доступной форме. Большинство теоретических понятий, концепций и схем заимствованы из опубликованных в последнее время учебных руководств (см. список литературы). При этом автор постарался уточнить и более детально изложить отдельные понятия.

В заключение автор выражает признательность М.В.Копейкину и другим сотрудникам кафедры компьютерных технологий и программного обеспечения за ценные замечания, способствовавшие улучшению пособия.

Глава 1. Основные понятия

1.1. Общие сведения об операционных системах

Операционная система (ОС) представляет собой комплекс взаимосвязанных программ, являющихся интерфейсом между приложениями пользователя и аппаратурой компьютера.

С одной стороны, ОС можно рассматривать как виртуальную машину [1]. Виртуальная машина управляется командами более высокого уровня, чем реальная. Для решения своих задач пользователь может обойтись без знания аппаратного устройства компьютера. Например, при работе с диском достаточно представлять его в виде некоторого набора файлов с именами. Последовательность действий при работе с файлом заключается в его открытии, выполнении одной или нескольких операций чтения/записи и в его закрытии.

С другой стороны, ОС можно рассматривать как систему управления ресурсами. К числу основных ресурсов относятся: процессор, память, диски, сетевые устройства и т.д. Управление ресурсами включает решение следующих общих задач:

- планирование ресурса, т.е. определение, какой задаче и в каком количестве следует выделить данный ресурс;
- удовлетворение запросов на ресурсы;
- разрешение конфликтов между задачами.

Для решения этих задач разные ОС используют различные алгоритмы. Например, применяемый алгоритм управления процессором в значительной степени определяет, может ли ОС использоваться как система разделения времени или система реального времени.

Большинство операционных систем постоянно развиваются. Однако основные механизмы, присущие современным ОС, были реализованы в 60-х и 70-х годах прошлого века. К их числу относятся: мультипро-

граммирование, мультипроцессирование, поддержка многопользовательского режима, виртуальная память, файловые системы и работа в сети.

Многозадачные операционные системы позволяют запускать сразу несколько программ в одном сеансе. *Мультипрограммирование* или *многозадачность* – способ организации вычислительного процесса, при котором на одном процессоре попеременно выполняются сразу несколько программ, совместно использующих ресурсы компьютера: процессор, оперативную и внешнюю память, устройства ввода/вывода. Например, MS-DOS является *однозадачной* ОС, а Windows 98 *многозадачной*.

Многопользовательские операционные системы разделения времени предоставляют пользователям возможность интерактивной работы с несколькими приложениями. Всем приложениям попеременно выделяется квант процессорного времени. Так как каждый пользователь может запустить несколько задач, данные системы являются и многозадачными. Например, Windows 2000 Server, UNIX системы и операционные системы мэйнфреймов [5] являются многопользовательскими. *Мэйнфреймами* называют большие компьютеры с сотнями дисков и терабайтами данных. Обычно они включают три вида обслуживания:

- *пакетная обработка* – выполнение задач в пакете без интерактивного воздействия (бухгалтерские пакеты, отчеты по продажам и т.д.);
- *обработка транзакций* – управление большим количеством маленьких запросов (банковские операции, бронирование авиабилетов и т.д.);
- *разделение времени.*

Операционные системы реального времени предназначаются для управления техническими объектами (станками, эксперименталь-

ными установками), технологическими процессами, а также для выполнения других задач, например обслуживание мультимедиа видеосервера. Главным параметром таких систем является время, в течение которого должна быть выполнена программа. Эта характеристика называется *временем реакции* системы. Кроме того, часто задаются повышенные требования к скорости обработки сигналов прерывания (например, от аварийных датчиков).

Многопроцессорные операционные системы устанавливаются на серверах и мощных компьютерах. *Многопроцессорная (мультипроцессорная) обработка* – такой способ организации вычислительного процесса в системах с несколькими процессорами, когда несколько программ (процессов) могут одновременно выполняться на разных процессорах системы [3,5,6]. По логической организации среди многопроцессорных ОС можно выделить две модели [5]:

- *Системы с общей памятью (Shared Memory)*.
- *Системы с распределенной памятью (Distributed Memory)*.

Системы с общей памятью

Все процессоры совместно используют общую память, и каждый имеет доступ к программам и данным в памяти. Наиболее распространенной архитектурой является *симметричная многопроцессорная система* (*SMP – Symmetric MultiProcessor*). Данная архитектура предполагает однородный доступ к памяти (UMA –Uniform Memory Access), т.е. каждое слово данных может быть считано с одинаковой скоростью. Простейшей из архитектур SMP UMA является архитектура с общей шиной [3], представленная на рис. 1.1. Несколько процессоров и модулей памяти одновременно используют общую шину. Когда процессору необходимо прочитать слово в памяти, он сначала проверяет, свободна ли шина. Если свободна, процес-

сор выставляет на нее адрес нужного слова, подает несколько управляющих сигналов и ждет, пока память не выставит нужное слово на шину данных. Если шина занята, процессор ждет, пока она не освободится. Для снижения времени простоя каждому процессору добавляется кэш. Кэш может располагаться внутри микросхемы процессора или рядом с процессором. Может быть несколько уровней кэш памяти. Наличие кэш памяти позволяет сократить число запросов к шине данных, так как большое количество обращений к памяти может быть удовлетворено прямо из кэша.

Так как в каждом локальном кэше хранится образ части основной памяти, в результате изменения слова в одном кэше, в других кэшах слово может оказаться неверным. Поэтому существует аппаратный механизм изменения слова в остальных кэшах, который называется *согласованностью кэшей* [5]. Даже при оптимальном использовании кэша наличие всего одной общей шины ограничивает число UMA – мультипроцессоров до 32 -64.

Реализация операционных систем для архитектуры SMP является более сложной задачей, чем для однопроцессорных систем. ОС для SMP должна управлять процессорами и другими ресурсами компьютера таким образом, чтобы с точки зрения пользователя она выглядела как *многозадачная однопроцессорная*. В частности, усложняются алгоритмы планирования. На однопроцессорной системе планирование одномерно, – какой процесс должен быть запущен следующим. На многопроцессорной системе планирование двухмерно, - какой процесс и на каком процессоре запустить. Механизмы организации памяти разных процессоров должны быть скоординированы. Например, необходимо обеспечить согласованность работы в ситуации, когда несколько процессоров используют одну и ту же страницу или один сегмент памяти. Поддержка одновременных параллельных процессов и потоков требует *реентерабельности* кода ядра, т.е. код ядра

может начинать выполняться до завершения предыдущего выполнения.

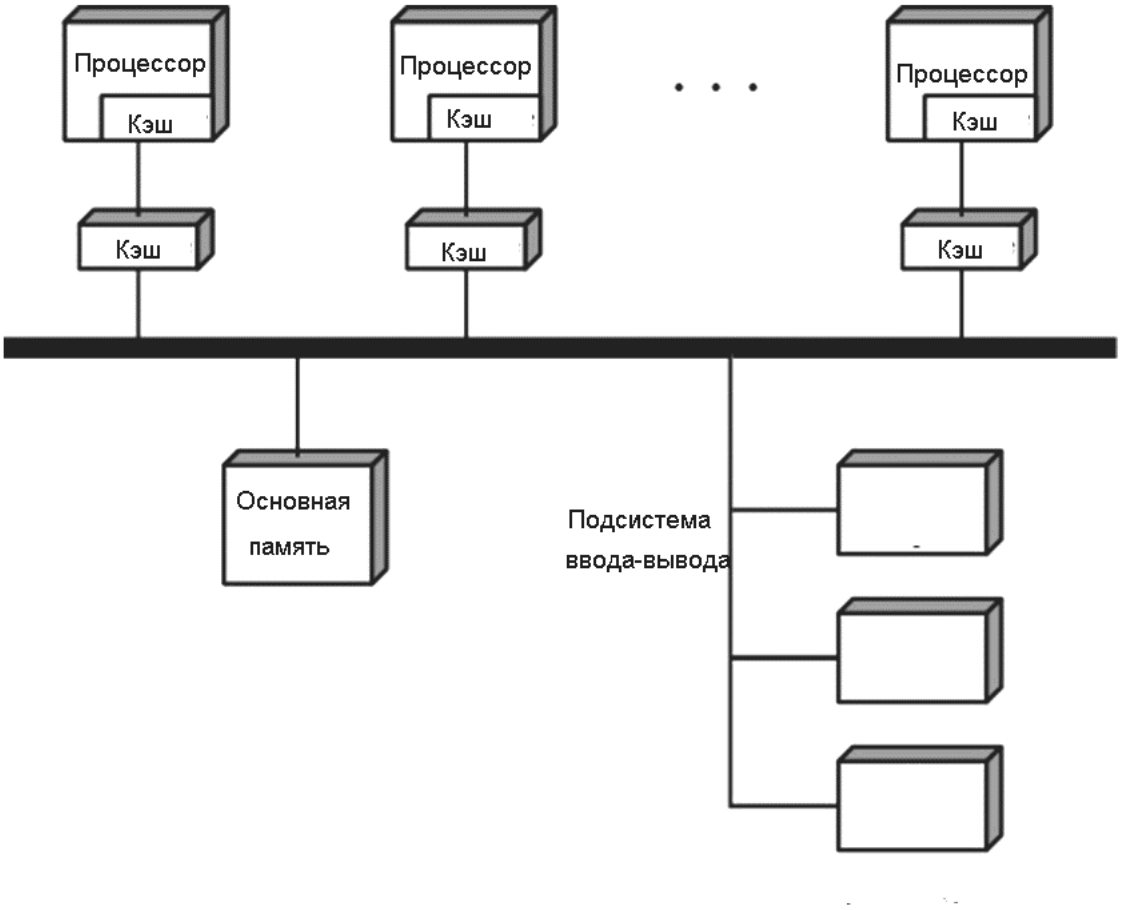


Рис. 1.1. Архитектура симметричной многопроцессорной системы

К другому типу архитектуры относятся системы с *неоднородным доступом* к памяти – *NUMA (Non Uniform Memory Access)*. Мультипроцессоры *NUMA* представляют единое адресное пространство для всех процессоров, однако доступ к локальной памяти осуществляется быстрее, чем к удаленным модулям [5]. Таким образом, все программы, написанные для *UMA*, будут работать и на *NUMA*, но

производительность будет ниже. Архитектура *NUMA* позволяет увеличить число процессоров до нескольких сотен и более.

Системы с распределенной памятью

Данный класс систем включает *многомашинные системы (мульти-компьютеры*) и *кластеры* [3,5].

Многомашинные системы представляют собой тесно связанные процессоры, у которых нет общей памяти. Каждый процессор имеет свое ОЗУ. Базовый узел многомашинной системы состоит из процессора, памяти, сетевого интерфейса и, иногда, жесткого диска. Данный узел может быть упакован в стандартный корпус ПК. В некоторых случаях может использоваться в качестве узла двух или четырех процессорная плата. Многомашинные системы могут включать до 100 – 1000 узлов. Используются различные топологии соединения узлов: кольца, решетки, торы и другие.

Кластеры представляют собой группу взаимосвязанных компьютеров, соединенных высокоскоростной сетью. При этом каждый компьютер может работать самостоятельно. Кластер способен создавать иллюзию работы единой вычислительной системы. Компьютеры, входящие в кластер, называются *узлами (node)*. На каждом из узлов устанавливается программное обеспечение, обеспечивающее унифицированный образ единой системы и поддержку определенных функций и сервисов. В число таких функций входят следующие:

- Единая иерархия файлов. Пользователь видит единую иерархию каталогов файлов в одном корневом.
- Единая виртуальная сеть. Узел кластера может обратиться к любому другому узлу.
- Единое пространство памяти.
- Единый пользовательский интерфейс.

- Единое пространство процессов. Используется единая схема идентификации процессов. Процесс на одном узле может создавать процесс на удаленном узле или взаимодействовать с ним.
- Миграция процессов. Эта функция обеспечивает балансировку загрузки.

Примером кластера является система *Windows 2000 Advanced Server* (до 8 процессоров, 8 Гбайт ОЗУ) [3].

Сетевые операционные системы можно рассматривать как набор операционных систем отдельных компьютеров, составляющих сеть [1]. Операционные системы отдельных компьютеров, работающих в сети, включают согласованный набор коммуникационных протоколов для организации взаимодействия процессов и разделения ресурсов компьютеров между пользователями сети. На рис. 1.2 показаны основные компоненты сетевой ОС [1]. В их число входят:

- *средства управления локальными ресурсами* компьютера реализуют все функции ОС автономного компьютера (распределение оперативной памяти между процессами, планирование и диспетчеризацию процессов, управление процессорами в мультипроцессорных машинах, управление внешней памятью, интерфейс с пользователем и т. д.);
- *сетевые средства*, включающие три компонента:
  - средства предоставления локальных ресурсов и услуг в общее пользование — *серверная часть* ОС;
  - средства запроса доступа к удаленным ресурсам и услугам — *клиентская часть* ОС;
  - *транспортные средства* ОС, которые совместно с коммуникационной системой обеспечивают передачу сообщений между компьютерами сети.

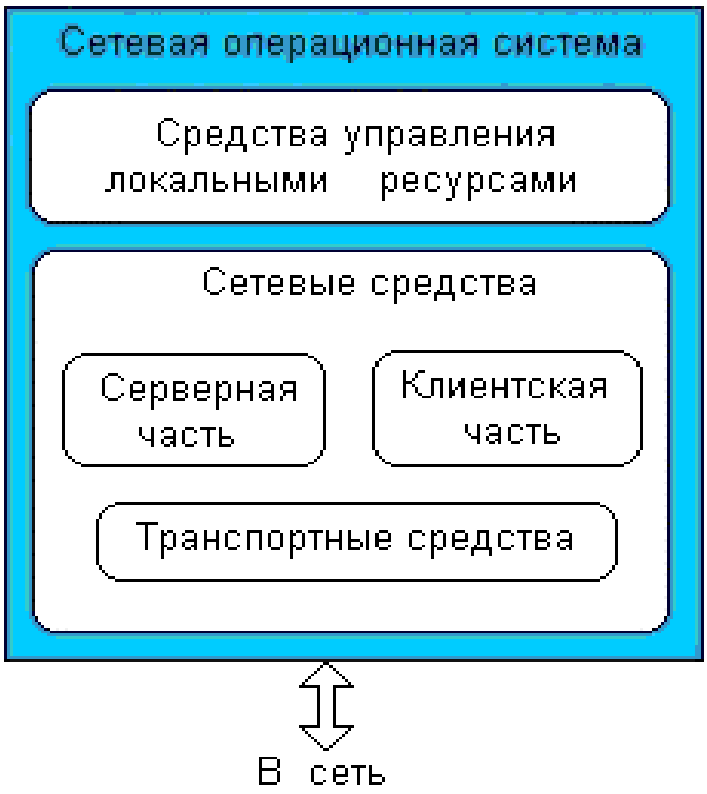


Рис 1.2. Функциональные компоненты сетевой ОС

Совокупность серверной и клиентской частей ОС, предоставляющих доступ к ресурсам компьютера через сеть, называется *сетевой службой*. В настоящее время существует несколько подходов к построению сетевых операционных систем [1]:

- сетевые службы *встроены* в ОС;
- сетевые службы объединены в виде *оболочки* ОС;
- сетевые службы поставляются в виде *отдельных компонент*.

На рис. 1.3 показаны варианты построения сетевых ОС [1]. Первые сетевые ОС представляли собой локальные ОС с надстроенными над ними сетевыми оболочками. В дальнейшем сетевые функции встраиваются в основные модули операционной системы, что позволяет обеспечить более высокую производительность. Примерами таких сетевых ОС являются OS Windows NT/2000, OS/2 Warp и различные версии UNIX.

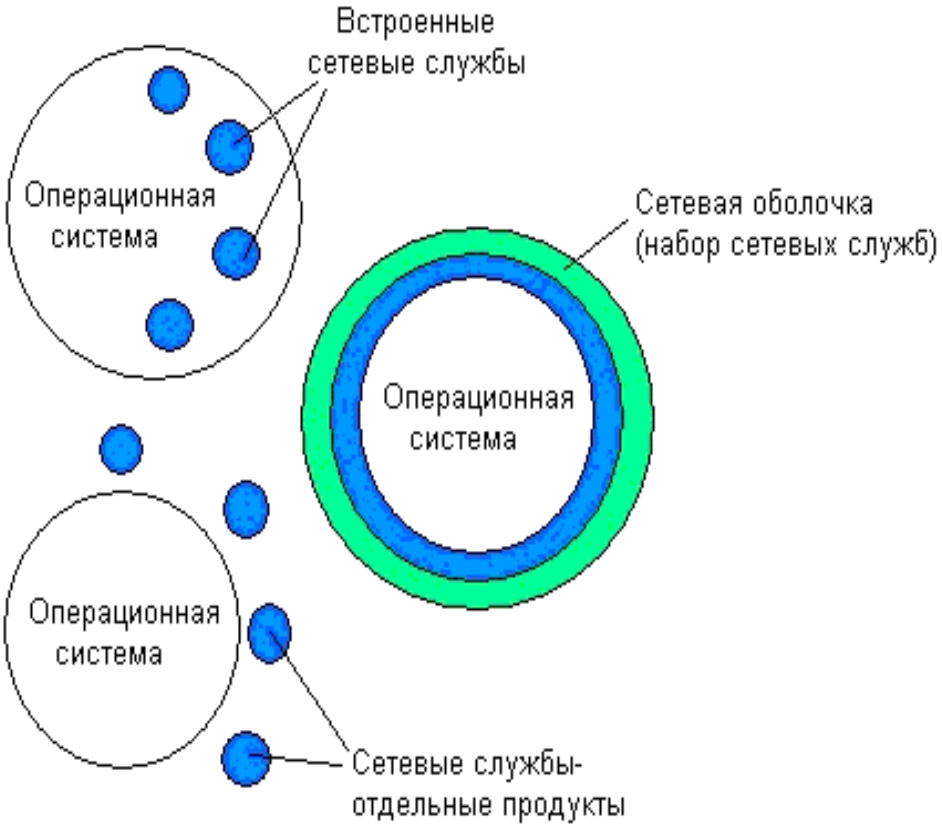


Рис 1.3. Варианты построения сетевых ОС

В настоящее время существуют две основные модели, на основе которых строится компьютерная сеть: одноранговая модель и модель «клиент/сервер»[1]. В одноранговых сетях все узлы равнозначны. Каждый компьютер узла является независимым и использует свои ресурсы, но может предоставлять их другим сетевым узлам. К числу таких ресурсов могут относиться отдельные каталоги с файлами, принтер, компакт-диск. В сетях, созданных на основе модели «клиент/сервер», управление большинством сетевых служб и сетевых ре-

сурсов, используемых клиентскими компьютерами сети, осуществляется одним или несколькими серверами. Сервер играет роль центрального узла администрирования и безопасности сети. Клиент – это программное обеспечение, которое запрашивает сетевые службы.

