Бессонов Л. В.

Операционные системы

Опорный конспект лекций

Бессонов Л. В.

Операционные системы

Опорный конспект лекций

Учебное пособие для студентов, обучающихся по направлениям подготовки бакалавриата 09.03.03 «Прикладная информатика», 38.03.05 «Бизнес-информатика»

Саратов
ООО Издательский Центр «Наука»
2016

УДК 004.451(075.8)
ББК 32.973–018.2я73
Б53

Бессонов Л.В.

Б53 **Операционные системы. Опорный конспект лекций:** Учебное пособие для студентов, обучающихся по направлениям подготовки бакалавриата 09.03.03 «Прикладная информатика», 38.03.05 «Бизнес-информатика» /Л.В. Бессонов. — Саратов: ООО Издательский Центр «Наука», 2016. — 44 с.

ISBN 978-5-9999-2728-6

Учебное пособие раскрывает понятие операционной системы, наиболее распространённые базовые концепции построения, а также основные понятия операционных систем, такие как процессы, потоки, взаимоблокировки, менеджер памяти, драйверы, файлы и др.

Для студентов, обучающихся по направлениям подготовки бакалавриата 09.03.03 «Прикладная информатика», 38.03.05 «Бизнес-информатика», также может быть полезно студентам других направлений подготовки бакалавриата для базовой подготовки в области информационных технологий и администрирования компьютерных систем.

Рецензенты:

Кафедра математического и компьютерного моделирования
Саратовского национального исследовательского государственного
университета имени Н.Г. Чернышевского

Зав. кафедрой, д.ф.-м.н., профессор ***Ю.А. Блинков***

УДК 004.451(075.8)
ББК 32.973–018.2я73

Работа издана в авторской редакции

ISBN 978-5-9999-2728-6

Содержание

1. Понятие операционной системы

Современная компьютерная система состоит из одного или нескольких процессоров, оперативной памяти, дисков, клавиатуры, монитора, принтеров, сетевого интерфейса и других устройств, то есть является сложной комплексной системой. Написание программ, которые следят за всеми компонентами, корректно используют их и при этом работают оптимально, представляет собой крайне трудную задачу. По этой причине компьютеры оснащаются специальным уровнем программного обеспечения, называемого операционной системой.

- def *Операционная система* — комплекс программ, который управляет ресурсами компьютерной системы, осуществляет организацию вычислительных процессов в широком смысле и обеспечивает взаимодействие между пользователями, программистами, прикладными программами, системными приложениями и аппаратным обеспечением компьютера.
- def *Операционная среда* — это программная среда, образуемая операционной системой, определяющая интерфейс прикладного программирования (API) как множество системных функций и сервисов (системных вызовов), предоставляемых прикладным программам. Операционная среда может включать несколько интерфейсов прикладного программирования.
- def *Оболочка операционной системы* — в общем случае, это часть операционной системы, определяющая интерфейс пользователя, его реализацию, командные и сервисные возможности по управлению прикладными программами и компьютером.
- def *Ресурсы компьютерной системы* — физические, а также информационные компоненты компьютерной системы и предоставляемые им возможности. Под ресурсами может пониматься, к примеру: время процессора, объем дискового пространства, оперативная память, физическое либо виртуальное устройство и прочее.

1.1. Назначение операционной системы

Развитие операционных систем непосредственно связано с развитием вычислительной техники. С увеличением производительности компьютерных систем постепенно менялся, расширялся и качественно усложнялся круг задач, решаемых компьютерными системами. Соответственно изменились и требования, предъявляемые к операционным системам. В настоящий момент можно сформулировать ряд задач, для решения которых должна быть предназначена ОС. Эти задачи можно разделить на четыре основных составляющих:

1. Организация удобного интерфейса между приложениями и пользователями, с одной стороны, и аппаратурой компьютера, с другой стороны. Сюда можно отнести:
   a. Разработка программ. ОС предоставляет различные инструменты разработки (от библиотек API до редактора).
   b. Исполнение программ. ОС берёт на себя все задачи по загрузке программы в память, предоставлению для программ единообразного интерфейса ввода-вывода различных устройств, подготовке ресурсов и т.п.
   c. Доступ к устройствам ввода-вывода. Для управления любым устройством необходимо знать технические параметры и специфический для данного устройства набор команд. Операционная система скрывает сложность взаимодействия с устройствами и предоставляет пользователю удобный универсальный пользовательский интерфейс всех устройств, а программисту — удобный программный интерфейс, использующий простые команды чтения и записи.
   d. Контролируемый доступ к файлам. Доступ к файлам контролируется ОС в зависимости от типа и структуры файла и описанных прав субъекта, желающего получить доступ к файлу. Кроме того, контролируют и урегулируются конфликтные ситуации, возникающие в случае одновременного доступа.
   e. Системный доступ. ОС управляет доступом к совместно используемой и общедоступной вычислительной системе в целом, а также к отдельным системным ресурсам, защищает от несанкционированного использования и разрешает конфликтные ситуации.