Сетевые операционные системы, обеспечивающие поддержку выделенного сервера, называются *серверными* [1]. *Серверные ОС* поддерживают более широкий набор функций и включают более развитые средства защиты. Примерами серверных ОС являются многие операционные системы семейств UNIX и Linux, Windows 2000 Server, Windows.NET Server. Большинство современных операционных систем, в частности ОС семейства Windows 95/98/XP, включают компоненты для подключения в качестве клиентского компьютера к сети с выделенным сервером и для подключения к одноранговой сети.

Распределенные операционные системы представляют собой глобальные операционные системы в масштабах вычислительной сети. Они обеспечивают для пользователя более прозрачный доступ к сетевым ресурсам. Примером распределенной ОС является система Sun Cluster, построенная как множество расширений системы Solaris и предоставляющая пользователям единый образ системы. Важными компонентами данной системы являются подсистемы глобального управления процессами и глобальная распределенная файловая система [3].

1.2. Архитектура операционной системы

Структурная организация ОС на основе различных программных компонент определяет ее архитектуру. В наиболее общем виде операционную систему можно разбить на две части: *ядро* и *вспомогательные модули*. *Ядро* является основным компонентом ОС и выполняет наиболее важные функции по организации вычислительного

процесса и поддержки приложений, в число которых входят управление процессами, виртуальной памятью, вводом-выводом и файлами, обработка прерываний [1]. К *вспомогательным модулям* относятся компиляторы, отладчики, редакторы, архиваторы, различные библиотеки и пользовательские оболочки. Для обеспечения эффективной работы ОС большая часть модулей ядра постоянно находится в оперативной памяти, т.е. являются *резидентными*. Важным свойством ядра является его работа в *привилегированном* режиме (режиме *ядра*) процессора, который позволяет осуществлять полный контроль доступа к памяти, регистрам, устройствам ввода-вывода, переключению процессора с задачи на задачу. По архитектурному исполнению операционные системы можно разбить на два класса: *монолитные* и с архитектурой *микроядра*.

Монолитные ОС представляют собой многоуровневые модульные структуры с иерархической организацией функций [1,3]. Взаимодействие осуществляется между функциями, находящимися на соседних уровнях в соответствии с определенными правилами. На рис. 1.4 показана многослойная структура ядра, состоящего из следующих слоев [1]:

- *Средства аппаратной поддержки ОС*, включающие средства поддержки привилегированного режима, систему прерываний, средства переключения контекстов процессов, средства защиты областей памяти и т. п.
- *Машинно-зависимые компоненты ОС*. Этот слой образуют программные модули, в которых отражается специфика аппаратной платформы компьютера. В идеале этот слой полностью экранирует вышележащие слои ядра от особенностей аппаратуры. Это позволяет разрабатывать вышележащие слои на основе машинно-независимых модулей для всех типов аппаратных платформ, под-

держиваемых данной ОС. Примером экранирующего слоя может служить слой HAL операционных систем Windows NT/2000.

- *Базовые механизмы ядра*. Данный слой выполняет примитивные операции ядра: программное переключение контекстов процессов, диспетчеризацию прерываний, перемещение страниц из памяти на диск и обратно и т. п. Модули данного слоя являются исполнительными механизмами для модулей верхних слоев. Например, решение о прерывании выполнения текущего процесса А и выполнении процесса В, принимается менеджером процессов на вышележащем слое, а слою базовых механизмов передается директива о выполнении переключения с контекста текущего процесса на контекст процесса В.

- *Менеджеры ресурсов*. Этот слой включает функциональные модули управления основными ресурсами вычислительной системы. В их число входят *менеджеры* (называемые также *диспетчерами*) процессов, ввода-вывода, файловой системы и виртуальной памяти.

- *Интерфейс системных вызовов*. Этот слой является самым верхним слоем ядра и взаимодействует непосредственно с приложениями и системными утилитами, образуя *прикладной программный интерфейс(API)* операционной системы. Функции API, обслуживающие системные вызовы, предоставляют возможность использования средств операционной системы при разработке прикладных программ. Например, в операционной системе UNIX с помощью системного вызова `fd = open("/doc/a.txt", 0_RDONLY)` приложение открывает файл `a.txt`, хранящийся в каталоге `/doc`, а с помощью системного вызова `read(fd, buffer, count)` читает из этого файла в область своего адресного пространства, имеющую имя `buffer`, некоторое количество байт [1]. Для осуществления таких действий *системные вызовы*

обращаются к функциям слоя *менеджеров ресурсов*, причем для выполнения одного системного вызова может понадобиться несколько таких обращений.

Рис 1.4. Многослойная структура ядра ОС

Микроядерные ОС предполагают наличие *микроядра*, работающего в привилегированном режиме и состоящего из машинно-зависимых модулей, реализующих базовые функции операционной системы [1,2,3]. В число таких функций входят:

- управление процессами;
- управление виртуальной памятью;
- управление устройствами ввода-вывода и прерываниями;

- межпроцессорные коммуникации.

Набор функций *микроядра* обычно соответствует функциям слоя *базовых механизмов монолитного ядра* [1]. Остальные функции ядра оформляются в виде системных сервисов, работающих в пользовательском режиме как приложения *микроядра*. Такие сервисы называются *серверами ОС*, т.е. модулями, основным назначением которых является обслуживание запросов приложений и других модулей ОС. На рис. 1.5 схематично показан механизм обращения к серверам ОС через *микроядро* [1]. Клиент (прикладная программа или другой компонент ОС) запрашивает выполнение некоторой функции у соответствующего сервера, посылая ему сообщение. *Микроядро* имеет доступ к адресным пространствам обоих приложений, поскольку выполняется в *привилегированном* режиме, и потому является посредником. *Микроядро* сначала передает сообщение, содержащее имя и параметры вызываемой процедуры нужному серверу, затем сервер выполняет запрошенную операцию, после чего ядро возвращает результаты клиенту с помощью другого сообщения. Таким образом, работа *операционной системы с микроядром* соответствует известной модели *клиент-сервер*, в которой роль транспортных средств выполняет *микроядро*.

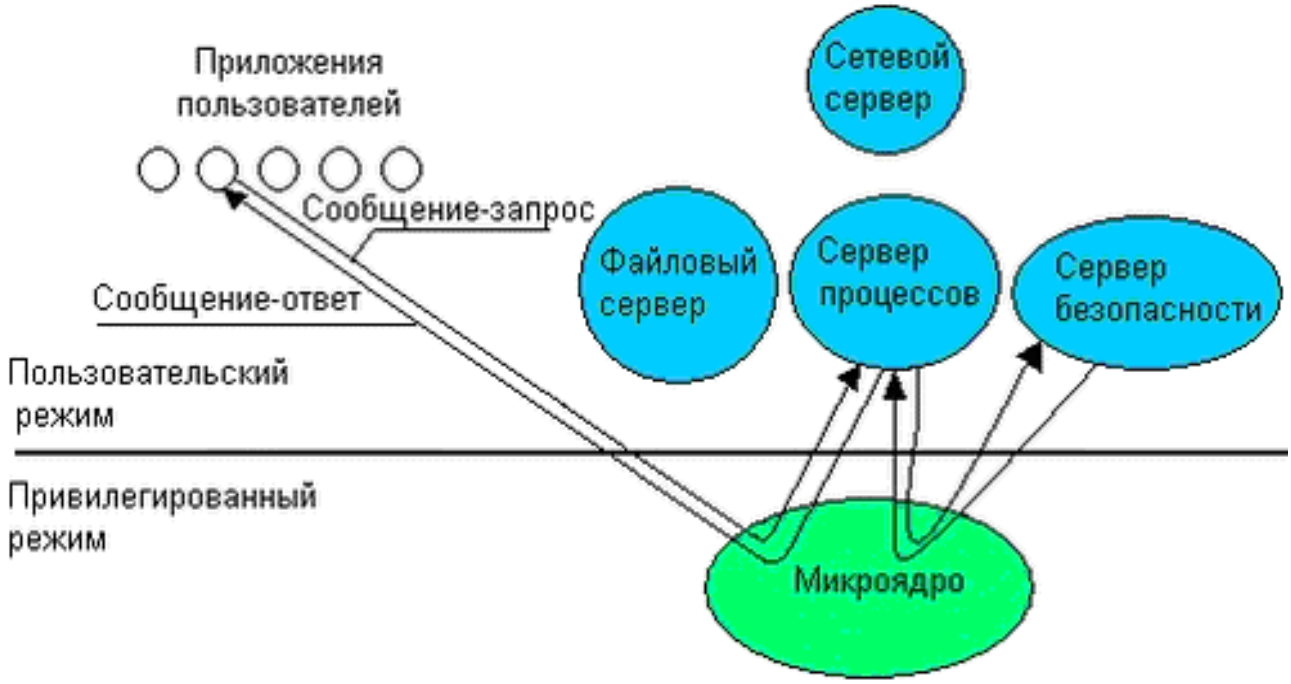


Рис 1.5. Схема системного вызова в архитектуре с микроядром

К числу достоинств архитектуры с микроядром относятся следующие качества [1,3]:

- *расширяемость*, т.е. возможность добавлять новые сервисы, или модифицировать, или убирать старые.
- *переносимость,* т.е. минимум изменений кода микроядра за счет его компактности.
- *надежность,* поскольку каждый сервер выполняется в виде отдельного процесса в своей области памяти и в случае краха может быть перезапущен без останова остальных серверов. Кроме того, компактное микроядро легче протестировать и получить вследствие этого более качественный код.
- *способность поддержки распределенных систем* за счет использования единообразного интерфейса запросов, генерируемых процессами, и поддержки механизма взаимодействия клиентов и серверов путем обмена сообщениями.

Основным недостатком архитектур с микроядром является их низкая *производительность*. Создание сообщения и отправка его через микроядро с последующим получением ответа занимает больше времени, чем непосредственный вызов сервиса. Наиболее ярким представителем ОС с архитектурой микроядра является ОС реального времени QNX [2]. Микроядро QNX имеет объем всего несколько десятков килобайт и поддерживает только планирование и диспетчеризацию процессов, взаимодействие процессов, обработку прерываний и сетевые службы нижнего уровня. Основная область ее применения – *встроенные* системы.

Глава 2. Процессы и потоки

2.1. Основные определения

Программа – статический объект, представляющий собой файл или совокупность файлов с кодами и данными. Для того чтобы программа могла быть запущена на выполнение, операционная система должна создать окружение или среду выполнения задачи, включающую возможности доступа к различным системным ресурсам (память, устройства ввода-вывода, файлы и т.д.). Такое окружение получило название *процесса* [4]. *Процесс* представляет собой исполняемый образ программы, включающий отображение в памяти исполняемого файла, полученного в результате компиляции и связывания исходного кода программы, кодов данных и библиотек, стека, а также ряда структур данных ядра, необходимых для управления процессом. В целом, процесс можно представить как совокупность данных ядра системы, необходимых для описания образа программы в памяти и управления ее выполнением.

Процессы можно условно разбить на три категории:

- *системные*;
- *фоновые (демоны)*;
- *прикладные (пользовательские)*.

Системные процессы являются частью ядра ОС и всегда расположены в оперативной (основной) памяти. Выполняемые инструкции и данные этих процессов находятся в ядре системы, и поэтому они могут вызывать функции и обращаться к данным, недоступным для остальных процессов, например диспетчер страничного замещения, диспетчер памяти ядра, диспетчер буферного кэша и другие.

Фоновые процессы или *демоны* – это неинтерактивные процессы, которые обычно запускаются при инициализации системы (после инициализации ядра) и обеспечивают работу различных подсистем. Например, системы терминального доступа, системы печати, системы сетевого доступа и другие. *Демоны* не связаны с пользовательскими сеансами работы и не могут непосредственно управляться пользователями.

Прикладные процессы, как правило, порождаются в рамках пользовательского сеанса. Они могут выполняться как в интерактивном, так и в фоновом режимах.

Большинство процессоров поддерживают два режима работы: *привилегированный*, или *режим ядра*, и *пользовательский*, или *режим задачи*. Определенные команды выполняются только в привилегированном режиме, например команды управления памятью и вводом-выводом. Режим работы устанавливается в регистре слова состояния процессора (PSW) битом режима выполнения, который может быть изменен при наступлении некоторых событий. Например, если в результате прерывания управление пользовательским процессом переходит к процедуре ОС, данная процедура изменяет режим выполнения на привилегированный. Перед возвращением управления

пользовательскому процессу режим выполнения изменяется обратно на пользовательский. Программы, выполняющиеся в режиме ядра, обладают полным контролем над процессором и имеют доступ ко всем ячейкам памяти.

2.2. Модель процесса

Можно рассматривать все функционирующее на компьютере программное обеспечение, включая операционную систему, в виде набора процессов. Каждый процесс можно описать набором параметров, включая текущие значения счетчика команд, регистров и переменных. С позиций данной абстрактной модели у каждого процесса есть собственный виртуальный центральный процессор [5].В действительности реальный процессор переключается с одного процесса на другой. Это переключение и называется *многозадачностью* или *мультипрограммированием*.

Для реализации модели процессов операционная система содержит системную таблицу процессов (массив структур) с одним элементом для каждого процесса. Данный элемент называется *блоком управления* или *дескриптором процесса*. В *дескрипторе процесса* прямо или косвенно (через указатели на связанные с процессом структуры) содержится информация о состоянии процесса, его приоритетах, идентификаторе, параметрах планирования, о расположении образа процесса в оперативной памяти и на диске, об ожидаемых процессом событиях, а также другая оперативная информация, необходимая ядру системы в течение всего жизненного цикла процесса, независимо от того, находится ли процесс в активном или пассивном состоянии [1]. При управлении процессами ядро ОС кроме дескриптора использует другую информационную структуру, называемую *контекстом процесса. Контекст процесса* содержит более объемную

часть информации о процессе, необходимую для возобновления выполнения прерванного процесса. Эта информация включает содержимое регистров процессора, данные об открытых файлах и незавершенных операциях ввода-вывода, коды ошибок выполняемых системных вызовов и другие данные, характеризующие состояние вычислительной среды в момент прерывания.

Каждому процессу операционной системой выделяется виртуальное адресное пространство, представляющее собой набор виртуальных адресов, необходимых для выполнения процесса. Для прикладных программ эти адреса первоначально назначаются транслятором при создании сегментов кода и данных. Затем, при создании процесса, ОС фиксирует назначенное виртуальное адресное пространство в собственных системных таблицах. В ходе выполнения процесс может увеличить размер назначенного виртуального адресного пространства, запросив у ОС создания дополнительных сегментов или увеличения существующих. Максимальный размер виртуального адресного пространства ограничивается разрядностью адреса данной архитектуры компьютера. Например, для 32-разрядных процессоров Intel Pentium ОС может предоставить каждому процессу виртуальное адресное пространство до 4 Гбайт. Содержимое назначенного процессу виртуального адресного пространства представляет собой *образ процесса.*

Выполнение процесса может происходить в двух режимах: в режиме *ядра (kernel mode)* или режиме *задачи (user mode)*. В режиме задачи процесс выполняет инструкции прикладной программы, допустимые на *непривилегированном* уровне защиты процессора. При этом процессу недоступны системные структуры данных [4]. Для получения услуг ядра процессу необходимо сделать системный вызов, после чего выполнение процесса переходит на *привилегированный* уровень (в режим *ядра*). Таким образом, ядро системы защищает соб-

ственное адресное пространство от доступа прикладного процесса, который может нарушить целостность структур данных ядра. Соответственно и образ процесса состоит из двух частей: данных режима задачи и данных режима ядра. Образ процесса в *режиме задачи* состоит из сегментов кода, данных, стека, библиотек. Образ процесса в *режиме ядра* состоит из структур данных, которые используются ядром для управления процессом.

2.3. Управление процессами

Для управления процессами и выделяемыми им ресурсами ОС использует четыре вида управляющих таблиц [3].

- *Таблицы памяти* – используются для отслеживания оперативной и виртуальной памяти. Содержат следующую информацию:
  - объем оперативной и виртуальной памяти, отведенной процессу;
  - атрибуты защиты блоков памяти (указывают, какой из процессов имеет доступ к той или иной совместно используемой памяти);
  - данные, необходимые для управления виртуальной памятью.
- *Таблицы ввода-вывода* – используются для управления устройствами ввода-вывода. В каждый момент времени устройство ввода-вывода может быть либо свободно, либо занято определенным процессом.
- *Таблицы файлов* – содержат информацию о существующих файлах, их расположении, текущем состоянии и других атрибутах. Основная часть этой информации поддерживается файловой системой.

❖ *Таблицы процессов* – содержат указатели на образы процессов.

Все четыре вида таблиц связаны между собой (рис.2.1) и имеют перекрестные ссылки. Данные таблицы создаются при генерации ОС на основе информации, содержащейся в конфигурационных файлах. Управление процессами осуществляется ОС в режиме ядра и включает следующие функции:

- создание и завершение процессов;
- планирование и диспетчеризация процессов;
- переключение процессов;
- синхронизация и обмен информацией между процессами;
- организация управляющих блоков процессов.

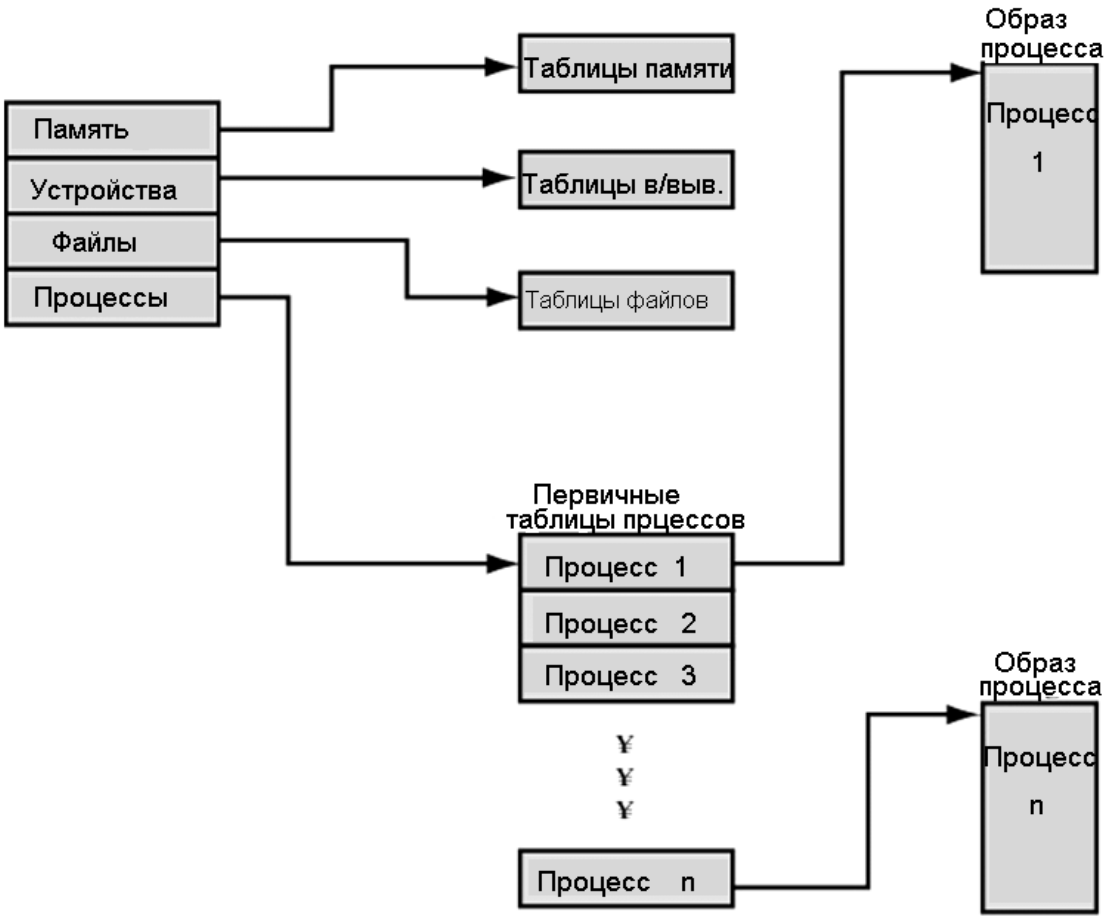


Рис 2.1. Структура управляющих таблиц ОС

2.3.1. Создание и завершение процессов

Для создания нового процесса операционной системе необходимо выполнить определенную последовательность действий [3]:

- присвоить новому процессу *уникальный идентификатор*, т.е. занести новую запись в таблицу процессов;
- выделить *пространство для процесса*, т.е. выделить адресное пространство для всех элементов образа процесса;
- инициализировать *управляющий блок процесса*;
- поместить процесс в список *“готовых” или “готовых приостановленных процессов”*;
- загрузить часть *кодов и данных процесса* в оперативную память.

Информация о состоянии процессора обычно инициализируется нулевыми значениями, за исключением счетчика команд (содержит точку входа в программу) и указателей системного стека (задающих границы стека процесса). Состояние процесса обычно инициализируется значением “готов” или “готов и приостановлен”.

Основными причинами создания процессов являются:

- запуск задач пользователей и заданий в среде пакетной обработки;
- поступление запросов от приложений на выполнение некоторых функций;
- порождение процессов другими процессами.

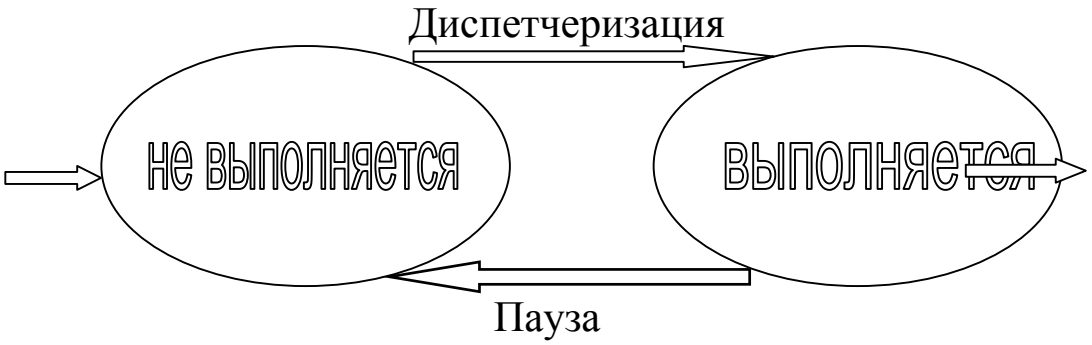
Когда один процесс порождает другой, то порождающий процесс называется *родительским (parent)*, а порождаемый процесс называется *дочерним (child)*. Порождение процессов используется для структурирования приложений или распараллеливания вычислений. Например, файловый сервер может генерировать новый процесс для каждого обрабатываемого им запроса.

Основными причинами завершения процессов являются:

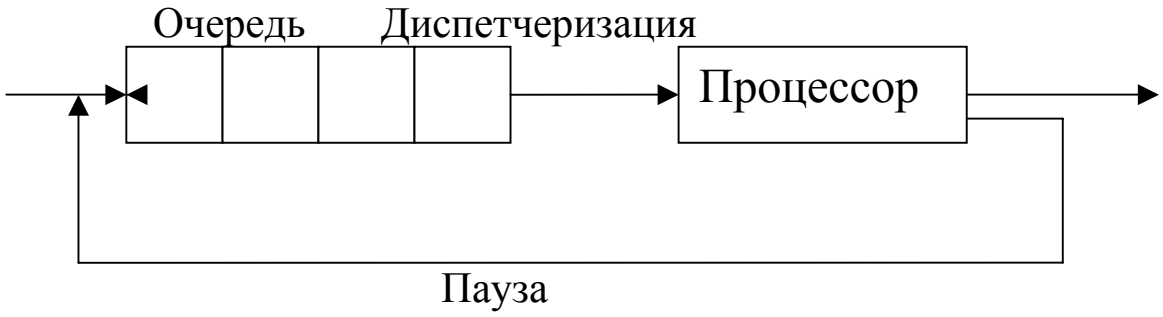
- нормальное завершение;
- превышение предельного лимита времени;
- превышение лимита отведенной памяти;
- ошибки при выполнении;
- вмешательство пользователя, администратора или ОС;
- завершение родительского процесса.

2.3.2. Планирование и диспетчеризация процессов

Самую простую модель *диспетчеризации*, т.е. определения схемы чередования процессов, можно построить, используя модель процесса с двумя состояниями: *выполняющийся* или *не выполняющийся*. На рис. 2.2 показана модель процесса с двумя состояниями.



а) Схема перехода состояний



б) Схема использования очереди

Рис 2.2. Модель процесса с двумя состояниями

Поведение диспетчера можно описать следующим образом. Прерванный процесс переходит в очередь процессов, ожидающих выполнения. В случае завершения процесс выводится из системы. Диспетчер выбирает из очереди следующий процесс для выполнения. Если бы все процессы всегда были готовы к выполнению, то очередь на рис 2.2 могла бы работать эффективно, т.е. обслуживать имеющиеся в наличии процессы *карусельным (round robin*) методом, когда каждому процессу отводится определенный промежуток времени, по истечении которого процесс возвращается обратно в очередь. Однако модель с двумя состояниями не является адекватной. Некоторые из невыполняющихся процессов готовы к выполнению, в то время как другие являются заблокированными и ждут окончания операции ввода-вывода. Поэтому более адекватной реальности является модель процесса с пятью состояниями:

- *выполняющийся*;
- *готовый к выполнению*;
- *блокированный* (процесс не может выполниться до наступления некоторого события, например до завершения операции ввода-вывода);
- *новый* (только что созданный процесс, который еще не помещен ОС в пул выполнимых процессов и не загружен в ОЗУ);
- *завершающийся* (процесс, удаленный ОС из пула выполнимых процессов).

На рис. 2.3 показаны типы событий, соответствующие каждому из возможных переходов из одного состояния в другое. Возможны следующие переходы:

- *Нулевое состояние/Новый*. ОС выполняет все необходимые действия для создания нового процесса. Присваивает

процессу идентификатор, формирует управляющие таблицы процесса.

- *Новый/Готовый*. ОС переводит процесс в состояние готового к выполнению по мере готовности к обработке дополнительных процессов. Существуют ограничения на количество запущенных процессов и на объем выделяемой для процессов виртуальной памяти (для поддержки нормальной производительности системы).
- *Готовый/Выполняющийся*. ОС выбирает один из готовых для выполнения процессов в соответствии с принятой стратегией планирования.
- *Выполняющийся/Завершающийся*. ОС прекращает выполнение процесса по одной из причин завершения процессов.
- *Выполняющийся/Готовый*. ОС прерывает выполнение процесса или вытесняет процесс по истечении заданного кванта времени или из-за наличия другого процесса с более высоким приоритетом.
- *Выполняющийся/Блокированный*. Процесс переводится в заблокированное состояние, если для продолжения работы требуется наступление некоторого события (ожидание завершения операции ввода-вывода, сообщения от другого процесса и т.п.).
- *Блокированный/Готовый*. Заблокированный процесс переходит в состояние готовности к выполнению при наступлении ожидаемого события.
- *Готовый/Завершающийся*. Родительский процесс может прервать выполнение дочернего процесса. Дочерний процесс может прекратиться при завершении родительского процесса.

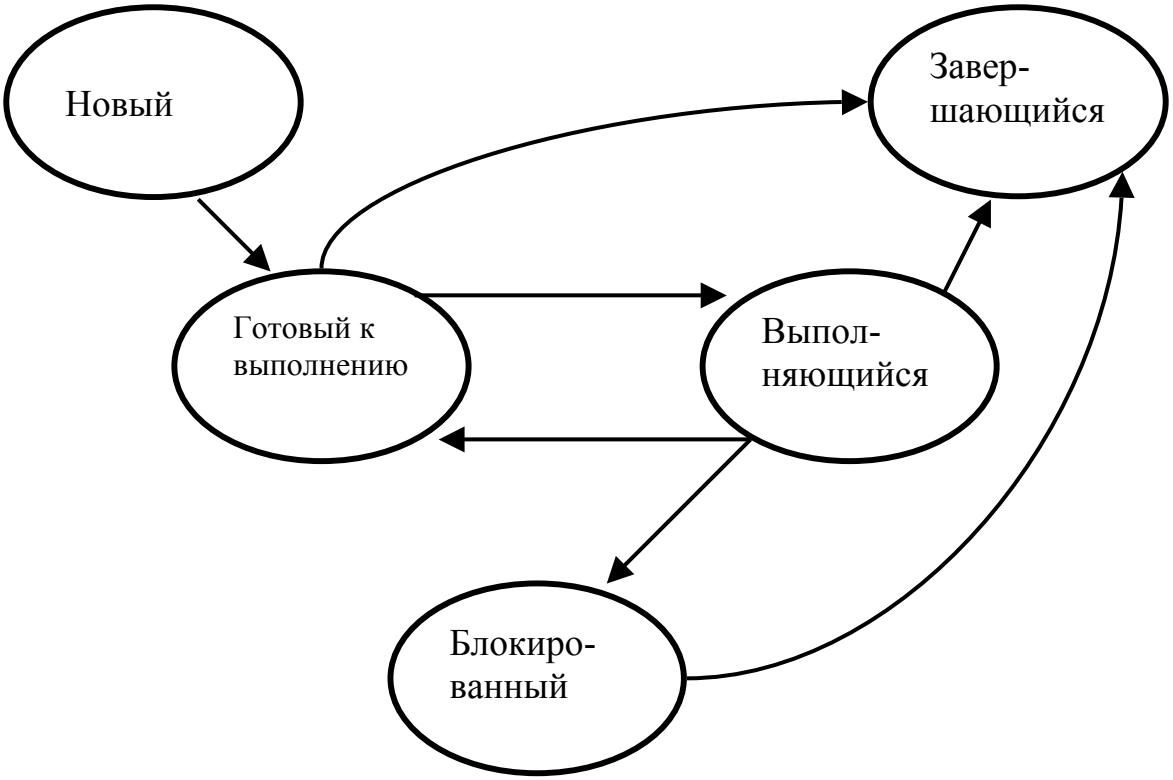
- *Блокированный/Завершающийся*. Аналогично предыдущему пункту.

Обобщением модели с пятью состояниями является модель с семью состояниями, позволяющая более адекватно смоделировать реализацию операционных систем. В данной модели вводится дополнительно понятие *приостановленного* процесса и два новых состояния:

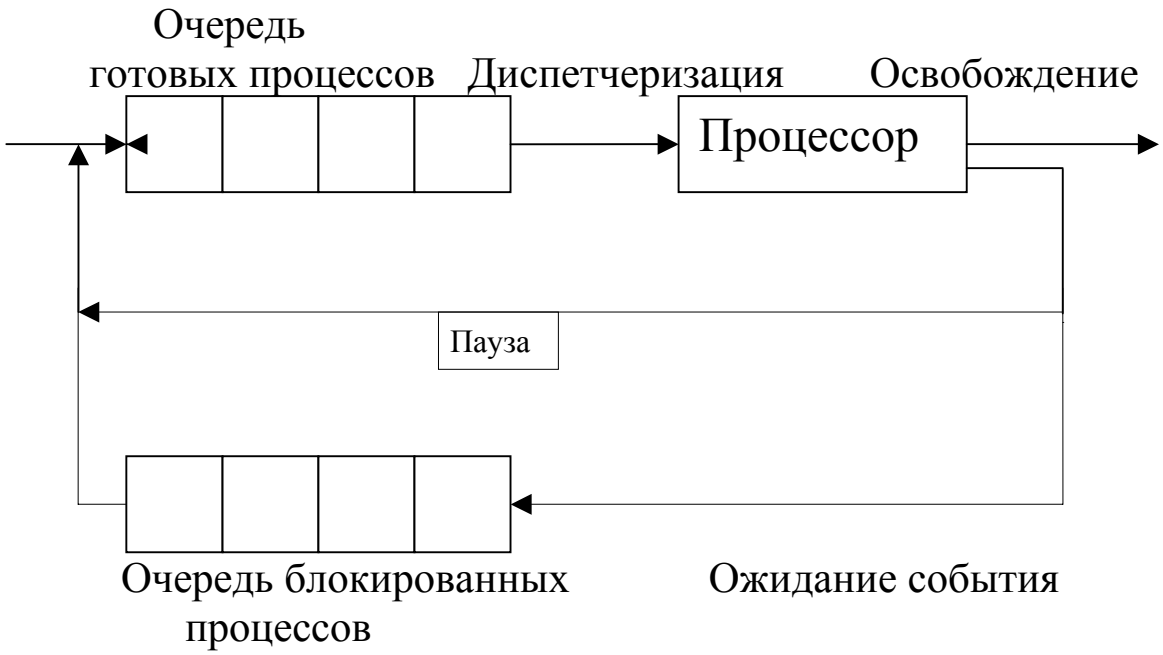
- *Блокированный/Приостановленный.* Процесс, выгруженный на диск и ожидающий какого-то события.
- *Готовый/Приостановленный.* Процесс, находящийся на диске, но уже готовый к выполнению. Для этого его необходимо только загрузить в основную память.

Приостановленный процесс является процессом, который отсутствует в основной памяти и не может быть запущен немедленно. Причинами приостановки процесса могут быть:

- необходимость освободить основную память для оптимизации производительности виртуальной памяти;
- периодический режим выполнения процесса (например, программа анализа использования ресурсов компьютера);
- приостановка дочерних процессов родительским для их проверки или координации;
- приостановка пользовательского процесса для отладки программы;
- приостановка фоновых процессов операционной системой в случае выявления проблем.



а) Схема перехода состояний



б) Схема с одной очередью блокированных процессов

Рис 2.3. Модель процесса с пятью состояниями

2.4. Потоки

Поток является последовательностью команд, обрабатываемых процессором. В рамках одного процесса могут находиться один или несколько потоков. Традиционный подход, при котором каждый процесс представляет собой единый поток выполнения, называется *однопоточным* [3]. Например, MS-DOS поддерживает один однопоточный пользовательский процесс. Некоторые ОС семейства UNIX поддерживают процессы множества пользователей, но при этом каждый из процессов содержит один поток. *Многопоточностью (multithreading)* называется способность ОС поддерживать в рамках одного процесса выполнение нескольких потоков. Примерами многопоточных систем являются среда выполнения Java, Windows 2000, Linux, Solaris и другие. На рис. 2.4 представлены варианты однопоточных и многопоточных процессов [3].

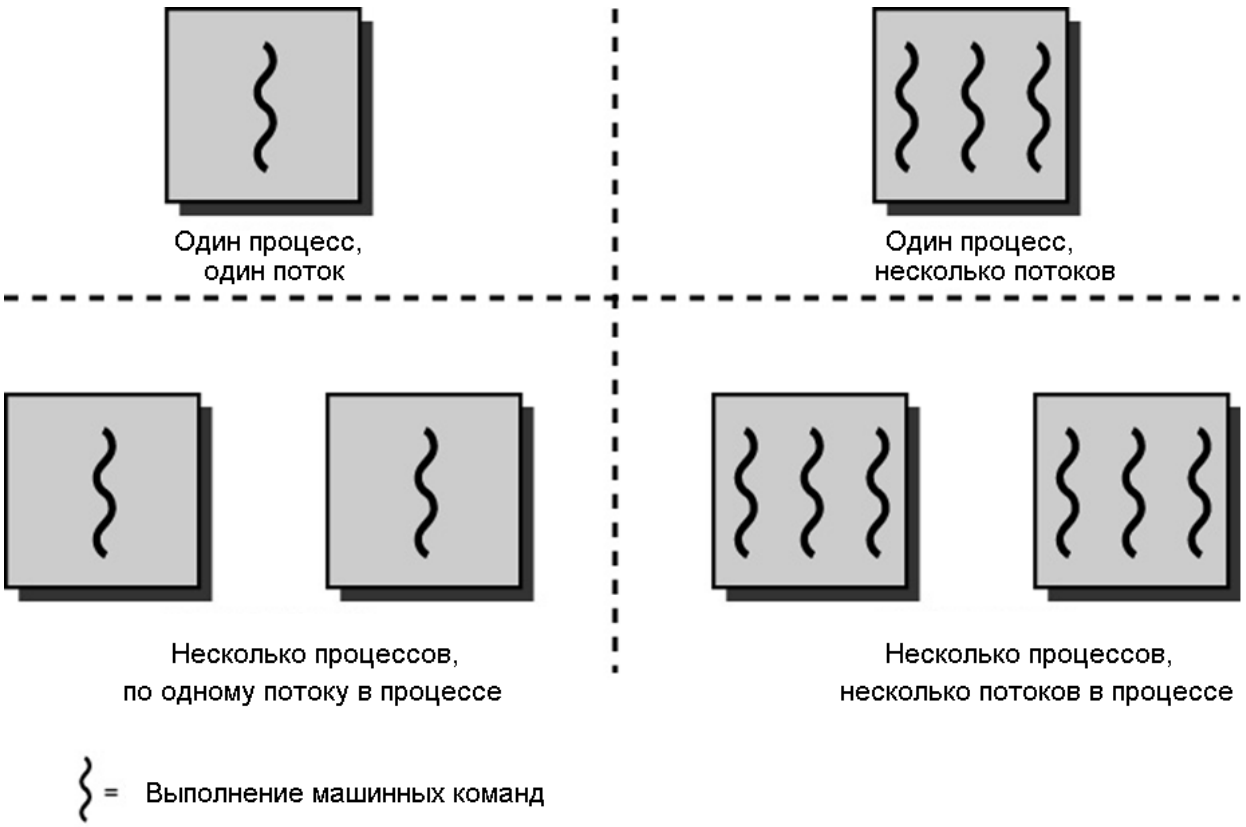


Рис 2.4. Однопоточные и многопоточные процессы

В многопоточной среде *процесс* можно рассматривать как *структурную единицу объединения ресурсов* и *структурную единицу защиты*. Ресурсами являются адресное пространство, открытые файлы, дочерние процессы, обработчики сигналов и многое другое. Под защитой подразумевается защищенный режим доступа к процессорам, другим процессам, файлам и ресурсам ввода-вывода. С другой стороны, *процесс* можно рассматривать как *общий поток исполняемых команд*, состоящий из нескольких отдельных потоков [5]. У каждого потока есть свой счетчик команд, регистры и стек. Таким образом, в многопоточной среде процессы используются для группирования ресурсов, а потоки являются объектами, поочередно выполняющимися на процессоре (в случае однопроцессорной вычислительной

системы), создавая впечатление параллельной работы потоков. Все потоки одного процесса разделяют между собой ресурсы процесса. Они находятся в общем адресном пространстве и имеют доступ к одним и тем же данным. Например, если один поток изменяет данные в памяти, то другие потоки имеют возможность отследить эти изменения. Если один поток открывает файл с правом чтения, другие потоки данного процесса могут читать из этого файла.