f. Обнаружение ошибок и их обработка. ОС имеет собственные средства контроля возникающих ошибок исполняемых программ и аппаратуры, а также имеет возможность самостоятельно обрабатывать эти ошибки, в случае если конкретная обработка возникшей ошибки не предусмотрена программистами в соответствующей программе или драйвере аппаратуры.
g. Учёт использования ресурсов. ОС, зачастую, имеет встроенные средства учёта потребления и доступа к ресурсам, примером могут служить *счётчики* (counters) потребления сетевого трафика в ОС Linux и *система аудита* действий с файлами в ОС Windows версии 2000 и старше.

2. Организация эффективного использования ресурсов компьютера в зависимости от некоторого выбранного разработчиками ОС *критерия эффективности*. Критерии выбираются разработчиками в зависимости от назначения ОС. К примеру, для системы, контролирующей некий технический процесс (конвейерная сборка, полёт вертолёта), критерием эффективности будет служить минимальное время реакции на возникающие внешние события, а для настольного компьютера — обязательная корректная обработка всех действия пользователя (реакции на нажатия клавиш, возможность снять задачу, сохранность данных), даже если какие-то программы работают нестабильно. Управление ресурсами включает решение ряда общих задач, независимо от типа ресурса:
   a. *Планирование (распределение)* — определение, какому процессу, когда и в каком количестве (если ресурс может выделяться частями) следует выделить этот ресурс
   b. *Отслеживание состояния ресурса*
   c. *Учёт использования ресурса*
   d. *Урегулирование конфликтов*, возникающих при запросе ресурсов процессами

3. Облегчение процессов эксплуатации аппаратных и программных средств вычислительной системы. Сюда можно отнести служебные программы, обеспечивающие резервное копирование, архивацию данных, проверку, очистку, дефрагментацию дисковых устройств, программы диагностики, средства восстановления данных и прочее.
4. Возможность развития. Многие современные ОС устроены так, что допускают эффективную разработку, тестирование и внедрение новых системных функций, не прерывая процесса функционирования системы.

1.2. Функции операционной системы

Современные операционные системы включают в себя сотни и даже тысячи модулей, ориентированных на решения различных задач. Часто эти модули группируются по назначению в *подсистемы*. Каждая из таких подсистем включает в себя набор модулей и функций для решения некоторого класса задач. Эти задачи можно разделить на семь крупных классов.

1. *Управление процессами.* Подсистема управления процессами распределяет между исполняемыми процессами главный ресурс вычислительной системы – процессорное время. Параллельно решается ряд общих задач по распределению других ресурсов и управлению межпроцессорными взаимодействиями, например: синхронизация процессов и предотвращение *эффекта гонок*[1].
2. *Управление памятью.* Подсистема управления памятью распределяет имеющийся объём физической памяти между всеми существующими в данный момент времени процессами, производит загрузку программ в память, настройку адресно-зависимых частей кода процесса на физические адреса выделенной области, а также защищает области памяти каждого процесса от влияния других процессов.

[1] См. раздел «Процессы и потоки»

Одним наиболее удобным способом управления памятью, используемым в настоящее время, является механизм *виртуальной памяти*. Этот механизм позволяет программисту работать с памятью как с потенциально бесконечным ресурсом (ограниченным лишь возможностями адресации конкретной архитектуры процессора). Более того, вне зависимости от реального (возможно, весьма сложного) распределения памяти, этот механизм предоставляет программе и программисту память как однородную последовательность ячеек, занумерованную, начиная с нуля.

3. *Управление файлами.* Файловая подсистема ОС виртуализирует в виде файлов набор данных[2], хранящихся на внешнем накопителе. Для удобства пользователя файлы могут объединяться в группы — каталоги, в свою очередь, каталоги и файлы также могут группироваться в каталоги, образуя древовидную структуру. Многие реализации файловых систем позволяют реализовать не только древовидную структуру организации информации, но более сложные структуры, когда один и тот же файл или каталог отображается одновременно в различных частях структуры (например, в разных каталогах). Такая организация файлов называется *сетевой*[3], а соответствующая математическая структура носит название — сеть и является частным случаем более сложного математического объекта — графа.

Виртуализация информации в виде файлов оказалась настолько удобной, что некоторые операционные системы обобщили этот подход на прочие задачи представления ресурсов компьютерной системы. Так, например, файловые системы семейства *nix (Linux, Unix, Free BSD и прочие) отображают в файловой системе специальный каталог /dev/ каждый файл которого на самом деле является интерфейсом какого-либо устройства, и для каждого подключенного в систему устройства в этом каталоге создаётся специальный файл. Таким образом, взаимодействия с устройствами сводятся к операциям записи и чтения, производимым с такими специальными файлами.

4. *Управление внешними устройствами.* Функции управления внешними устройствами образуют подсистему ввода-вывода. Основная сложность построения этой подсистемы заключается в том, что она должна обеспечивать работу с любым подключенным устройством. Изначально ОС не может «знать» как управлять всеми возможными устройствами. Для каждого конкретного устройства производитель пишет специальную программу, встраиваемую в подсистему ввода-вывода ОС и обеспечивающую управление данным устройством. Такая программа называется *драйвером*. Т.е. подсистема ввода-вывода должна быть устроена так, чтобы допускать встраивание модулей (драйверов), написанных сторонними программистами (например, производителями оборудования), при этом взаимодействия между подсистемой ввода-вывода и прочими частями ОС должны оставаться корректными.
5. *Защита и администрирование.* Соответствующая подсистема обеспечивает сохранность данных, контроль доступа, отказоустойчивость, контроль и отработку ошибок исполнения процессов и аппаратуры. Эта подсистема влияет на работу прочих подсистем.

Одна из важнейших её задач — определение прав субъекта, получающего доступ к компьютерной системе. С этой целью используется процедура логического входа в систему, в процессе которого «устанавливается личность пользователя» (введённые имя и пароль проверяются на соответствие хранимым). Такая процедура называется *аутентификацией*[4].

[2] Здесь следует понимать, что на физическом носителе, в частности на жестком диске (НЖМД), упорядочить данные при их записи не представляется возможным. Данные хранятся разрозненно (фрагментировано), а для их корректной «сборки» в единое целое (например, в файл), в специальных областях диска записано, где расположен каждый фрагмент файла и в какой последовательности эти фрагменты следует собирать.

[3] Не следует путать с компьютерными сетями. Название «сетевая» в данном случае отражает лишь структурную организацию и представление информации. Существует также самостоятельное понятие сетевой файловой системы, подразумевающей доступ к файлам, расположенным на различных компьютерах в локальной или глобальной сети.

[4] Данную процедуру следует отличать от идентификации (опознавания субъекта информационного взаимодействия) и авторизации (проверки прав доступа к ресурсам системы).

def *Аутентификация* (*Authentication*) — *подтверждение подлинности* — процедура проверки соответствия субъекта и того, за кого он пытается себя выдать, с помощью некой уникальной информации, в простейшем случае — с помощью имени и пароля.

При доступе к конкретному ресурсу компьютерной системы подсистемой защиты и администрирования производится другая, не менее важная процедура — *авторизация*.

def *Авторизация* — процесс, а также результат процесса проверки необходимых параметров и предоставление определённых полномочий (прав доступа) лицу или группе лиц на выполнение некоторых действий в системах с ограниченным доступом.

Кроме того, во многих современных ОС предусмотрена возможность протоколирования (аудита) пользовательских действий, от которых зависит безопасность системы.

Также подсистема защиты и администрирования обеспечивает отказоустойчивость[5] вычислительной системы с использованием как программных, так и аппаратных средств.