Использование потоков имеет следующие преимущества с точки зрения производительности [3]:

- Создание нового потока в уже существующем процессе занимает существенно меньше времени, чем создание нового процесса.
- Поток можно завершить быстрее, чем процесс.
- Переключение потоков в рамках одного процесса происходит намного быстрее.
- Использование потоков повышает эффективность обмена информацией между процессами. В большинстве операционных систем обмен информацией между процессами происходит с участием ядра, обеспечивающего механизм осуществления обмена и защиту. Обмен информацией между потоками может производиться без участия ядра.

2.4.1. Многопоточная модель процесса

Однопоточная модель процесса включает управляющий блок процесса, пользовательское адресное пространство (программы и данные), а также стеки ядра и пользователя, с помощью которых осуществляются вызовы процедур и возвраты из них при выполнении процесса. *Многопоточная модель процесса*, помимо управляющего блока процесса и адресного пространства пользователя, включает до-

полнительно для каждого потока счетчик команд, регистры, стек, локальную память, состояние выполнения.

- *Счетчик команд* – отслеживает порядок выполнения.
- *Регистры* – используются для хранения текущих переменных.
- *Стек* – содержит протокол выполнения процесса, где для каждой вызванной, но еще не вернувшейся процедуры, отведен отдельный фрейм. Во фрейме находятся локальные переменные процедуры и адрес возврата. Каждый поток может вызывать различные процедуры и соответственно иметь различный протокол выполнения процесса.
- *Локальная память* – статическая память, выделяемая потоку для локальных переменных.
- *Состояние выполнения потока* – одно из четырех основных состояний (готовый к выполнению, выполняющийся, блокированный, завершающийся).

В многопоточном режиме процессы обычно запускаются с одним потоком. Этот поток может создавать новые потоки, определив их указатели команд и аргументы. Новые потоки создаются со своим собственным контекстом регистров и стековым пространством, после чего помещаются в очередь готовых к выполнению потоков. В случае необходимости ожидания некоторого события поток блокируется (при этом сохраняется содержимое его пользовательских регистров, счетчика команд и указателя стека). После этого процессор может перейти к выполнению другого потока. После завершения потока его контекст регистров и стеки удаляются.

Многопоточная модель процесса охватывает две категории потоков: *потоки на уровне пользователя* (user-level threads – ULT) и *потоки на уровне ядра (*kernel – level threads – KLT) [3,5]. *Потоки на уровне пользователя* управляются самим приложением. Обычно приложение в начале своей работы состоит из одного потока, с которого

начинается выполнение данного приложения, и который размещается в процессе, управляемом ядром. Приложение может создать новый поток при помощи вызова библиотечной процедуры работы с потоками, например thread_create(). В результате создается структура данных для нового потока и управление передается к одному из готовых к выполнению потоков данного процесса в соответствии с заданным алгоритмом планирования. При этом контекст текущего потока сохраняется. При возврате управления к данному потоку его контекст восстанавливается. Все управление ULT осуществляется в пользовательском пространстве в рамках одного процесса. Связь с ядром отсутствует. Ядро продолжает осуществлять планирование процесса как единого целого. Однако существует взаимосвязь между планированием потоков и планированием процессов. В случае возникновения прерывания процесса, например по вводу-выводу или таймеру, выполняющийся поток процесса продолжает оставаться в состоянии выполнения, хотя перестает выполняться на процессоре. При возврате управления процессу возобновляется выполнение потока на процессоре.

Использование *потоков на уровне пользователя* обладает определенными преимуществами по сравнению с использованием *потоков на уровне ядра* [3]. К числу таких преимуществ относятся:

- Для управления потоками процессу не нужно переключаться в режим ядра, что позволяет избежать дополнительных расходов.
- Планирование потоков может производиться в зависимости от специфики приложения. Для одних приложений лучше подходит простой алгоритм круговой диспетчеризации, а для других - алгоритм планирования с использованием приоритетов.
- Использование потоков на уровне пользователя применимо для любой операционной системы. Библиотека потоков представля-

ет собой набор процедур, работающих на уровне приложения и совместно используемых всеми приложениями.

К числу недостатков использования ULT относятся:

- При выполнении операционной системой системного вызова блокируется не только данный поток, но и все другие потоки данного процесса.
- Поскольку ядро закрепляет за каждым процессом только один процессор, несколько потоков одного процесса не могут выполняться одновременно даже в случае многопроцессорной системы.

Потоки на уровне ядра управляются самим ядром через интерфейс прикладного программирования (API) средств ядра, управляющих потоками [3]. Ядро поддерживает контекст процесса, а также контексты каждого отдельного потока процесса. Планирование выполняется ядром исходя из состояния потоков. Благодаря этому ядро может осуществлять планирование работы нескольких потоков одного процесса на нескольких процессорах. Кроме того, при блокировке одного из потоков процесса, ядро может выбрать для выполнения другой поток данного процесса. Основным недостатком использования *потоков на уровне ядра* являются дополнительные накладные расходы на переключения между потоками внутри одного процесса за счет переходов в режим ядра.

Если для большинства потоков приложения необходим доступ к ядру, то использование схемы KLT более целесообразно. Примерами использования потоков на уровне ядра являются ОС W2K, Linux. В некоторых операционных системах, например в ОС Solaris, используется комбинированный подход [3]. Потоки на пользовательском уровне, входящие в приложение, отображаются в такое же или меньшее число потоков на уровне ядра (рис 2.5). При этом число потоков на уров-

не ядра является изменяемым параметром, что позволяет оптимизировать работу приложения.

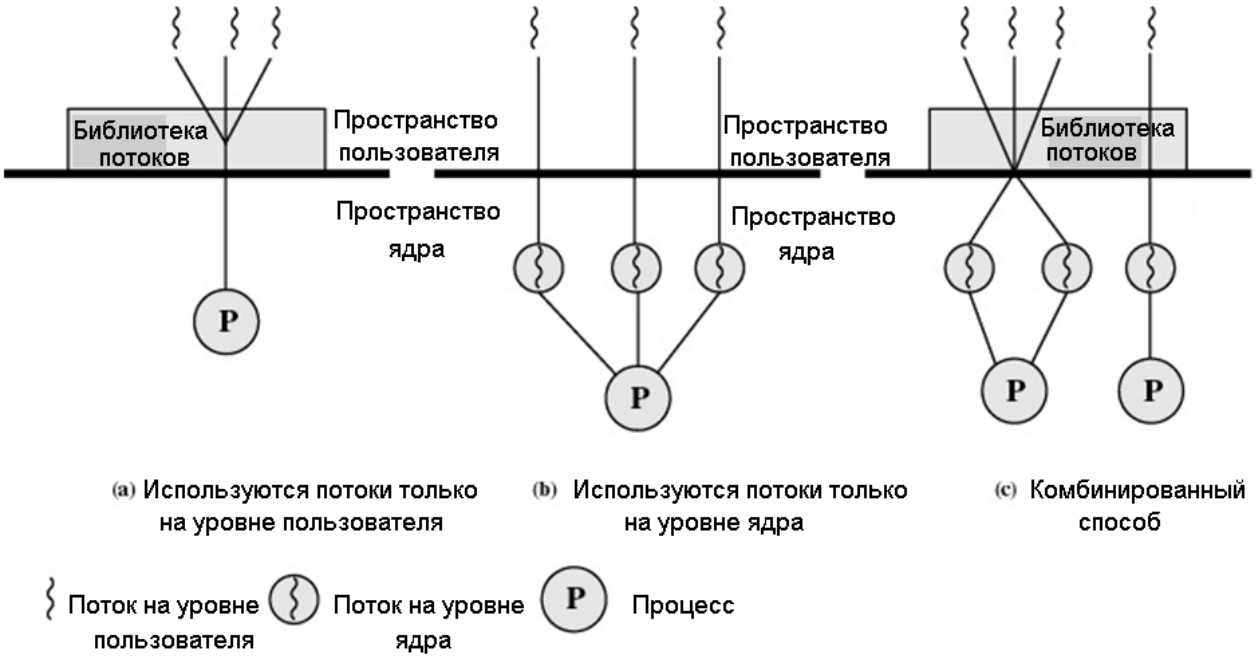


Рис 2.5. Потоки на уровне пользователя и ядра

2.4.2. Примеры реализации потоков

Рассмотрим простые примеры многопоточного программирования на языке С в среде ОС Linux [9]. В Linux реализована библиотека API-функций работы с потоками, которая называется *Pthreads*. Все функции и типы данных библиотеки объявлены в файле `<pthread.h>`. Эти функции не входят в стандартную библиотеку

языка C, поэтому при компоновке программы необходимо задавать опцию –lpthread в командной строке.

Для создания потока используется функция `pthread_create()`. Данной функции передаются следующие параметры:

- Указатель на переменную типа `pthread_t`, в которой сохраняется идентификатор нового потока. При ссылке на идентификаторы потоков в программах на C или C++ необходимо использовать тип данных `pthread_t`.
- Указатель на объект атрибутов потока. Этот объект определяет взаимодействие потока с остальной частью программы. Если задать его равным NULL, поток будет создан со стандартными атрибутами.
- Указатель на потоковую функцию. Функция имеет следующий тип: `void* (*) (void*)`
- Значение аргумента потока (тип void*). Данное значение передается потоковой функции.

При вызове программы Linux создает для нее новый процесс с единственным потоком, последовательно выполняющим программный код. Этот поток может создавать дополнительные потоки, которые находятся в одном процессе и выполняют части программного кода. Ниже приводится программа создания нового потока, который совместно с главным потоком изменяет разделяемую статическую глобальную переменную. Для вывода идентификатора выполняемого в текущий момент потока используется функция `pthread_self()`. Для того, чтобы родительский поток не завершился раньше дочернего, используется функция `pthread_join()`. Данная функция принимает два аргумента: идентификатор ожидаемого потока и указатель на переменную

void*, в которую будет записано значение, возвращаемое потоком. Если данное значение не важно, в качестве второго аргумента задается NULL.

/* thread1.c

* This program creates one new thread.
* New thread and main thread increase static shared variable on 1.
* Each thread in a process is identified by a thread ID.

to compile: gcc -o thread1 thread1.c -lpthread
http://www.intuit.ru/department/os/osintropractice/examples.html */

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

int a = 0;

void *mythread(void *dummy)
{
    pthread_t mythid;
    mythid = pthread_self();
    a = a+1;
    printf("Thread-1  %d,  Calculation  result  =  %d\n",
mythid, a);
    return NULL;
}

int main()
{
    pthread_t thid, mythid;
    int result;
```

```
    result = pthread_create( &thid, (pthread_attr_t
*)NULL, mythread, NULL);
    if(result != 0){
    printf ("Error on thread create,return value =
%d\n", result);
    exit(-1);
    }
    printf("Thread-1 created, Thread-1 id = %d\n",
thid);
    mythid = pthread_self();
    a = a+1;
    printf("Thread-main %d, Calculation result = %d\n",
mythid, a);
    pthread_join(thid, (void **)NULL);
    return 0;
}
```

Строка компиляции и компоновки имеет следующий вид:

```
gcc -o thread1 thread1.c -lpthread
```

В результате запуска программы на выполнение на экран выводится следующая информация:

Thread-1 1083714480, Calculation result = 1
Thread-1 created, Thread-1 id = 1083714480
Thread-main 1075320224, Calculation result = 2

Нижеследующий пример иллюстрирует асинхронность работы потоков в ОС Linux. Создаваемые два новых потока непрерывно записывает символы “Y” и “N” в стандартный поток ошибок. При достаточно больших выборках символы “Y” и “N” чередуются непредсказуемым образом.

```
/**********************************************************
thread2.c
This program creates two new threads, one to print Y's and the other to
print N's.
* Code listing from "Advanced Linux Programming," by CodeSourcery
LLC  *
* Copyright (C) 2001 by New Riders Publishing                    *
* Modified 2005 by Boris Ilyushkin
  to compile: gcc -o thread2 thread2.c -lpthread                  *
http://www.advancedlinuxprogramming.com
**********************************************************/

#include <pthread.h>
#include <stdio.h>

/* Parameters to print_function.  */

struct char_print_parms
{

  char character;          /* The character to print */

  int count;     /* The number of times to print it */
};

/* Prints a number of characters to  stderr,  as given by PARAMETERS,
which is a pointer to a  struct char_print_parms.  */

void* char_print (void* parameters)
{
   /* Cast the cookie pointer to the right type.  */
```

```
  struct     char_print_parms*     p     =     (struct
char_print_parms*) parameters;
  int i;
  pthread_t thread_id;
  thread_id = pthread_self();
  printf("\n Thread new id = %d\n",thread_id);
  for (i = 0; i < p->count; ++i)
    fputc (p->character, stderr);
  return NULL;
}

/* The main program.  */

int main ()
{
  int i;
  pthread_t thread1_id,thread_main_id;
  pthread_t thread2_id;
  struct char_print_parms thread1_args;
  struct char_print_parms thread2_args;

  /* Create a new thread to print 30 Y's.  */
  thread1_args.character = 'Y';
  thread1_args.count = 30;
  pthread_create   (&thread1_id,   NULL,   &char_print,
&thread1_args);
  printf("\nThread1 new created ");

  /* Create a new thread to print 30 N's.  */
  thread2_args.character = 'N';
  thread2_args.count = 30;
  pthread_create   (&thread2_id,   NULL,   &char_print,
&thread2_args);
  printf("\nThread2 new  created");
```

```
  i = 1;
  thread_main_id = pthread_self();
  printf("\nThread-main id = %d\n", thread_main_id);

  /* Make sure the first thread has finished.  */
  pthread_join (thread1_id, NULL);
  /* Make sure the second thread has finished.  */
  pthread_join (thread2_id, NULL);

  /* Now we can safely return.  */
  return 0;
}
```

В результате запуска программы на выполнение при заданных небольших выборках на экран выводится следующая информация:

YYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYNNNNNNNNNN
NNNNNNNNNNNNNNNNNNNN
Thread new id = 1083714480

Thread1 new created
Thread new id = 1092111280

Thread2 new created
Thread-main id = 1075320224

Некоторые языки программирования, например *Java*, *Modula-3* и другие, поддерживают создание потоков средствами самого языка. Рассмотрим в качестве примера реализацию потоков в языке *Java*. Особенностью языка *Java* является наличие встроенной поддержки многопоточного программирования [6]. Многопоточность языка *Java* позволяет осуществить многозадачность внутри одной

программы. В *Java* используется достаточно простая модель синхронизации потоков. Организация связи и контекстное переключение потоков контролируется исполняющей системой *Java*. Унифицированный многопоточный интерфейс поддерживается всеми виртуальными машинами *Java*. Многопоточность реализуется достаточно просто при разработке объектно-ориентированного приложения *Java*. Такое приложение состоит из набора объектов, взаимодействующих друг с другом. Каждый объект - независимый компонент, который может выполняться потоком и запускаться параллельно с другими объектами.

Когда *Java* приложение запускается на выполнение, то автоматически создается один поток, который называется главным (*main thread*), так как является единственным потоком, выполняющимся при запуске программы. Главный поток вызывает метод `main()`, обеспечивающий основную логику его выполнения. Метод `main()` является статическим, поскольку программа запускается только с одним `main()`. Для запуска новых потоков используется метод `run()`, который реализует основную логику выполнения потока и объявляется как `public void run()`.

Обеспечить реализацию метода `run()` можно двумя способами: путем расширения класса `Thread` и через интерфейс `Runnable`.

Класс `Thread` скрывает механизм запуска и контроля выполнения параллельного потока. Необходимо создать новый класс, который расширяет класс `Thread` ,а затем создать экземпляр этого класса. В расширенном классе нужно заменить метод `run()` класса `Thread`, который является точкой входа для нового потока. Для запуска потока нужно вызвать метод `start()`.

Интерфейс `Runnable` позволяет определить, что должен делать класс. Объявленные в нем методы не имеют тела. Назначение интерфейса - обеспечить динамический выбор метода по ходу выполнения

программы. Необходимо описать класс, который реализует интерфейс Runnable (выполняемый), а затем создать экземпляр данного класса. Интерфейс Runnable определяет только один абстрактный метод с именем run(), являющийся точкой входа потока. Поэтому, достаточно реализовать метод run(), внутри которого поместить операторы, которые должны выполняться в новом потоке. Далее необходимо создать экземпляр класса Thread, передав ему объект Runnable. Для поддержки интерфейса Runnable в ряд конструкторов класса Thread был введен отдельный параметр Runnable. Для запуска потока нужно вызвать метод start(). При выполнении поток будет вызывать метод run() объекта Runnable.

Структура программы, запускающей два потока, созданных обоими способами, выглядит следующим образом:

```
//Создание 1-го потока путем расширения класса Thread
  class One extends Thread {
// точка входа 1-го потока
  public void run() {
.................
  выполнение 1-го потока
.................
    }
  }

 //Создание 2-го потока путем реализации интерфейса Runnable
  class Two implements Runnable {
// точка входа 2-го потока
    • public void run() {
.................
  выполнение 2-го потока
.................
    }
  }
```