6. *Интерфейс прикладного программирования.* Развитие модулей этой подсистемы происходит особенно бурно в последнее время. Изначально предусматривалось, что подсистема интерфейса прикладного программирования (API, Application Programming Interface) будет предоставлять прикладным программам набор функций, упрощающий написание приложений. Например, функции, отвечающие за графический интерфейс (отрисовка окон приложений, их масштабирование, перенос на экране и т.п.). Приложения выполняют обращения к функциям API с помощью системных вызовов, по логике работы похожих на вызовы подпрограмм. Таким образом, в прикладных программах эти функции не описаны, но успешно используются, что сокращает объём кода и времени написания программ, а также повышает надёжность. В последствие различных библиотек таких «удобных» функций становилось всё больше, сами библиотеки расширялись, покрывая целые предметные области. Со временем концепция интерфейса прикладного программирования эволюционировала в концепцию программных прикладных сред, которая будет рассмотрена позже.
7. *Пользовательский интерфейс.* Подсистема пользовательского интерфейса обеспечивает удобство взаимодействия пользователя (программиста, администратора) с компьютерной системой, предоставляет удобный и интуитивно понятный для человека интерфейс, обеспечивает интерактивность работы за терминалом (алфавитно-цифровым либо графическим). При работе с алфавитно-цифровым терминалом, пользователь взаимодействует с ОС с помощью команд, набираемых в командной строке. Если ОС поддерживает графический интерфейс, то взаимодействие осуществляется через множество объектов GUI (Graphical User Interface) — окна, иконки и прочие объекты.

Кроме того, существуют системы с голосовым пользовательским интерфейсом, но они менее распространены из-за сложности задачи распознавания голоса произвольно взятого человека. Такие системы, как правило, либо распознают очень ограниченный набор голосовых команд произвольного человека; либо распознают достаточно большой набор команд, но при этом «натренированы» на голос одного конкретного человека. Во втором случае процесс «тренировки» занимает много времени.

1.3. Структура и состав ОС

В состав ОС входят исполняемые и объектные модули стандартных для данной ОС форматов, программные модули специального формата (например, загрузчики ОС, драйверы ввода-вывода), конфигурационные файлы, файлы документации, модули справочной системы и т.д.

[5] Следует отметить, что подсистема защиты и администрирования *предоставляет* широкий спектр средств обеспечения отказоустойчивости и безопасности. И всё же, безопасность и отказоустойчивость наибольшим образом зависит от настройки указанных средств. А соответственно, является прямой задачей администратора системы.

Первые ОС разрабатывались как монолитные системы без какой-либо выраженной структуры и представляли собой набор функций и самостоятельных программ, каждая из которых могла беспрепятственно вызывать любую другую. Постепенно такой подход к проектированию (скорее отсутствие какого-либо подхода к проектированию) привел к возникновению ряда трудностей: во-первых, всё сложнее становилось обеспечить межмодульное взаимодействие, во-вторых, такие монолитные системы почти не способны развиваться, над ними крайне трудно работать большому коллективу программистов. Постепенно, с опытом, сложилось понятие архитектуры ОС и несколько классических архитектурных подходов.

def *Архитектура ОС* — это структурная и функциональная организация ОС на основе некоторой совокупности программных модулей. Архитектура определяет принципы действия, информационные связи и взаимодействие основных компонентов ОС.

Большинство современных ОС представляют собой хорошо структурированные модульные системы, способные к развитию, расширению и переносу на новые платформы. Какой-либо одной единой унифицированной архитектуры ОС не существует, но известны универсальные подходы к структурированию ОС. Принципиально важными универсальными подходами к разработке архитектуры ОС являются:

- Модульная организация
- Функциональная избыточность
- Функциональная избирательность
- Параметрическая универсальность
- Концепция многоуровневой иерархической вычислительной системы, по которой ОС представляется многослойной структурой
- Разделение модулей на 2 группы по функциям: ядро — модули, выполняющие основные функции ОС, и модули, выполняющие вспомогательные функции ОС
- Разделение модулей на 2 группы по размещению в памяти вычислительной системы: резидентные, постоянно находящиеся в оперативной памяти, и транзитивные, загружаемые в оперативную память только на время выполнения своих функций
- Реализация двух режимов работы вычислительной системы: привилегированного режима (или режима ядра — Kernel mode), или режима супервизора (supervisor mode), и пользовательского режима (user mode), или режима задачи (task mode)
- Ограничение функций ядра (а, следовательно, и количества модулей ядра) до минимального количества необходимых самых важных функций

Классической считается архитектура ОС, основанная на концепции иерархической многоуровневой машины, привилегированном ядре и пользовательском режиме работы транзитных модулей. Модули ядра выполняют базовые функции ОС: управление процессами, памятью, устройствами ввода-вывода и т.п. Ядро является основой ОС, без которого работа не возможна. Ядром решаются внутрисистемные задачи организации вычислительного процесса в широком смысле.

Особый класс функций ядра служит для обеспечения работы приложений. Приложения могут обращаться к ядру с запросами — системными вызовами — для выполнения тех или иных действий: обращение к элементам файловой системы, взаимодействие с устройствами и т.п.

def *Интерфейсом прикладного программирования (API, Application Programming Interface)* называются функции ядра, которые могут вызываться приложениями и системными утилитами, предоставляющие доступ к ресурсам компьютерной системы в удобной форме, абстрагировано от деталей их физического расположения и специфики взаимодействия с ними [ресурсами].

Пример 1.1. Конкретная реализация многослойной структуры ядра ОС (по слоям)

1. *Средства аппаратной поддержки ОС*: система прерываний, средства поддержки виртуальной памяти, системный таймер, средство переключения контекста процессов, средства защиты областей памяти и т.д.
2. *Машинно-зависимые модули ОС*: слой, отражающий специфику аппаратной платформы компьютера. Назначение слоя заключается в абстрагировании вышележащих слоёв от особенностей аппаратуры. В ОС Windows 2000, XP этот слой — HAL (Hardware Abstraction Layer).
3. *Базовые механизмы ядра*. Этот слой выполняет наиболее примитивные операции ядра: переключение контекста процессов, диспетчеризацию прерываний, перемещение страниц между основной памятью и диском. Модули этого слоя не принимают никаких решений самостоятельно и служат лишь исполнителями.
4. *Менеджер ресурсов*. На этом слое происходит решение задач планирования ресурсов системы.
5. *Интерфейс системных вызовов*. Это слой ядра ОС, взаимодействующий с приложениями и системными утилитами, он образует прикладной программный интерфейс.

Классическая многоуровневая архитектура имеет ряд недостатков. Первый существенный недостаток в негибкости изменений. В случае внесения серьёзных изменений в модули уровня, влияние на смежные уровни может оказаться непредсказуемым и пагубным. Вторым существенным недостатком является трудоёмкость обеспечения безопасности при множественных межуровневых взаимодействиях. Поэтому альтернативой классическому варианту часто используют *микроядерную архитектуру ОС*.

Содержательно микроядерная архитектура означает, что в привилегированном режиме остаётся работать только небольшая часть ОС — микроядро — защищённая от остальных частей и приложений. В его состав входят машинно-зависимые модули, а также модули, реализующие базовые механизмы ядра. Прочие модули функционируют в пользовательском режиме.

Преимущества и недостатки микроядерного проектирования операционных систем являются спорными. На данный момент технология развивается в сторону дальнейшего уменьшения микроядра. Если типичное ядро первого поколения микроядерных ОС занимало порядка 300 Кбайт кода и включало до 140 интерфейсов системных вызовов, то типичное микроядро второго поколения — занимает 12 Кбайт кода и 7 интерфейсов системных вызовов[6].

1.4. Классификация ОС

С целью классифицировать ОС введём категории классификации.

1. *Назначение.* По назначению ОС делятся на *универсальные* и *специализированные*. Специализированные, как правило, работают с фиксированным набором функциональных задач. Универсальные ОС рассчитаны на решение любых задач пользователя.
2. *Способ загрузки.* Можно выделить *загружаемые ОС*, коих превалирующее большинство, и *системы, постоянно находящиеся в памяти* компьютерной системы. Последние, как правило, используются для управления специальными комплексами устройств (марсоходы, спутники, баллистические комплексы).
3. *Реализация алгоритмов планирования ресурсов.*
   3.1. Поддержка многозадачности. Однозадачные позволяют исполнять одновременно не более одной программы (задачи), многозадачные — множество. Примером однозадачной ОС может служить MS DOS, а многозадачной — Linux, Windows, OS/2.

[6] Таненбаум Э. Современные операционные системы.: Пер. с англ. 2-е изд. — СПб,: Питер, 2007

3.2. Поддержка многопользовательского режима. Классификационный признак основанные на количестве пользователей, которые могут одновременно работать с системой. Главное отличие многопользовательских систем от однопользовательских — наличие средств защиты личных данных и процессов пользователя от несанкционированного доступа прочих пользователей.

Замечание 1.1.

Может быть однопользовательская многозадачная ОС.

3.3. Специфика многозадачности. По этому классифицирующему признаку могут быть выделены ОС с невытесняющей многозадачностью и с вытесняющей многозадачностью. В первом случае активный процесс выполняется пока сам не отдаст управление операционной системе, во втором случае решение о переключении процессов принимает сама ОС.