```
// запуск программы
 public class OneTwo {
 public static void main(String args[]) {
// создание экземпляров классов
 One c = new One() ;
 Runnable r = new Two() ;
 Thread t = new Thread(r) ;// передача объекта Runnable
                                  // классу Thread
.................
.................
// запуск потоков
 c.start() ;
 t.start() ;
  }
 }
```

Поток *Java* может находиться в одном из следующих состояний в течение периода существования:

- новый (*new thread*)
- исполняемый (*runnable or ready to run*)
- неисполняемый (*not runnable*)
- пассивный (*dead*)

Новый - поток создан, но еще не запущен. *Исполняемый* - поток запущен и готов продолжить выполнение, т.е. ему может быть выделено операционной системой процессорное время (когда процессор окажется свободным). Поток, которому выделено процессорное время, является выполняющимся (*running*). *Неисполняемый* - в данное состояние поток переходит после наступления определенного события (ожидание завершения операции ввода/вывода, перевод в неактивный режим на определенное время методом sleep(), вызов методов wait() или suspend(). Неисполняемый поток становится опять исполняемым при изменении его состояния (за-

вершен ввод/вывод, завершен период пребывания в неактивном режиме и т.д.). В течение периода своего существования поток может неоднократно переходить из состояния *исполняемый* в состояние *неисполняемый*.

Пассивный - поток становится пассивным, когда он завершается. Обычно поток становится пассивным, когда завершается его метод run(). Кроме того, поток может стать пассивным при вызове его методов stop() или destroy(). *Пассивный* поток является таковым постоянно, воскресить его невозможно.

Диспетчеризация или *планирование* (*scheduling*) потоков представляет собой механизм, используемый для определения способа выделения процессорного времени для исполняемых потоков. Для виртуальной машины *Java* исполняемый поток с наивысшим приоритетом всегда выбирается для выполнения. Механизм реализации потоков *Java* является *вытесняющим* (*preemptive*), т.е. диспетчер приостанавливает поток, если появится поток с более высоким приоритетом. Как выполняются потоки с одинаковым приоритетом, в спецификации *Java* не определено. Порядок их выполнения определяется операционной системой. В реализации *Java* на *Windows 95/98/NT/2000* используется базовый диспетчер потоков с выделением *квантов* времени (выделяется промежуток времени для каждого потока). В реализации *Java* на *Solaris* используется диспетчер потоков без выделения *квантов* времени. Поэтому при реализации *Java* приложений на конкретной платформе необходимо учитывать метод организации многозадачности в операционной системе. В случае диспетчера с выделением квантов времени, контекстное переключение будет выполняться даже при отсутствии добровольной передачи управления с использованием метода `yield()` или блокировки ввода-вывода. Соответственно поток с более высоким приоритетом использует больше процессорного времени. В случае диспетчера без

выделения квантов времени контекстного переключения потоков не будет. Поток с наибольшим приоритетом использует 100 % процессорного времени (в случае равных приоритетов потоки будут выполняться последовательно, один за другим). В данном случае для контекстного переключения можно использовать метод перевода потоков в неактивный режим на определенный период времени `sleep()` или метод явной передачи управления `yield()`.

Ниже приводится в качестве примера простая 2-х поточная программа с переключением потоков. При вызове метода `sleep()` используется механизм обработки исключений языка *Java*. Исключения представлены классами. Их можно обрабатывать при помощи конструкции `try() {…} catch() {…}`. Если в то время, пока поток спит, произойдет исключение класса `InterruptedException`, будет выполнено предложение `catch()`.

```
//  OneTwo.java

class One extends Thread {

    public void run() {
        for (int i=0; i<8; i++) {
            System.out.println("One");
            try {
                Thread.sleep(100);
            } catch (InterruptedException e) {}
        }
    }
 }

 class Two implements Runnable {

    public void run() {
        for (int i=0; i<8; i++) {
            System.out.println("Two");
            try {
                Thread.sleep(100);
            } catch (InterruptedException e) {}
        }
```

```
    }
}

public class OneTwo  {

    public static void main(String args[]) {
    One c = new One();
    Runnable r = new Two();
    Thread t = new Thread(r);
    Thread m = Thread.currentThread();
        System.out.println("Main thread : " + m);
        System.out.println("First thread :  " + c);
        System.out.println("Second thread :  " + t);

         c.start();
         t.start();
    }

}
```

```
javac  OneTwo.java      ; компиляция

java   OneTwo           ;  запуск на выполнение
```

Листинг результата

```
Main thread :  Thread[main,5,main]
First thread :  Thread[Thread-0,5,main]
Second thread :  Thread[Thread-1,5,main]
One
Two
One
Two
One
Two
One
Two
One
Two
One
Two
One
Two
One
Two
```

Метод `sleep()` используется для контекстного переключения потоков. Без него, например, на платформе *Solaris* переключения потоков не будет.

Класс ***`Thread`*** предназначен для создания нового потока. Он определяет следующие основные конструкторы:

```
Thread() ,
Thread(Runnable object) ,
Thread(Runnable object,String name) ,
Thread(String name) ,
```

где *name* - имя,присваиваемое потоку ;
object - экземпляр объекта ***`Runnable.`***

Если имя не присвоено, система сгенерирует уникальное имя в виде ***`Thread-N`***, где *N* - целое число.

Некоторые из методов управления потоками:

`void start()` - начинает выполнение потока;

`final void stop()` - заканчивает выполнение потока;

`static void sleep(long msec)` - прекращает выполнение потока на указанное количество миллисекунд;

`static void yield()` - заставляет поток передать ресурсы процессора другому потоку;

`final void suspend()` - приостанавливает выполнение потока;

`final void resume()` - возобновляет выполнение потока.

2.5. Межпроцессорное взаимодействие

Существует необходимость взаимодействия процессов. Даже в случае наличия независимых процессов, не предназначенных для выполнения общей задачи (пакетные задания, интерактивные сессии), операционная система должна решать вопросы конкурентного использования ресурсов. Например, два независимых приложения могут запросить доступ к одному диску или порту. Поэтому необходимо правильно организованное взаимодействие между процессами – IPC (Inter Process Communication) [2,3,5,6].

Параллельными называются такие процессы, которые одновременно находятся в активном состоянии. Два параллельных процесса могут быть *независимыми* (independent processes) либо *взаимодействующими* (cooperating processes) [2]. *Независимыми* являются процессы, множества переменных которых не пересекаются. Переменными могут быть файлы данных, области оперативной памяти и т.д. Независимые процессы не влияют непосредственно на результаты работы друг друга, однако могут явиться причиной задержек исполнения других процессов, так как вынуждены разделять ресурсы системы. *Взаимодействующие* процессы совместно используют некоторые переменные, и выполнение одного процесса может повлиять на выполнение другого. Взаимодействующие процессы могут либо конкурировать за получение доступа к ресурсам, либо сотрудничать для выполнения общей задачи. Например, запрос чтения данных с диска, направляемый процессом файловой системы процессу управления диском, должен ожидать записи этих данных в память. Для успешного взаимодействия процессов необходимы два механизма [6]:

- *Синхронизация*. Процессам, выполняющим общую задачу, нужно ЖДАТЬ друг друга и СИГНАЛИЗИРОВАТЬ друг другу.

- *Взаимное исключение*. Конкурирующие процессы должны ЖДАТЬ освобождения общего ресурса и СИГНАЛИЗИРОВАТЬ об окончании его использования.

Ресурсы, которые не допускают одновременного использования несколькими процессами, называются *критическими*, а те участки программ, в которых происходит обращение к критическим ресурсам, называются *критическими секциями* (critical sections) [2]. Для корректного выполнения конкурирующих процессов необходимо наличие механизма *взаимного исключения* (mutual exclusion), не позволяющего двум процессам одновременно использовать критический ресурс. Для реализации механизма взаимного исключения, а также для обмена данными между процессами, в операционной системе должны быть предусмотрены средства синхронизации взаимодействующих процессов. Например, если один процесс использует в данный момент критический ресурс, то все остальные процессы, которым нужен этот ресурс, должны ждать его освобождения. Реализация *взаимных исключений* может создать проблему *взаимной блокировки* или *тупика* (*deadlock*) [2,3]. Пусть имеются два процесса P1, P2 и два ресурса R1, R2. Предположим, что каждому процессу для выполнения некоторых функций требуется доступ к обоим ресурсам. Тогда возможно возникновение ситуации, когда ОС выделяет ресурс R1 процессу P2, а ресурс R2 процессу P1. В результате каждый процесс ожидает получения одного из двух ресурсов. При этом ни один из процессов не освобождает уже имеющийся у него ресурс, ожидая получения второго ресурса для выполнения требуемых функций. В результате процессы оказываются взаимно заблокированными. Реализация взаимных исключений может также создать проблему *зависания процесса* (starvation – «умирание от голода») [3,5,6]. Пусть имеется три процесса P1, P2, P3 , каждому из которых периодически требуется доступ к ресурсу R. Пусть P1 обладает ресурсом R, а процессы

P2 и P3 приостановлены. После выхода P1 из критической секции доступ к ресурсу будет получен одним из процессов P2, P3. Пусть ОС предоставила доступ к ресурсу R процессу P3. Пока ресурс занят процессом P3, доступ к ресурсу может снова потребоваться процессу P1. Может оказаться, что ОС снова предоставит доступ к ресурсу процессу P1. Пока ресурс занят процессом P1, процессу P3 может снова потребоваться доступ к ресурсу R. Таким образом, возможна ситуация, при которой процесс P2 никогда не получит доступ к требуемому ему ресурсу, несмотря на отсутствие взаимной блокировки.

2.5.1. Средства низкоуровневой синхронизации

Программный подход

Программный подход может быть реализован для параллельных процессов, которые выполняются как в однопроцессорной, так и в многопроцессорной системе с разделяемой основной памятью [3]. Обычно такой подход предполагает взаимоисключения на уровне доступа к памяти. Одновременный доступ (чтение и/или запись) к одной и той же ячейке основной памяти упорядочивается при помощи следующего механизма. Рассмотрим для простоты два процесса P0 и P1. Пусть зарезервирована глобальная ячейка памяти turn. Перед выполнением критического участка процесс (P0 или P1) проверяет содержимое turn. Если значение turn равно номеру процесса, то процесс может войти в критический участок. В противном случае он должен ждать, постоянно опрашивая значение turn до тех пор, пока оно не позволит ему войти в критический участок. Пусть также определен логический вектор flag, в котором flag[0] соответствует процессу P0, а flag[1] – процессу P1. Каждый процесс может ознакомиться с флагом другого процесса, но не может его изменить. Когда процессу

требуется войти в критический участок, он периодически проверяет состояние флага другого процесса до тех пор, пока тот не примет значение false, указывающее, что другой процесс покинул критический участок. Процесс немедленно устанавливает значение своего собственного флага равным true и входит в критический участок. По выходе из критического участка процесс сбрасывает свой флаг, присваивая ему значение false. Ниже приводится пример алгоритма Петерсона для двух процессов [3].

```
     boolean flag[2] ;
     int turn ;
     void P0()
     {
          while(true)
          {
               flag[0] = true ;
               turn = 1 ;
               while (flag[1] && turn = = 1)
     /* блокировка входа в критическую секцию для P0 */  ;
               flag[0] = false ;
     /*  остальной код     */ ;
          }
     }
void P1()
{
     while(true)
     {
          flag[1] = true ;
          turn = 0 ;
          while (flag[0] && turn = = 0)
/* блокировка входа в критическую секцию для P1 */  ;
          flag[1] = false ;
     /*  остальной код     */ ;
          }
     }
     void main ()
     {
          flag[0] = false ;
```

```
        flag[1] = false ;
/* приостановка выполнения основной программы
   и запуск   P0,P1  */
        parbegin (P0,P1) ;
    }
```

Условия *взаимоисключения* выполняются. Рассмотрим процесс P0. Если flag[0] = true, P1 не может войти в критическую секцию. Если P1 уже находится в критической секции, то flag[1] = true и для P0 вход в критическую секцию заблокирован [3]. В данном алгоритме взаимная блокировка предотвращается. Предположим, что P0 заблокирован в своем цикле while. Следовательно, flag[1] = true, а turn = 1. P0 может войти в критическую секцию только в том случае, когда либо flag[1] =false, либо turn = 0. Может быть три случая:

P1 не намерен входить в критическую секцию. Однако это невозможно, поскольку при этом выполнялось бы условие flag[1] =false.

P1 ожидает входа в критическую секцию. Это также невозможно, так как если turn =1 , то P1 способен войти в критическую секцию.

P1 циклически использует критическую секцию. Это также невозможно, поскольку P1 перед каждой попыткой входа в критическую секцию устанавливает значение turn = 0 , давая возможность доступа в критическую секцию процессу P0.

Алгоритм Петерсона обобщается на случай n процессов.

Аппаратный подход

Процессоры многих компьютеров поддерживают команды, которые за один цикл выборки *атомарно* выполняют над ячейкой па-

мяти два действия: чтение и запись или чтение и проверка значения. *Атомарность* означает, что операция неделима, т.е. никакой другой процесс не может обратиться к слову памяти, пока команда не выполнена. В качестве примера рассмотрим команду TSL (Test and Set Lock – проверить и заблокировать), которая действует следующим образом [5].

```
TSL RX,LOCK  ; содержимое слова памяти lock копируется в
             ; регистр rx , значение lock устанавливается
             ; равным 1
```

Процессор, выполняющий команду TSL, блокирует шину памяти, чтобы остальные процессоры не могли обратиться к памяти. Таким образом, гарантируется, что операция считывания слова и записи значения неделима. Ниже приводится код программы с использованием TSL на типовом ассемблере для поддержки *взаимного исключения* [5].

```
enter_region ;
TSL REGISTER,LOCK
 CMP REGISTER,0          ;предыдущее значение lock сравнивается
                         ; с нулем
 JNE enter_region    ; если оно не нулевое – блокировка уже
                     ; была установлена и цикл завершается
 RET                 ; если оно нулевое – установка блокировки,
                     ; т.е. lock:=1, и возврат в вызывающую программу.
                     ; Процесс вошел в критический участок

leave_region ;
MOVE LOCK, 0         ; снятие блокировки, т.е. lock:=0.
                     ; Процесс вышел из критической секции.
RET
```

Совместно используемая переменная lock управляет доступом к общим данным. Если lock=0, любой процесс может изменить значение lock, т.е. lock:=1 и обратиться к общим данным, а затем изменить его обратно, т.е. lock:=0. Прежде чем попасть в критическую секцию, процесс вызывает процедуру enter_region, которая выполняет активное ожидание вплоть до снятия блокировки (когда другой процесс, находившийся в критической секции, вышел из нее), после чего она устанавливает блокировку и возвращает управление процессу, который входит в критическую секцию. По выходе из критической секции процесс вызывает процедуру leave_region, снимающую блокировку.

Взаимное исключение для доступа к общим данным может быть реализовано аппаратно путем [6]:

- запрета прерываний в однопроцессорной системе;
- использования аппаратно-поддерживаемой команды.

Запрет прерываний может быть обеспечен использованием примитивов, определенных ядром для запрета и разрешения прерываний.

Недостатком аппаратного подхода и алгоритма Петерсона является наличие цикла активного ожидания входа в критическую секцию, расходующего процессорное время.

Использование механизмов операционной системы

Фундаментальный принцип взаимодействия двух или нескольких процессов заключается в использовании обмена сигналами [3]. Для сигнализации используются специальные переменные, называемые *семафорами*. Для передачи сигнала через семафор s процесс выполняет примитив signal(s) , а для получения сигнала – примитив wait(s). В последнем случае процесс приостанавливается до тех пор, пока не осуществится передача соответствующего сигнала.

Будем рассматривать *семафор* как переменную, принимающую целые значения, над которой определены три операции:

1. Семафор инициализируется неотрицательным значением.
2. Операция `wait` уменьшает значение семафора. Если это значение становится отрицательным, процесс, выполняющий операцию `wait` , блокируется.
3. Операция `signal` увеличивает значение семафора. Если это значение становится отрицательным, то заблокированный операцией `wait` процесс деблокируется.

Ниже приводится алгоритм определения семафорных примитивов [3].

```
struct semaphore {
     int count ;
     queueType  queue ;
}
void wait (semaphore s)
{
     s.count-- ;
     if  (s.count < 0)
     {
     Поместить процесс P в s.queue и заблокировать его
     }
}
void signal (semaphore s)
{
     s.count++ ;
     if (s.count < = 0)
     {
     Удалить процесс P из s.queue и разблокировать его,
      т.е. поместить P  в список активных процессов.
     }
}
```

Семафор, который может принимать только значения 0 или 1 называется *бинарным семафором*. Для хранения процессов, ожидающих как обычные, так и бинарные семафоры, используется очередь[3]. При использовании принципа FIFO (first-in-first-out – “первым вошел - первым вышел”) первым из очереди освобождается процесс, который был заблокирован дольше других. Ниже приводится алгоритм решения задачи взаимоисключений с использованием семафора [3].

```
const int n = N        /* количество процессов */
semaphore s = 1 ;
void P(int i)
{
     while(true)
     {
          wait(s);
     /*   критическая секция    */ ;
          signal(s);
     /*  остальной код    */ ;
     }
}
void main()
{
parbegin (P(1),P(2), … ,P(n)); // запуск P1,P2,...,Pn
}
```

Пусть имеется массив `P(i),i=1,...,n` процессов. Каждый процесс перед входом в критическую секцию выполняет вызов `wait(s)`.Если `s<0`, процесс приостанавливается. Если `s=1`, то `s:=0` и процесс входит в критическую секцию. Поскольку `s` не является больше положительным, любой другой процесс при попытке войти в критическую секцию обнаружит, что она занята. При этом данный процесс блокируется и `s:=-1`. Пытаться войти в критическую сек-

цию может любое количество процессов, при этом значение семафора каждый раз уменьшается на единицу. После того как процесс, находившийся в критической секции, покидает ее, `s:=s+1` и один из заблокированных процессов удаляется из очереди и активизируется, т.е. перемещается из списка приостановленных процессов в список готовых к выполнению процессов. Данный процесс сможет войти в критическую секцию, как только планировщик операционной системы выберет его.

В случае потоков, выполняющихся в пространстве пользователя, используется упрощенная версия семафора, называемая *мьютексом* (mutex, сокращение от mutual exclusion – взаимное исключение) [5].

Мьютекс может управлять взаимным исключением доступа к совместно используемым ресурсам, и его реализация более проста и эффективна.

Мьютекс – переменная, которая может находиться в одном из двух состояний: *блокированном* (`<>0`) или *неблокированном* (`=0`). Значение мьютекса устанавливается двумя процедурами. Для входа в критическую секцию поток или процесс вызывает процедуру `mutex_lock`. Если мьютекс не заблокирован, поток может войти в критическую секцию [5]. В противном случае вызывающий поток блокируется до тех пор, пока другой поток, находящийся в критической секции, не выйдет из нее, вызвав процедуру `mutex_unlock`. Если мьютекс блокирует несколько потоков, то из них случайным образом выбирается один. Ниже приводятся коды процедур `mutex_lock` и `mutex_unlock` в случае потоков на уровне пользователя с использованием команды TSL [5].