3.4. Поддержка многопроцессорности. Наличие или отсутствие возможности работы на с несколькими процессорами одновременно. В свою очередь многопроцессорные ОС классифицируются на *асимметричные* и *симметричные*. Первые выполняются на одном процессоре, распределяя прикладные задачи по остальным процессорам. Вторые — выполняют задачи ОС и прикладные процессы между всеми процессорами равномерно.

4. *Область использования и форма эксплуатации.* Эта категория образована тремя классическими типами систем:
   4.1. Системы пакетной обработки. Предназначены для решения задач вычислительного характера, не требующих быстрого получения результата и интерактивности.
   4.2. Системы разделения времени. Обеспечивают удобство и эффективность работы пользователя, который взаимодействует с ОС и программами через некоторый интерфейс.
   4.3. Системы реального времени. Предназначены для управления техническими комплексами (конвейер, автопилоты и т.п.), для которых задано предельное время реакции на то или иное событие управляемого объекта.
5. *Поддерживаемая аппаратная платформа.*
   5.1.ОС для смарт-карт. Обеспечивают работы кредитных карт, сим-карт сотовых телефонов и т.п.
   5.2.Встроенные ОС. Управляют компактными устройствами (Pasm OS для Palm, Windows CE и т.д.)
   5.3.ОС для ПК.
   5.4.ОС мини-ЭВМ.
   5.5.ОС мейнфреймов (больших машин). Обычно такие ОС подразумевают несколько видов одновременного обслуживания: пакетную обработку, обработку транзакций и разделение времени.
   5.6.Серверные ОС. Обслуживают ЛВС, региональные сети, сегменты Internet.
   5.7.Кластерные ОС. Обеспечивают функционирование *кластера*.

def *Мейнфрейм* (от англ. mainframe) — высокопроизводительный компьютер со значительным объёмом оперативной и внешней памяти, предназначенный для организации централизованных хранилищ данных большой ёмкости и выполнения интенсивных вычислительных работ. Как правило, занимают немалую площадь и обслуживаются большим штатом специалистов.

def *Кластер* — это разновидность параллельной или распределенной системы, которая состоит из нескольких связанных между собой компьютеров и используется как

единый, унифицированный компьютерный ресурс[7]. Иными словами, кластер представляет собой несколько объединенных компьютеров, управляемых и используемых как единое целое. Компьютеры кластера называются узлами. В классической схеме при работе с приложениями все узлы разделяют внешнюю память на специальном массиве жестких дисков, используя собственные внутренние дисковые накопители для специальных функций (например, системных).

1.5. Множественные прикладные среды и совместимость

Архитектурные особенности ОС непосредственно касаются программистов. С другой стороны, концепция множественных прикладных средств непосредственно сопряжены с нуждами конечного пользователя системы: возможность ОС выполнять приложения, написанные для других ОС. Такое свойство ОС называется *совместимостью*.

Современное приложение может храниться в виде двоичных кодов или в виде исходных текстов. Приложения обычно хранятся в ОС в виде исполняемых файлов, содержащих двоичные образы кодов и данных. Двоичная совместимость достигается в том случае, если можно взять исполнимую программу и запустить её на выполнение в среде чужеродной ОС.

Совместимость на уровне исходных кодов требует наличия соответствующего компилятора в составе программного обеспечения компьютера, на котором предполагается выполнять данное приложение, а также совместимость на уровне библиотек и системных вызовов. При этом необходима перекомпиляция исходных текстов приложения в новый исполняемый модуль.

Самый главный фактор совместимости — архитектура процессора.

Необходимое и достаточное условия двоичной совместимости. Достаточно, чтобы различные процессоры использовали один и тот же набор базовых команд и один и тот же диапазон адресов. В этом случае для достижения двоичной совместимости необходимо соблюдение условий:

- API, которые использует приложение, должны поддерживаться данной ОС;
- Внутренняя структура исполняемого файла приложения должна соответствовать структуре исполняемых файлов данной ОС, по крайней мере, не противоречить ей.

В случае различной архитектуры процессоров, кроме указанных в *необходимости* условий требуется организовать эмуляцию двоичного кода. Эта задача возлагается на некоторый программный эмулятор, который последовательно выбирает двоичные инструкции написанные для одного процессора и выполняет эквивалентные им инструкции другого процессора. При этом иногда часть аппаратных особенностей, отсутствующих у целевого процессора, по сравнению с исходным, должна полностью реализовываться эмулятором (например, недостающий регистр или набор флагов состояний).

Эмуляция не представляет принципиальной трудности, но ведёт к большим затратам времени, поскольку одна команда исходного процессора выполняется существенно быстрее, чем эмулирующий набор команд конечного. Выходом в таком случае является использование прикладных программных сред или операционных сред. Одной из составляющих такой среды является набор функций интерфейса прикладного программирования API, который ОС предоставляет своим приложениям. Для сокращения времени выполнения чужих программ прикладные среды имитируют обращение к библиотечным функциям.

Эффективность этого подхода связана с тем, что большинство программ работают с использованием GUI (графический интерфейс пользователя), при этом приложения тратят 50…80% времени на выполнение функций GUI (отрисовка элементов интерфейса, реакция на интерфейсные события). Это свойство приложений помогает прикладным средам компенсировать большие затраты времени на эмуляцию.

[7] Определение Грегори Пфистер (Gregory F. Pfister), одного из первых архитекторов кластерной технологии

Описанный выше подход называется *кроссплатформенной трансляцией*.

Замечание. При осуществлении трансляции необходимо учитывать особенности исходной и конечной ОС. Некоторые разрешенные в исходной ОС операции могут быть запрещены в конечной ОС.

2. Концепция операционной системы

Опыт работы с операционной системой имеет любой современный пользователь, однако если попросить дать определение операционной системы, это вызовет определённые затруднения. Операционная система выполняет такие функции, что в понимании большинства пользователей не отделима от компьютерной системы.

Очертив круг задач, решаемых операционной системой, можно условно разделить его на две: расширение возможностей машины (как аппаратной платформы) и управление ее ресурсами.

2.1. Операционная система как виртуальная машина

def *Виртуальная машина* — концептуальный подход в программировании, позволяющий отделить программное обеспечение от нижележащей аппаратной платформы по средствам формирования промежуточного программного слоя.

Следует заметить абстрактность и широчайшие возможности виртуальных машин. Подход оказался настолько удачным, что получил применение в различных областях: от проектирования и создания операционных систем до программ специального назначения (мониторов виртуальных машин), назначение которых — эмулировать работу чужеродной (гостевой) операционной системы в рамках некоторой установленной на компьютере операционной системы.

Итак, сложности различных архитектур аппаратных платформ (система команд, организация памяти, ввод-вывод данных и структура шин) уже была рассмотрена. Особенно хорошо эта сложность заметна на задачах эмуляции, рассмотренных в параграфе о совместимости программ и операционных систем между собой.

При программировании современных приложений (тем более при их эксплуатации) встаёт ряд высокоуровневых задач проектирования, требующих рассуждений в рамках логики, оторванной от физической реализации расчетов, необходимых для решения этих задач.

К примеру, при решении простейшей задачи на языке Паскаль, осуществляя вывод полученного значения на экран, программист не должен и не хочет задумываться о том, как это значение размещено в памяти, какие шины и регистры процессора должны быть задействованы для вывода на экран и т.п. Даже если не вдаваться глубже в подробности этого процесса, становится ясно, что обыкновенный программист вряд ли захочет столкнуться с такими деталями. Вместо этого программисту нужны простые высокоуровневые абстракции. В случае работы с дисками типичной абстракцией является имена файлов, содержащихся на диске, в случае вывода на экран — *поток вывода*[8].

Программа, скрывающая истину об аппаратном обеспечении и представляющая набор удобных для работы абстракций, и является операционной системой. Операционная система не только устраняет необходимость работы непосредственно с дисками и предоставляет простой, ориентированный на работу с файлами интерфейс, но и скрывает множество неприятной работы с прерываниями, счетчиками времени, организацией памяти и другими элементами низкого уровня. В каждом случае абстракция, предлагаемая операционной системой, намного проще и удобнее в обращении, чем то, что предлагает непосредственно аппаратура.

[8] Поток — удобная абстракция, под потоком может пониматься файл, в который осуществляется вывод. В свою очередь таким файлом может являться *специальный файл* представляющий интерфейс некоторого устройства, например, монитора или принтера.

- *Абстракция данных* – подход к обработке данных по принципу "черного ящика". Данные обрабатываются функцией высокого уровня с помощью вызова функций низкого уровня. Обычно такой подход используется в объектно-ориентированном программировании, что позволяет работать с объектами, не вдаваясь в особенности их реализации.
- *Абстракция аппаратная* – концепция взаимодействия программ и устройств в рамках ОС, подразумевающая работу с устройством как с "черным ящиком", имеющим определенное количество входов и выходов. В процессе взаимодействия программы осуществляют запись на входы черного ящика и чтение с выходов, при этом операции по корректной пересылке данных и управлению реальным устройством берут на себя низкоуровневые процедуры, встроенные в ОС.