```
mutex_lock ;
TSL REGISTER,MUTEX      ; предыдущее значение mutex
                        ; копируется в регистр и устанавливается
```

```
                              ; новое значение mutex:=1
  CMP REGISTER,0              ; предыдущее значение mutex сравнивается
                              ; с нулем
   JZE   ok                   ; если оно нулевое – mutex не был блокирован
  CALL thread_yield           ; mutex занят, управление передается
                              ; другому потоку
   JMP mutex_lock             ; повторить попытку позже
ok: RET                       ; возврат в вызывающую программу
                              ;  вход в критический участок

mutex_unlock ;
   MOVE MUTEX,0               ; снятие блокировки, т.е. mutex:=0 и
                              ; выход из критической секции
   RET                        ; возврат
```

Процедура `mutex_lock` отличается от вышеприведенной процедуры `enter_region` следующим образом. Процедура `enter_region` находится в состоянии активного ожидания, т.е. проверяет в цикле наличие блокировки до тех пор, пока критическая секция не освободится или планировщик не передаст управление другому процессу по истечении кванта времени. Однако в случае потоков ситуация меняется, поскольку нет прерываний по таймеру, останавливающих слишком долго работающие потоки [5]. Поток, пытающийся получить доступ к семафору и находящийся в состоянии активного ожидания, зациклится навсегда, так как он не позволит предоставить процессор другому потоку, желающему снять блокировку. Поэтому, если войти в критическую секцию невозможно, `mutex_lock` вызывает `thread_yield`, чтобы предоставить процессор другому потоку. При следующем запуске поток снова проверяет блокировку. Поскольку вызов `thread_yield` является обращением к планировщику потоков в пространстве пользователя, об-

ращения к ядру не требуется, и он выполняется быстро. Синхронизация потоков на уровне пользователя происходит полностью в пространстве пользователя [5,6].

2.5.2. Средства высокоуровневой синхронизации

Средства высокоуровневой синхронизации разрабатываются для упрощения написания программ и обычно являются структурными компонентами языка программирования.

Монитор представляет собой набор процедур, переменных и других структур данных, объединенных в программный модуль, который обеспечивает функциональность, эквивалентную функциональности семафора [3,5]. Мониторы реализованы во множестве языков программирования, включая *Pascal-Plus*, *Modula-3*, *Java*. *Мониторы* также реализуются как программные библиотеки. Это позволяет их использовать для блокировки любых объектов [3].

Основными характеристиками *монитора* являются следующие характеристики:

- локальные переменные *монитора* доступны только его процедурам и не доступны внешним процедурам;
- процесс входит в *монитор* путем вызова одной из его процедур;
- в *мониторе* может находиться только один активный процесс и любой другой процесс, вызвавший *монитор*, будет приостановлен в ожидании доступности *монитора*.

Если в монитор поместить совместно используемые структуры данных, то будет обеспечено взаимное исключение при обращении к данным. Для применения в параллельных вычислениях мониторы должны включать инструменты синхронизации. Рассмотрим реализацию мониторов в языке *Java* [6]. Возможность синхронизации в *Java* реализована в виде ключевого слова synchronized, указывающего на необходимость получения монопольной блокировки, и трех методов

условной синхронизации, wait(), notify(), notifyall(), определенных в классе Object и поэтому наследуемых любым другим классом. В каждый объект Java встроена возможность монопольной блокировки и одна условная переменная. Если объявить метод объекта как synchronized , для его выполнения вызывающему коду необходимо получить монопольную блокировку объекта. У каждого объекта в *Java* имеется свой собственный неявный *монитор*. Когда метод типа synchronized вызывается для объекта, происходит обращение к *монитору* объекта чтобы определить, выполняет ли в данный момент какой-либо другой поток метод типа synchronized для данного объекта. Если нет, то текущий поток получает разрешение войти в *монитор*. Вход в *монитор* называется также *блокировкой* (*locking*) *монитора*. Если при этом другой поток уже вошел в *монитор*, то текущий поток должен ожидать до тех пор, пока другой поток не покинет *монитор*. Таким образом, монитор *Java* вводит поочередность в параллельную обработку. Этот способ называется также преобразованием в последовательную форму (*serialization*). Объявление метода synchronized не подразумевает, что только один поток может одновременно выполнять этот метод, как в случае критического участка. В любой момент времени только один поток может вызвать этот метод (или любой другой метод типа synchronized) для конкретного объекта. Таким образом, мониторы *Java* связаны с объектами, но не с блоками кода. Два потока могут параллельно выполнять один и тот же метод типа *synchronized* при условии, что этот метод вызван для разных объектов. Мониторы не являются объектами языка *Java*, у них нет атрибутов или методов. Доступ к мониторам возможен на уровне собственного кода *JVM*.

В *Java* есть два способа синхронизации потоков.

1. Создание синхронизирующего метода внутри класса

Используется при наличии метода или группы методов, обрабатывающих внутреннее состояние объекта в многопоточной среде. Для организации последовательного доступа потоков к методу объявление метода предваряется ключевым словом `synchronized`.

```
class Callme{
synchronized void call(String msg){
  . . . . . .
  . . . . . .
  }
```

Для условной синхронизации потоков предназначены три стандартных метода [6]:

- `wait()` – вызов данного метода в теле метода, объявленного как `synchronized`, приводит к снятию блокировки объекта, блокированию текущего потока и установке его в очередь потоков, ожидающих у данного объекта;
- `notify()` - вызов данного метода в теле метода, объявленного как `synchronized`, приводит к пробуждению одного из потоков, ожидающих у данного объекта;
- `notifyAll()` – подобен методу `notify()` ,но вызывает пробуждение всех потоков, ожидающих у данного объекта.

Поток, выполняющий синхронизированный метод некоторого объекта и достигший точки, в которой ему требуется условная синхронизация, вызывает метод `wait()` этого объекта. В результате выполнение потока приостанавливается и он включается в набор ожидающих потоков, связанных с данным объектом. Для выполнения ожидаемого потоком условия необходимо перевести в определенное состояние соответствующие данные. Ожидающий поток не может их изменить, так как заблокирован. Это должен сделать другой поток, который затем выполнит вызов `notify()` или `notifyAll()`. Метод `notify()` инициирует выбор одного из потоков, ожидающих у данного объекта, и переводит выбранный поток в состояние готовности.

Выбранный поток начинает соревноваться с другими потоками за возможность продолжить выполнение синхронизированного метода, из которого он был переведен в состояние ожидания. Получив доступ к этому методу, он продолжит его выполнение с инструкции, следующей непосредственно за вызовом метода `wait()`. Если включить вызов `wait()` в цикл `while`, поток сможет повторно проверить условие, прежде чем войдет в критический участок.

Метод `notify()` используется, когда нужно активизировать только один из ожидающих потоков, причем не важно, какой именно.

Метод `notifyAll()` применяется, если необходимо активизировать множество потоков или важно выбрать конкретный поток. После его выполнения будут активизированы все потоки, ожидающие у данного объекта.

2. Создание синхронизирующего блока

Используется для организации доступа к объектам класса, не разработанного для многопоточного доступа (например, к массивам) или созданного другим программистом (нет доступа к исходному коду). Необходимо поместить вызовы методов в синхронизирующий блок путем использования оператора `synchronized`, имеющего следующий синтаксис:

```
    synchronized (object){
// операторы, которые необходимо синхронизировать
          }
```

Оператор `synchronized` полезен при непосредственном изменении общих переменных объекта.

Пример

```
void call(SomeClassobj){
  synchronized(obj){
    obj.variable=5;
```

```
    }
  }
```

Следует учесть, что синхронизированные методы выполняются медленнее несинхронизированных. В качестве примера использования синхронизированных методов внутри класса рассмотрим нижеприведенный код [6].

```
// BufferExample.java
```

// Координация потоков, записывающих в буфер и читающих из буфера
// с использованием методов wait() и notifyAll()

// Класс, представляющий буфер для хранения одного значения

```
class Buffer {
private int value = 0;
private boolean full = false;  // буфер пуст

public synchronized void put(int a)
    throws InterruptedException {

  while (full)
  wait();

  value = a;
  full  = true;   // буфер полон

  notifyAll();
 }
public synchronized int get()
    throws InterruptedException {

  int result;
  while (!full)
  wait();

  result = value;
  full  = false;

  notifyAll();
  return result;
 }
```

```
    }
```

// Класс, представляющий поток записи в буфер
// Поток пытается по очереди поместить в буфер значения 10,20,30,40

```
    class PutThread extends Thread {
    Buffer buf;

    public PutThread(Buffer b) {buf = b;}

    public void run() {
      try {
        buf.put(10);
    System.out.println(" buf = 10");
        buf.put(20);
    System.out.println(" buf = 20");
        buf.put(30);
    System.out.println(" buf = 30");
        buf.put(40);
    System.out.println(" buf = 40");
      } catch (InterruptedException e) {
   System.out.println("Поток прерван и завершил работу,");
   System.out.println("не закончив запись в буфер");
    }
     }
       }
```

// Класс, представляющий поток чтения из буфера
// Поток пытается прочитать из буфера и вывести 12 значений

```
    class GetThread extends Thread {
    Buffer buf;

    public GetThread(Buffer b) {buf = b;}

    public void run() {
      try {
        for (int l = 0; l<12; l++) {
         System.out.println(buf.get());
        }
      } catch (InterruptedException e) {
   System.out.println("Поток прерван и завершил работу,");
   System.out.println("не закончив чтение из буфера");
```

```
        }
      }
         }
```

// Класс, представляющий метод `main`, который создает буфер,
// три записывающих в него потока и один читающий из него поток
// и запускает потоки

```
     public class BufferExample {

     public static void main(String[] args) {
     Buffer buf = new Buffer();

     Thread p1 = new PutThread(buf);
     Thread p2 = new PutThread(buf);
     Thread p3 = new PutThread(buf);
     Thread g1 = new GetThread(buf);

// запуск потоков

     p1.start();
     p2.start();
     p3.start();
     g1.start();

      }
     }
```

javac `BufferExample.java` ; компиляция

java `BufferExample` ; запуск на выполнение

Листинг результата

```
buf = 10
buf = 20
  10
buf = 10
  20
buf = 10
  10
buf = 30
```

```
  10
buf = 20
  30
buf = 20
  20
  20
buf = 40
  40
buf = 30
  30
buf = 40
  40
  30
buf = 30
  40
buf = 40
```

Объект класса `Buffer` представляет собой буфер для хранения единственного значения. Экземпляры класса `PutThread` записывают в буфер числа 10, 20, 30, 40, а экземпляры класса `GetThread` пытаются прочесть из буфера 12 чисел и вывести их на экран. Программа `BufferExample` создает объект-буфер, три объекта `PutThread`, один объект `GetThread` и запускает все четыре потоковых объекта на выполнение. Рассмотрим схему работы класса `BufferExample` [6]. Если первым будет выполняться поток `p1`, последовательность действий может быть следующей:

- Поток `p1` вызывает метод `put(10)` и получает доступ к этому методу, так как методы `get()` и `put()` данного объекта `Buffer` не используются никаким другим потоком.
- Метод `put(10)` проверяет флаг `full` и так как `full=false`, записывает в буфер значение `10`, после чего устанавливает флаг `full` в `true` и вызывает метод `notifyAll()`. Поскольку ожидающих потоков нет, вызов `notifyAll()` не производит должного эффекта. Выполнение метода `put()` завершается,

поток `p1` снимает с буфера взаимоисключающую блокировку и продолжает выполняться.

- Предположим, что `p1` снова вызывает метод `put(20)`. Поскольку теперь `full=true`, потоку `p1` приходится вызывать метод `wait()`, чтобы дождаться освобождения буфера. Вызов `wait()` блокирует поток `p1` у входа в буфер и снимает блокировку с буфера для входа другого потока.
- Допустим, что следующим начинает выполняться поток `g1`. Он вызывает метод `get()`, блокирует буфер для входа другого потока, и поскольку буфер не пуст, считывает из него значение `10`. Затем меняет флаг `full` на `false` и вызывает метод `notifyAll()`. После этого освобождается ждавший у буфера поток `p1`.
- Поток `p1` не может сразу продолжить выполнение метода `put(20)`, так как буфер еще заблокирован. Возвращаясь из метода `get()`, поток `g1` снимает блокировку с объекта-буфера, и теперь поток `p1` может продолжить выполнение `put(20)`.

Данная схема описывает лишь один из возможных вариантов развития событий. Последовательность выполнения потоков зависит от планировщика, и для разных реализаций виртуальной машины Java может быть различной.

В качестве следующего примера приведем классический тестовый код вычисления числа **π** на *Java*.

```
/******
*     Pi.java
*
*     compute pi by integrating f(x) = 4/ (1 + x**2)
*
*     each thread:
*     - receives the interval used in the approximation
```

```
*      - calculates the areas of its rectangles
*      - synchronizes for a global summation
*      Process 0 prints the result and the time it took
******/

class Pi extends Thread
{
    int from,to ;
 static int nn,n=72000000,np=4; // np - number of processors
   double h,sum,x;
    static double ssum = 0.0 ;
    static int j ;
    static long st,end ;

    public Pi(int from, int to)
    {
        this.from = from;
        this.to = to;
    }

    public double f(double a)

    {
        return(4.0 / (1.0 + a*a));
    }

// synchronization

    synchronized void count()
    {
       j = j + 1;
       ssum += h * sum ;
  System.out.println("ssum == " + ssum);
       if ( j == np )
  System.out.println("ssum-pi="+(ssum-Math.PI));
     }

    public void run()
    {
     st = System.currentTimeMillis();     // start
          System.out.println(this);
       h = 1.0 / (double) n;
       sum = 0.0;

       for (int i=from; i<to; i++)
```

```
          {
 //        System.out.println("i== " + i);

             x = h * ((double) i - 0.5);
             sum += f(x);
          }
        count();
        end = System.currentTimeMillis();    // end
      System.out.println("Time == " + (end - st));

    }

    public static void main(String[] args)
   {
        j = 0;

// spawn   np   threads, each of  wich calculates one area

        for (int i=0; i<np; i++)
        {
            Pi t = new Pi(i*n/np, (i+1)*n/np);
            t.start();
        }
//System.out.println("ssum-pi="+(ssum-Math.PI));
    }
}
```

Листинг результата

```
Thread[Thread-0,5,main]
Thread[Thread-1,5,main]
Thread[Thread-2,5,main]
Thread[Thread-3,5,main]
ssum == 0.9799146557752786
Time == 2140
ssum == 1.8545904471141887
Time == 2453
ssum == 2.574004455173001
Time == 2687
ssum == 3.1415926813673596
ssum - pi = 2.777756646921148E-8
Time == 2828
```

Данные результаты получены на процессоре AMD Athlon 1.47 ГГц. Суммарное время счета равно сумме времен для каждого потока в миллисекундах. В случае многопроцессорной архитектуры (в данном случае 4-х процессорной) потоки распараллеливаются на разных процессорах и время счета будет существенно ниже.

Глава 3. Управление памятью

В однозадачных ОС основная память разделяется на *системную память*, используемую операционной системой и *пользовательскую память*, используемую выполняющейся программой. В многозадачных ОС пользовательская часть памяти должна распределяться между несколькими процессами. Основными функциями ОС по управлению памятью являются:

- выделение памяти процессам и освобождение памяти по завершении процессов;
- перемещение кодов и данных процессов из основной памяти на диск для оптимизации загрузки процессора;
- трансляция виртуальных адресов программы в реальные физические адреса;
- защита кодов и данных процессов от нежелательного воздействия других процессов;
- обеспечение возможности совместного доступа нескольких процессов к одному сегменту памяти.

Совокупность виртуальных адресов процесса называется *виртуальным* адресным пространством. *Виртуальные* или *логические* адреса генерируются компилятором, переводящим исходную программу в машинный код. Преобразование виртуальных адресов в физические адреса осуществляется операционной системой и аппаратным обес-

печением, *диспетчером памяти* (**MMU – Memory Management Unit**). Содержимое назначенного процессу виртуального адресного пространства, т.е. коды команд, исходные и промежуточные данные, а также результаты вычислений, представляют собой *образ процесса* [1]. Для хранения образа процесса (или его части) используется внешняя память на жестких дисках. На этом принципе основан *механизм виртуальной памяти*, используемый в операционных системах для управления памятью. В настоящее время *механизм виртуальной памяти* реализуется тремя основными способами [1]:

- *сегментная виртуальная память* предусматривает перемещение данных между основной и внешней памятью сегментами произвольного размера;
- *страничная виртуальная память* предусматривает перемещение данных между основной и внешней памятью страницами фиксированного размера;
- *сегментно-страничная виртуальная память* использует двухуровневое деление виртуального адресного пространства сначала на сегменты, а затем на страницы.

3.1. Сегментный способ организации виртуальной памяти

Сегментный способ организации виртуальной памяти исторически реализован был первым. Виртуальное адресное пространство процесса разбивается на логические части – сегменты. Отдельный сегмент может представлять собой программный модуль, массив данных и так далее. Деление виртуального адресного пространства на сегменты осуществляется компилятором с учетом введенных программных инструкций или по умолчанию в соответствии с принятыми соглашениями. Максимальный размер сегмента определяется разрядностью виртуального адреса. Для 32-разрядных процессоров он

равен 4 Гбайт. Размер максимально возможного виртуального адресного пространства процесса равен суммарному размеру составляющих его сегментов максимального размера.

Каждый сегмент имеет соответствующую информационную структуру, называемую *дескриптором сегмента* [2]. Для каждого процесса операционная система создает *таблицу дескрипторов сегментов*, в которой для каждого сегмента указывается его текущее расположение (в основной или дисковой памяти), базовый адрес, размер, тип, уровень привилегий, права доступа и так далее.

Рассмотрим в качестве иллюстрации способ организации виртуальной памяти 32-разрядного процессора семейства Intel Pentium [7]. Основу виртуальной памяти в защищенном режиме процессора Intel Pentium составляют таблицы дескрипторов. Наиболее важной является таблица глобальных дескрипторов **GDT** (Global Descriptor Table), которая содержит информацию о системных сегментах, включая саму операционную систему. Сегменты, описываемые глобальными дескрипторами, доступны всем задачам, выполняемым процессором. Кроме того, в памяти могут находиться таблицы локальных дескрипторов **LDT** (Local Descriptor Table) для каждой выполняемой программы, описывающие сегменты кода, данных, стека и так далее. К сегментам **LDT** может обращаться только та задача, в которой эти дескрипторы описаны.

В *реальном режиме* работы микропроцессора поддерживается выполнение всего одной программы. Поэтому для реального режима достаточно простых механизмов распределения оперативной памяти. Достаточно знать адреса, по которым располагаются сегменты кода, данных и стека, при этом размер сегмента не должен превышать 64 Кбайт и аппаратные средства контроля доступа к сегменту отсутствуют.

В защищенном режиме работы микропроцессора может одновременно выполняться несколько программ, и поэтому встает вопрос об их защите от взаимного влияния. Если каждому сегменту программы присвоить определенные атрибуты, то часть контроля над доступом к ним может осуществлять микропроцессор. В число основных атрибутов входят следующие атрибуты [7]:

- расположение сегмента в памяти;
- размер сегмента;
- уровень привилегий – определяет права данного сегмента относительно других сегментов;
- тип доступа – определяет назначение сегмента.

Таким образом, в защищенном режиме микропроцессор поддерживает два типа защиты – по привилегиям и доступу к памяти.

На рис. 3.1 представлена 8-байтовая структура дескриптора, содержащая следующие поля:

- **Limit_1** - младшие биты 0 – 15 20-разрядного поля границы сегмента, определяющего размер сегмента в единицах, определяемых битом гранулярности G
- **Base_1** - биты 0 – 15 32-разрядной базы сегмента, которая определяет значение линейного адреса начала сегмента в памяти
- **AR** - байт, поля которого определяют следующие права доступа к сегменту :
  - **A** - бит доступа (Accessed) к сегменту. Устанавливается аппаратно при обращении к сегменту
  - **R** - для сегментов кода – бит доступа по чтению (Readable); определяет, возможно ли чтение из сегмента кода при осуществлении замены префикса сегмента: 0 – чтение запрещено; 1 – чтение разрешено

- **W** - для сегментов данных – бит записи : 0 – модификация данных в сегменте запрещена; 1 – модификация данных в сегменте разрешена
- **C** - для сегментов кода – бит подчинения (Conforming): 1 – подчиненный сегмент кода; 0 – обычный сегмент кода
- **ED** - для многозадачного режима определяет особенности смены значения текущего уровня привилегий. Для сегментов данных – бит расширения вниз (Expand Down); служит для различения сегментов стека и данных: 0 – сегмент данных, 1 – сегмент стека
- **I** - бит предназначения (Intending): 0 – сегмент данных или стека; 1 – сегмент кода
- **DPL** - поле уровня привилегий сегмента (Descriptor Privilege Level). Содержит численное значение в диапазоне от 0 до 3. Самым привилегированным является уровень 0.
- **P** - бит присутствия (Present): 0 – сегмента нет в основной памяти; 1 – сегмент находится в основной памяти

- **Limit_2** - старшие биты 16 – 19 20- разрядного поля границы сегмента
- **U** - бит пользователя (User). Используется по усмотрению программиста: 0 – бит не используется
- **D** - бит разрядности операндов и адресов: 0 – используются 16-разрядные операнды и режимы 16-разрядной адресации; 1 – используются 32-разрядные операнды и режимы 32-разрядной адресации
- **G** - бит гранулярности: 0 – размер сегмента равен значению в поле **Limit** в байтах; 1 – размер сегмента равен значению в поле **Limit** в страницах
- **Base_2** - биты 16 – 23 32-разрядной базы сегмента
- **Base_3** - биты 24 – 31 32-разрядной базы сегмента

Base_3							**Limit_2**			**Байт AR**				**Type**				**Base_2**		
63	…	56	55	54	53	52	51	…	48	47	46	45	44	43	42	41	40	39	…	32
Базовый адрес сегмента (24-31)			**G**	**D**	**O**	**U**	Размер сегмента (16-19)			**P**	**DPL**		**S**	**I** E	**C ED**	**R W**	**A**	Базовый адрес сегмента (16-23)		
Base_1										**Limit_1**										
Базовый адрес сегмента (0-15)										Размер сегмента (0-15)										
31	…								16	15	…									0

Рис.3.1. Структура дескриптора сегмента процессора Intel Pentium

Граница сегмента (**Limit**) представляет собой номер последнего байта сегмента. Поле границы состоит из 20 бит и разбито на две части. Если бит гранулярности G=0, то граница указывается в байтах и максимальный размер сегмента равен 1 Мбайт. Если G=1, то граница указывается в страницах размером 4Кбайт и максимальный размер сегмента равен 4Гбайт (1М страниц). База сегмента (**Base**) определяет начальный 32-битовый линейный адрес сегмента в адресном пространстве процессора. Выведение информации о базовом адресе сегмента и его размере на уровень микропроцессора позволяет аппаратно контролировать работу программ с памятью и предотвращать обращения по несуществующим адресам. Поле **DPL** служит для защиты программ друг от друга. Программам операционной системы обычно назначается уровень 0 (максимальные привилегии), прикладным программам назначается уровень 3 (минимальные привилегии), в резуль-

тате чего исключается возможность разрушения операционной системы некорректными программами. Поле **Type** типа сегмента определяет целевое назначение сегмента. Типичными комбинациями битов, входящих в состав поля типа сегмента являются следующие [7]:

- 000 - сегмент данных, только для чтения;
- 001 - сегмент данных с разрешением чтения и записи;
- 011 - сегмент стека с разрешением чтения и записи;
- 100 - сегмент кода с разрешением только выполнения;
- 101 - сегмент кода с разрешением выполнения и чтения из него.

Дескриптор сегмента помещается в одну из дескрипторных таблиц. Обращение к сегментам в защищенном режиме возможно только через дескрипторы этих сегментов [8]. Для этого в один из сегментных регистров заносится *селектор дескриптора*, в состав которого входит номер (индекс) соответствующего сегменту дескриптора. Во время выполнения программы регистр CS содержит селектор для сегмента кода, регистр DS содержит селектор для сегмента данных. Процессор по этому номеру находит нужный дескриптор в локальной или глобальной таблице и сохраняет его в микропрограммных рабочих регистрах для обеспечения быстрого доступа к нему с целью извлечения базового адреса сегмента. Формат селектора дескриптора представлен на рис. 3.2, где:

- **RPL** - поле запрашиваемого уровня привилегий сегмента (Requested Privilege Level). Содержит численное значение в диапазоне от 0 до 3.
- **TI** - поле индикатор таблицы дескрипторов (Table Index). Определяет, в какой таблице нужно искать соответствующий дескриптор, т.е. является ли данный сегмент локальным или глобальным. Если TI=0, то выбирается таблица глобальных деск-

рипторов сегментов **GDT**. Если TI=1, то выбирается таблица локальных дескрипторов сегментов **LDT** текущей задачи.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Номер дескриптора													**TI**	**RPL**	

Рис.3.2. Селектор дескриптора

Виртуальный адрес при этом способе задания состоит из двух полей: номера сегмента и смещения относительно начала этого сегмента. На рис. 3.3 показано обращение к ячейке, виртуальный адрес которой равен сегменту с номером 10 и смещением от начала этого сегмента, равным 624. Поскольку бит присутствия P=1, сегмент находится в оперативной памяти и операционная система расположила данный сегмент, начиная с ячейки с номером 256000. В поле размер сегмента указывается количество адресуемых ячеек памяти. Данное поле используется, в том числе и для контроля обращений кода выполняющейся задачи в пределах текущего сегмента. Для формирования исполнительного адреса 32-разрядное поле **Base** прибавляется к смещению, и таким образом формируется 32-разрядный линейный адрес 256624, который интерпретируется как физический адрес и посылается в память для чтения или записи.

При передаче управления задаче ОС заносит в соответствующий системный регистр адрес таблицы дескрипторов сегментов данной задачи. Таблица дескрипторов также представляет собой сегмент данных, который обрабатывается диспетчером памяти. Таким образом, появляется возможность размещать в основной памяти только те сегменты задачи, к которым в настоящий момент происходит обра-

щение, что, в свою очередь, позволяет разместить в основной памяти больше задач и увеличить загрузку вычислительной системы.

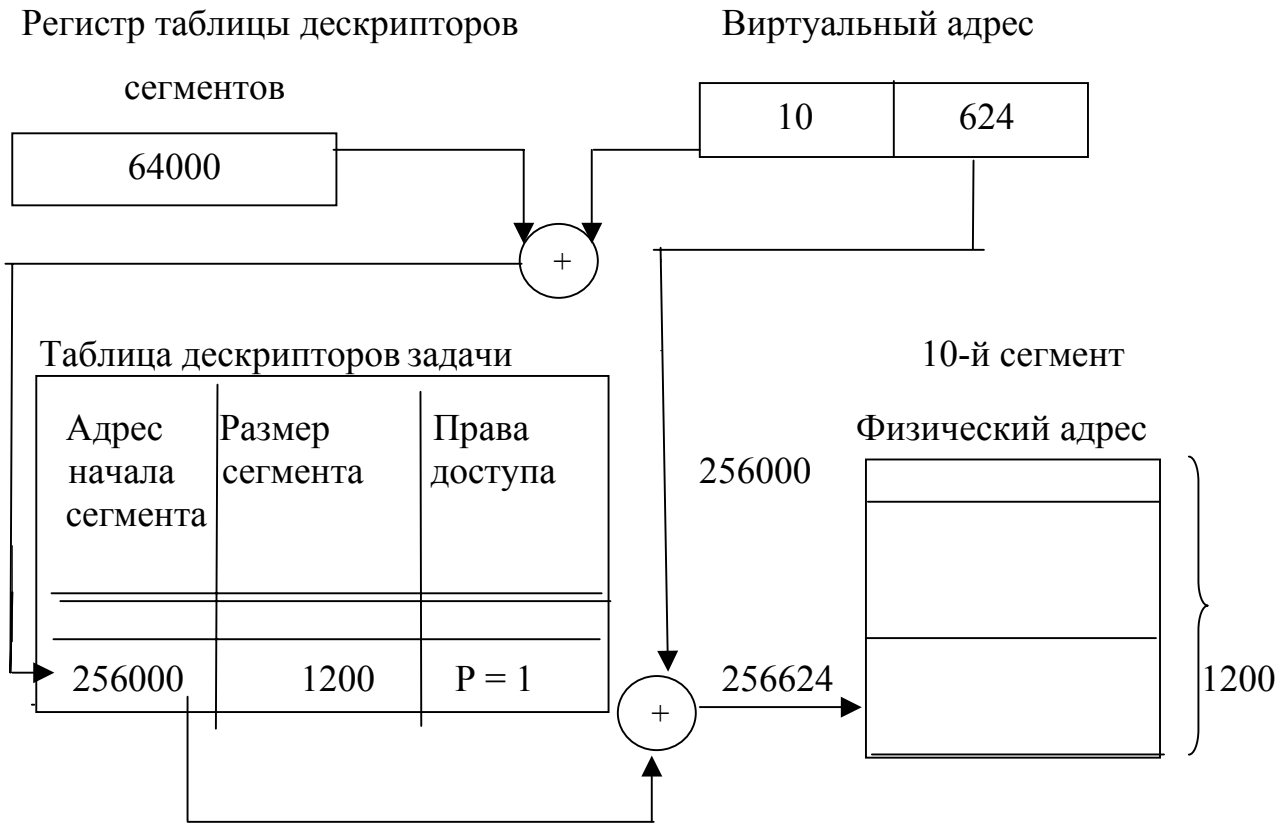


Рис. 3.3. Сегментный способ организации виртуальной памяти

Если требуемого сегмента в основной памяти нет, то возникает прерывание и управление передается программе загрузки сегмента. Загрузка в память нового сегмента управляется диспетчером памяти. Если для сегмента не хватает места, но сумма фрагментов свободной памяти достаточная, то происходит уплотнение памяти. Если свободной памяти недостаточно, принимается решение о выгрузке какого-

нибудь сегмента во внешнюю память. Для решения проблемы замещения сегментов используются следующие алгоритмы [2]:

- **FIFO** (First – In, First – Out, «первым прибыл - первым выбывает»);
- **LRU** (Least Recently Used, «последний из недавно используемых» или «дольше всего неиспользуемый»);
- **LFU** (Least Frequently Used, «используемый реже всех остальных»);
- **Random** (случайный выбор сегмента).

Алгоритмы **FIFO** и **Random** наиболее просты в реализации. Так, для алгоритма **FIFO** операционная система поддерживает список всех сегментов, находящихся в данный момент в памяти. Первый сегмент списка является старейшим, а в конце списка находятся недавно попавшие сегменты. При возникновении прерывания для замещения выбирается наиболее старый сегмент из головы списка, а новый сегмент добавляется в его конец. Алгоритм **FIFO** учитывает только время нахождения сегментов в памяти, а не их фактическое использование. Для реализации алгоритмов **LRU** и **LFU** необходимо использовать дополнительные аппаратные средства процессора, например, бит *обращения*, который менял бы свое значение при обращении к дескриптору сегмента для получения физического адреса размещения сегмента в памяти. При этом данная информация должна накапливаться по каждому сегменту в таблице дескрипторов выполняющихся задач. Тогда диспетчер памяти сможет просматривать таблицы дескрипторов для обработки статистической информации об обращениях к сегментам [2]. В результате возможно составление списка, упорядоченного либо по длительности неиспользования (для **LRU**), либо по частоте использования (для **LFU**).

Основным достоинством сегментной организации памяти является возможность задания *дифференцированных прав доступа* процесса к его сегментам. Например, сегмент данных, содержащий исходную информацию, может иметь права доступа «только чтение». Кроме того, относительно легко организовать доступ нескольких процессов к одному и тому же сегменту памяти, который называется *разделяемой памятью* (*shared memory*).

Недостатком сегментной организации памяти является *более медленное преобразование* виртуального адреса в физический адрес, чем при страничной организации. Поскольку сегмент может располагаться в физической памяти с любого адреса, необходимо задавать его полный начальный физический адрес. Другим недостатком сегментного распределения является его *избыточность*, поскольку объем сегмента в общем случае больший, чем страницы. Кроме того, существенным недостатком сегментного распределения является *фрагментация*, которая больше, чем при страничной организации, из-за непредсказуемости размеров сегментов. То есть в процессе работы системы в памяти образуются небольшие участки свободной памяти, в которые не может быть загружен ни один сегмент.

Сегментный способ организации оперативной памяти использовался в операционной системе OS/2.

3.2. Страничный способ организации виртуальной памяти

При страничном способе организации виртуальной памяти пространство виртуальных адресов разбивается на страницы одинаковой длины, кратной степени двойки. При этом виртуальный адрес может быть представлен в виде пары **(p,s)**, где **p** – порядковый номер виртуальной страницы процесса, начиная с нуля, а **s** – смещение в пределах виртуальной страницы. Соответственно и физическая память разби-

вается на страничные блоки (физические страницы) фиксированного размера. Физический адрес также может быть представлен в виде пары **(n,s)**, где **n** – номер физической страницы, а **s** – смещение в пределах физической страницы. Поскольку размер страницы выбирается равным 2^k , смещение **s** может быть получено отделением **k** младших разрядов в двоичной записи адреса. Оставшиеся старшие разряды представляют собой двоичную запись номера страницы.

Для отображения виртуального адресного пространства на физическую память, как и при сегментном способе организации, необходимо иметь *таблицу дескрипторов страниц* для трансляции адресов. На рис. 3.4 показана аппаратная трансляция виртуального адреса в физическую память. При выполнении процесса начальный адрес его таблицы страниц извлекается из специального регистра процессора, а номер **p** виртуальной страницы используется в качестве индекса для определения адреса нужного дескриптора в таблице страниц. Если бит присутствия **P**=1, то страница размещена в оперативной памяти и в дескрипторе содержится номер **n** физической страницы. Данный номер объединяется со смещением **s** внутри страницы из виртуального адреса, формируя физический адрес ячейки памяти. Если бит присутствия **P**=0, то страница размещена во внешней памяти и в дескрипторе содержится номер **p** виртуальной страницы.

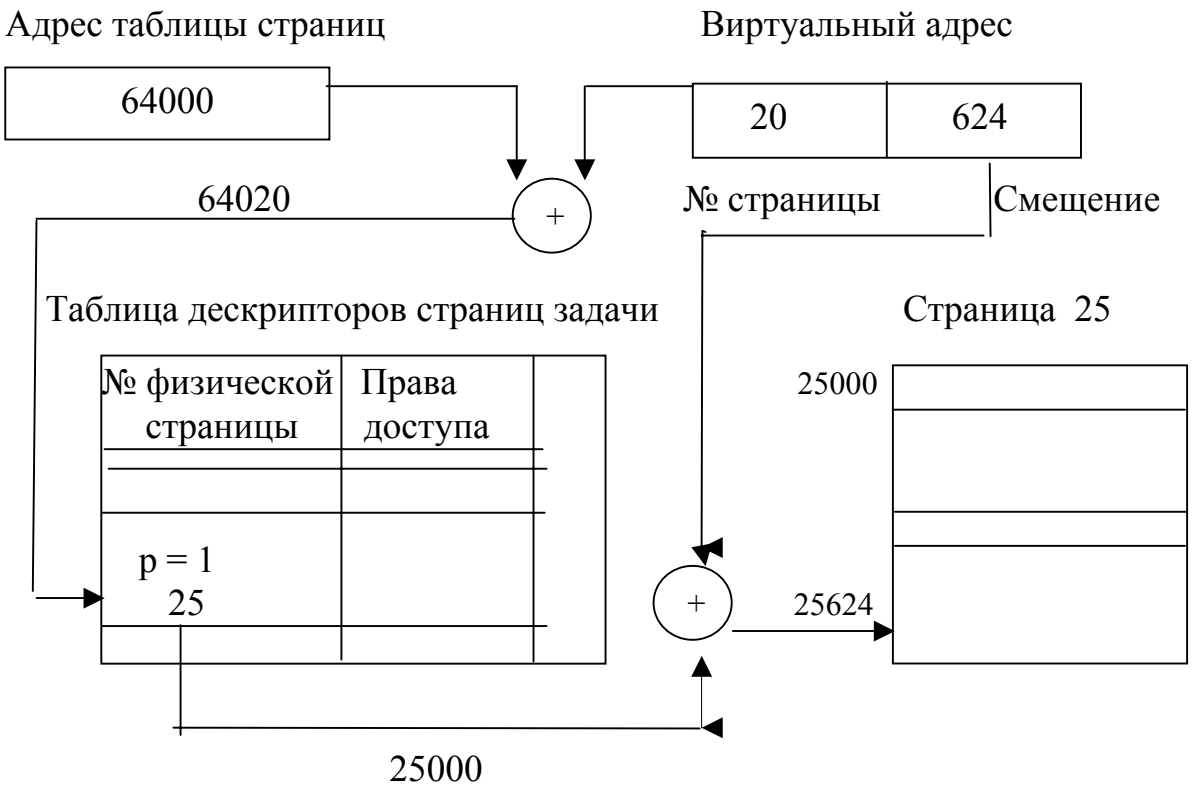


Рис.3.4. Страничный способ организации виртуальной памяти

Для каждой задачи операционная система создает таблицу дескрипторов, в которой каждый дескриптор соответствует странице, занимаемой задачей. Точная структура записи в дескрипторе зависит от архитектуры компьютера, но составляющие информационных битов примерно одни и те же. Длина записи обычно составляет 32 бита.

На рис.3.5 изображен формат дескриптора страницы:

- 20 битов (12 – 31) определяют *номер страничного блока* (№ страницы). Если добавить к этому номеру 12 нулей, то получается физический адрес начала страницы в памяти.

- Важным полем является *бит присутствия* P. Если P=0, то происходит страничное прерывание и управление передается диспетчеру памяти. Обычно предоставляется первая свободная страница. Если свободной физической страницы нет, то используется один из вышеописанных алгоритмов замещения (**LRU**, **LFU**, **FIFO**, **Random**) для определения страницы, подлежащей выгрузке во внешнюю память.
- Поля *битов защиты* контролируют права доступа к странице. В простейшем случае поле содержит один бит, равный 1 для чтения/записи и равный 0 только для чтения. Более сложная схема использует три бита, по одному для допуска каждой из операций чтения, записи и выполнения страницы [5].
- *Бит изменения или модификации* отслеживает использование страницы. Если производится модификация страницы, данный бит автоматически устанавливается и такая страница считается «грязной». При освобождении данной страницы ее новая версия должна быть переписана на диск. Если страница не была модифицирована, т.е. она «чистая», ее можно просто удалить из оперативной памяти, так как ее копия на диске действительна.
- *Бит обращения* устанавливается каждый раз при обращении к странице для чтения или записи. Этот бит играет важную роль в алгоритмах перемещения страниц.
- *Биты 9 – 11* зарезервированы для использования системными программистами.

№ страницы				Для ОС					Бит изменения	Бит обращения		Биты защиты			Бит **P** присутствия
31	…	…	12	11	10	9	8	7	6	5	4	3	2	1	0

Рис.3.5. Дескриптор страницы

Типичная машинная инструкция требует 3-4 обращений к памяти (выборка команды, извлечение операндов, запись результата). При каждом обращении происходит либо преобразование виртуального адреса в физический адрес, либо обработка страничного прерывания. Время выполнения данных операций существенно влияет на общую производительность вычислительной системы [1]. Поэтому большое внимание разработчиков уделяется оптимизации виртуальной памяти. Простейшее конструкторское решение заключается в поддержке таблицы страниц, состоящей из массива быстрых аппаратных регистров с одной записью для каждой виртуальной страницы, индексированных по номерам виртуальных страниц [5]. При запуске процесса ОС загружает в регистры таблицу страниц процесса из копии, хранящейся в оперативной памяти. Таким образом, нет необходимости обращаться к памяти во время преобразования адресов. Недостатком данного решения является высокая стоимость (для большой таблицы страниц) и необходимость загрузки полной таблицы страниц в регистры при каждом контекстном переключении процесса, что снижает общую производительность. Альтернативное решение заключается в размещении таблицы страниц целиком в оперативной памяти. Тогда достаточно одного регистра, указывающего на начало таблицы страниц. Такая схема позволяет изменять карту памяти при контекстном переключении процесса путем перезагрузки только одного регистра.

Недостатком данной схемы является увеличение времени преобразования адресов за счет нескольких обращений к памяти для чтения записей таблицы страниц во время выполнения каждой инструкции программы, а также использование дополнительного объема оперативной памяти для хранения таблицы страниц. Данная схема в чистом виде обычно не используется, однако ее усовершенствования позволяют повысить производительность. К их числу относится использование многоуровневых таблиц страниц для уменьшения объема оперативной памяти для хранения таблиц. На рис. 3.6 приводится схема двухуровневой таблицы страниц для 32-разрядной адресации [3]. При размере страницы 4 Кбайт (2^{12} байт) и максимальном объеме виртуального адресного пространства процесса 4 Гбайт (2^{32} байт) получаем 2^{20} страниц. Поскольку дескриптор каждой страницы содержит 4-байтовую запись в таблице страниц, общий объем *пользовательской таблицы страниц* из 2^{20} записей составляет 4 Мбайт (2^{22} байт). Такая большая таблица может быть размещена в 2^{10} страницах виртуальной памяти, которые отображаются *корневой таблицей страниц*, занимающей 4 Кбайт (2^{12} байт) оперативной памяти.

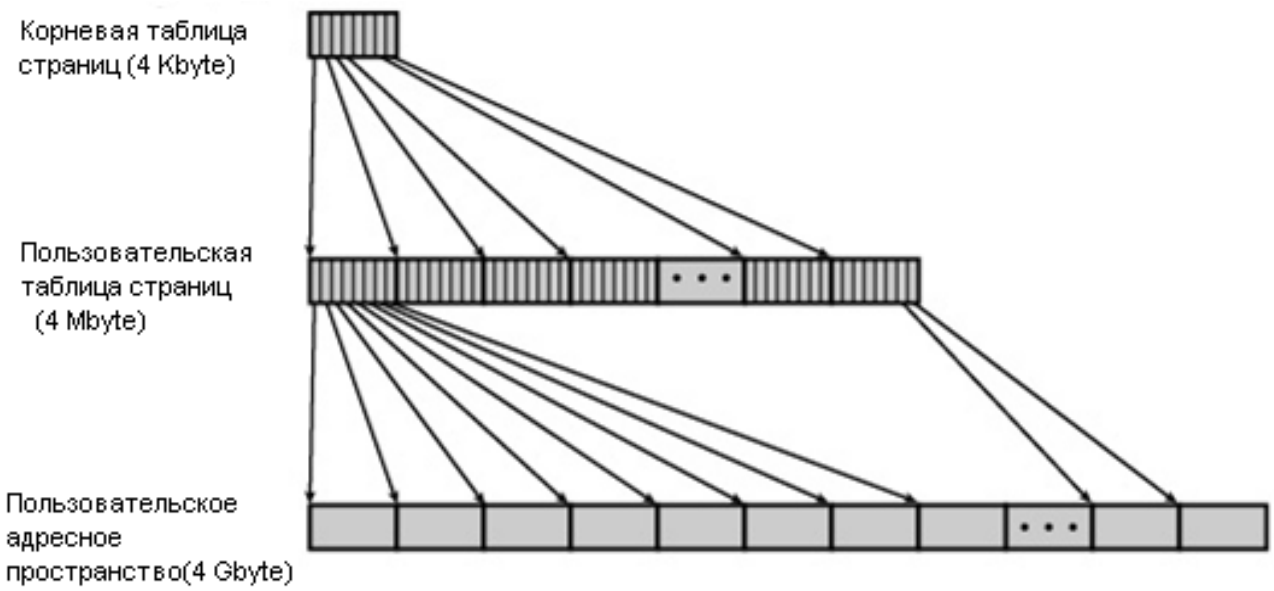


Рис.3.6. Двухуровневая таблица страниц

Трансляция виртуального адреса при двухуровневой структуре таблиц осуществляется следующим образом [3],[5]. Виртуальный 32-разрядный адрес разделяется диспетчером памяти на 10-разрядное поле для индексации в корневой таблице страниц одной из 1024 записей о странице в пользовательской таблице, 10-разрядное поле для индексации в пользовательской таблице страниц одной из 1024 записей номеров страниц, на которые ссылается виртуальный адрес и 12-разрядное поле смещения внутри страницы. Корневая таблица всегда остается в оперативной памяти. Каждая из страниц корневой таблицы представляет 4 Мбайт памяти. Если нужная страница отсутствует в оперативной памяти, генерируется страничное прерывание, в противном случае следующие 10 разрядов виртуального адреса используются для поиска записи о странице, на которую ссылается исходный виртуальный адрес.

Использование многоуровневой таблицы страниц позволяет хранить в оперативной памяти только необходимые в требуемый момент страницы [5]. Предположим, что процессу нужно 12 Мбайт памяти: 4 Мбайт для кода, 4 Мбайт для данных и 4 Мбайт для стека. Хотя адресное пространство содержит более миллиона страниц, при использовании двухуровневой таблицы страниц в оперативной памяти нужны только четыре таблицы: корневая таблица и таблицы пользовательского уровня для памяти от 0 до 4M, от 4M до 8M и для верхних 4M.

При использовании страничной организации памяти каждый виртуальный адрес вызывает обращение к двум физическим адресам: для выборки соответствующей записи из таблицы страниц и для обращения к адресуемым данным. Чтобы избежать уменьшения производительности вычислительной системы, большинство реальных схем виртуальной памяти используют высокоскоростной кэш для отображения виртуальных адресов в физические без прохода по таблице страниц. Данное устройство называется *буфером быстрого преобразования адреса* (TLB – Translation Lookaside Buffer). Оно содержит наиболее используемые записи таблицы страниц. Идея создания буфера TLB основана на анализе работы программ. Большинство существующих программ делает большое количество обращений к небольшому количеству страниц. Аппаратная реализация буфера TLB обычно осуществляется внутри диспетчера памяти. В этом случае устройство управления буфером TLB и обработка его ошибок осуществляются аппаратурой диспетчера памяти (MMU) внутри процессора. При получении виртуального адреса диспетчер памяти сначала просматривает буфер TLB. При отсутствии страницы в буфере выполняется обычный поиск в таблице страниц. Затем диспетчер удаляет одну из записей из буфера TLB и заменяет ее вновь найденной записью из таблицы страниц. В случае отсутствия страницы в опера-

тивной памяти генерируется страничное прерывание, и управление передается операционной системе, которая загружает требуемую страницу из внешней памяти и обновляет таблицу страниц.

Многие современные RISC-компьютеры, включая SPARC, MIPS, Alpha и HP PA, выполняют программное управление буфером TLB [5]. При небольшом размере буфера (64 записи) программное управление оказывается приемлемо результативным. В этом случае записи буфера TLB явно загружаются операционной системой, и освобождается пространство в микросхеме процессора для кэша и других устройств, способных повысить производительность вычислительной системы. Другим важным фактором повышения производительности является правильно выбранная стратегия замещения страниц. Более подробно алгоритмы замещения страниц рассмотрены в работе [5].

Основным достоинством страничной организации виртуальной памяти является минимально возможная фрагментация. Поскольку задача может занимать несколько страниц, то все, кроме последней, страницы будут использованы полностью.

Однако страничный способ организации памяти имеет недостатки. Страничная трансляция виртуальной памяти требует существенных накладных расходов. Таблицы страниц нужно размещать в памяти. Кроме того, программы разбиваются на страницы случайно, без учета логических взаимосвязей. Это приводит к тому, что межстраничные переходы осуществляются чаще, чем межсегментные.

Большинство современных операционных систем используют страничный способ организации виртуальной памяти.

3.3. Сегментно-страничный способ организации виртуальной памяти

Данный способ представляет комбинацию страничного и сегментного механизмов управления памятью с целью использования достоинств обоих методов.

Виртуальное адресное пространство процесса разбивается на сегменты, а каждый сегмент разбивается на страницы фиксированного размера. Перемещение данных между основной и внешней памятью осуществляется страницами. Однако разбиение на сегменты позволяет определять разные права доступа к разным частям кода и данных программы. На рис. 3.7 представлена схема преобразования виртуального адреса в физический адрес [2]. С каждым процессом связана одна таблица сегментов и несколько (по одной на сегмент) таблиц страниц. Начальный адрес таблицы сегментов извлекается из регистра процессора. Первая компонента виртуального адреса содержит номер сегмента, который служит указателем в таблице сегментов на таблицу страниц. Вторая компонента виртуального адреса (смещение относительно начала сегмента) состоит из двух полей: номера виртуальной страницы, который служит указателем в таблице страниц на физическую страницу и смещения относительно начала страницы.

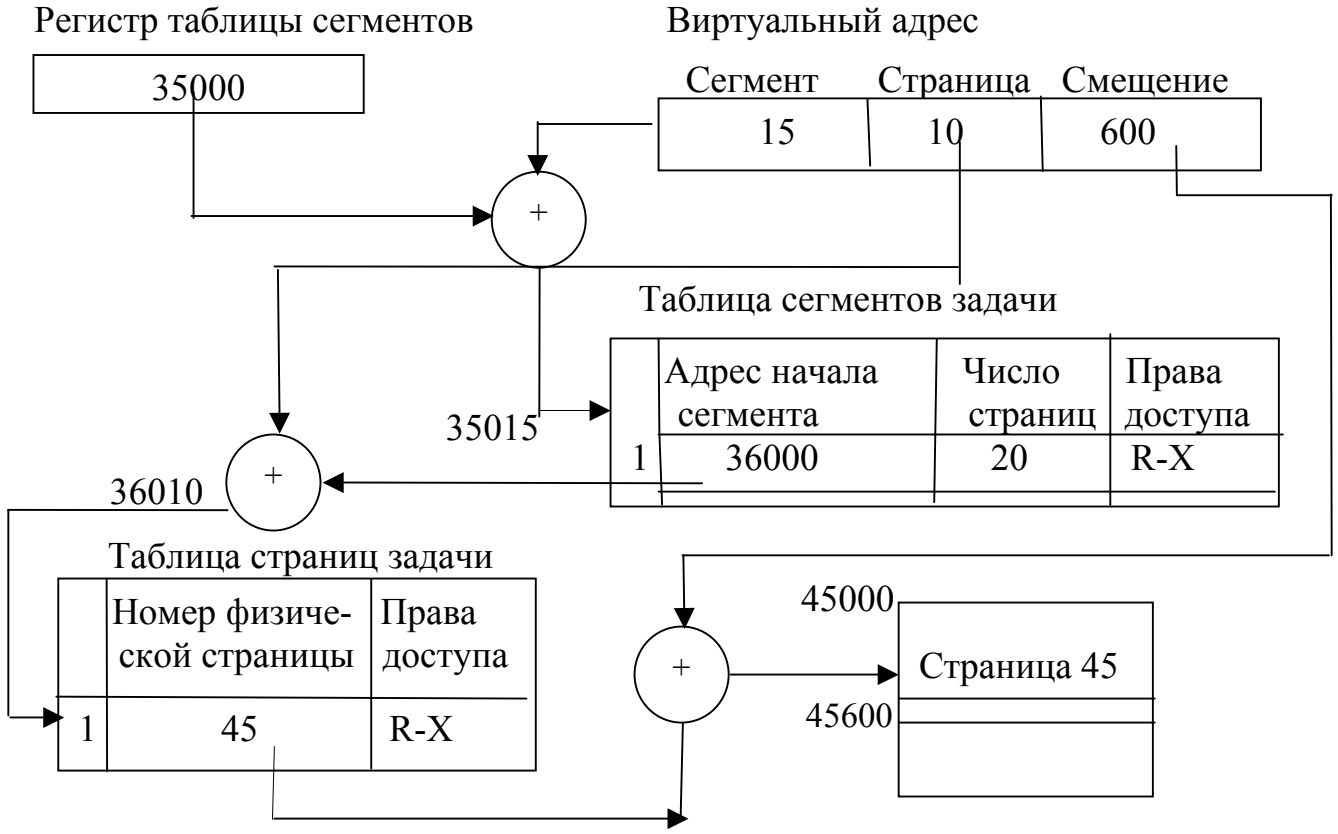


Рис.3.7. Схема преобразования виртуального адреса в физический при сегментно-страничной организации памяти

В целом, преобразование виртуального адреса в физический адрес происходит в два этапа [1].

- На первом этапе исходный виртуальный адрес, заданный в виде пары (номер сегмента, смещение), преобразуется в промежуточный *линейный виртуальный адрес* байта. Линейный виртуальный адрес вычисляется путем сложения базового адреса сегмента, извлеченного из дескриптора, и смещения (если доступ к сегменту разрешен).
- На втором этапе полученный линейный виртуальный адрес преобразуется в искомый физический адрес на основе страничного механизма преобразования. При этом линейный виртуальный

адрес представляется в виде пары (номер страницы, смещение в странице).

Рассмотренный выше способ организации виртуальной памяти 32-разрядного процессора семейства Intel Pentium включает наличие как сегментной, так и сегментно-страничной организации виртуальной памяти [5]. Если разбиение на страницы блокировано (с помощью бита в глобальном управляющем регистре), используется чистая схема сегментной организации. В противном случае, используется сегментно-страничная организация. При этом линейный виртуальный адрес отображается на физический адрес с помощью таблицы страниц. Для уменьшения размера таблицы страниц используется двухуровневое отображение.

Достоинством сегментно-страничного способа организации является загрузка всех страниц сегмента в память. Это позволяет уменьшить обращения к отсутствующим страницам. Кроме того, появляется возможность задания дифференцированных прав доступа процесса к сегментам, а также реализация разделяемой памяти между процессами. Выделение памяти страницами минимизирует фрагментацию. Однако сегментно-страничный способ требует более мощной аппаратной поддержки и программной со стороны операционной системы.

Библиографический список

1. Олифер В.Г. Сетевые операционные системы: Учебник для вузов/В.Г.Олифер, Н.А.Олифер. - СПб.: Питер, 2002. – 538с.

2. Гордеев А.В. Системное программное обеспечение: Учебник для вузов/А.В.Гордеев А.Ю.Молчанов. - СПб.: Питер, 2002. – 736с.

3. Столингс, Вильям Операционные системы/В.Столингс -4-е изд.– М.: ИД «Вильямс», 2002. – 848с.

4. Робачевский А.М. Операционная система UNIX/А.М.Робачевский. - СПб.: БХВ-Петербург, 2002. – 528с

5.Таненбаум Э. Современные операционные системы/Э.Таненбаум. 2-е изд. – СПб.: Питер, 2002. –1040с.

6. Бэкон Д. Операционные системы/Д.Бэкон Т.Харрис - СПб.: Питер; Киев: ИГ BHV, 2004. – 800с.

7. Юров В.И. Assembler : Учебник для вузов/В.И. Юров. – СПб.: Питер, 2002. – 624с.

8. Рудаков П.И., Язык ассемблера: Уроки программирования: Учеб.-справ. изд./П.И.Рудаков К.Г.Финогенов – М.: Диалог-Мифи, 2001. – 640с.

9. Митчелл М., Программирование для Linux. профессиональный подход/М.Митчелл, Д.Оулдем, А.Самьюэл . – М.: ИД «Вильямс», 2002. – 288с.

10. Операционные системы: Задание и методические указания к выполнению контрольной работы: методические указания к выполнению лабораторных работ./Сост.: Б.И.Илюшкин – СПб.: СЗТУ, 2004. – 31с.

11. Губаревич Я.Н., Системное программное обеспечение: Учеб.пособие/Я.Н.Губаревич, Б.И.Илюшкин, Р.Р.Фаткиева . – СПб.: СЗТУ, 2005. –157 с.

Предметный указатель

О

П

Р

С

Т

Я

Оглавление

Илюшкин Борис Игоревич

Операционные системы

Учебное пособие

Редактор И.Н.Садчикова

Сводный темплан 2005 г.

Лицензия ЛР № 020308 от 14.02.97

Санитарно-эпидемиологическое заключение № 78.01.07.953.П.005641.11.03 от 21.11.2003 г.

Подписано в печать .06.2005. Формат 60x84 1/16.

Б. кн.-журн. П.л.6,00. Б.л. 1,00. РТП РИО СЗТУ.

Тираж 200 Заказ

Северо-Западный государственный заочный технический университет
РИО СЗТУ, член Издательско- полиграфической ассоциации
Университетов России
191186, Санкт-Петербург, ул. Миллионная, 5