С точки зрения пользователя операционная система выполняет функцию расширенной машины или виртуальной машины, в которой проще программировать и легче работать, чем непосредственно с аппаратным обеспечением, составляющим реальный компьютер.

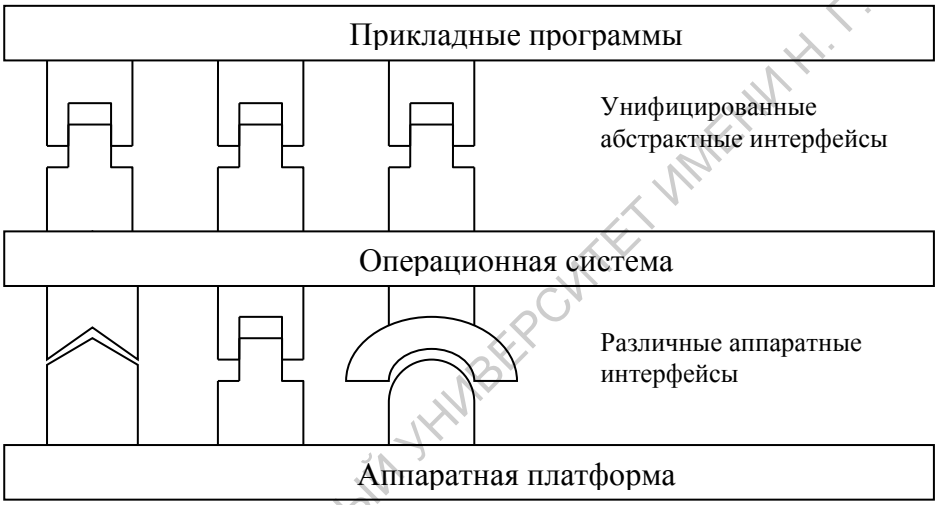


Рисунок 1. Операционная система как виртуальная машина

2.2. Операционная система как менеджер ресурсов

Концепция, рассматривающая операционную систему, прежде всего, как удобный интерфейс пользователя, — это взгляд сверху вниз. Альтернативный взгляд, снизу вверх, дает представление об операционной системе как о механизме, присутствующем в устройстве компьютера для управления всеми частями этой сложнейшей машины. В соответствии со вторым подходом работа операционной системы заключается в обеспечении организованного и контролируемого распределения процессоров, памяти и устройств ввода-вывода между различными программами, состязающимися за право их использовать.

Пусть на одном компьютере запущено три программы, и все они одновременно отправляют на один и тот же принтер свои выводные данные. Возможно, первые несколько строк на листе появились бы от первой программы, следующие несколько — из второй программы, затем бы следовало несколько строк от третьей программы и т.д. Результат неприемлем. Операционная система наводит порядок в подобных ситуациях, буферизируя на диске все данные, предназначенные для печати. В процессе работы программы операционная система сохраняет ее выходные данные на диске во временном файле. Затем, по окончании работы этой программы, система отправляет данные на принтер, в то время как другая программа может продолжать формировать свои выходные данные, не обращая внимания на то, что они пока еще фактически не посылаются на печатающее устройство.

Когда компьютером (или сетью) пользуются несколько пользователей, необходимость в управлении памятью, устройствами ввода-вывода, другими ресурсами и их защите сильно возрастает, поскольку пользователи могут обращаться к ним в абсолютно непредсказуемом порядке. К тому же часто приходится распределять между пользователями не только оборудование, но и информацию (файлы, базы данных и т. д.). С этой точки зрения основная задача операционной системы заключается в отслеживании того, кто и какой ресурс использует, в обработке запросов на ресурсы, в подсчете коэффициента загрузки и разрешении проблем конфликтующих запросов от различных программ и пользователей.

Управление ресурсами включает в себя их *мультиплексирование* (распределение) двумя способами: во времени и в пространстве. Когда ресурс распределяется во времени, различные пользователи и программы используют его по очереди. Сначала один из них получает доступ к использованию ресурса, потом другой и т.д. Например, несколько программ хотят обратиться к центральному процессору. В этой ситуации операционная система сначала разрешает доступ к процессору одной программе, затем, после того как она поработала достаточное время, другой программе, затем следующей и, в конце концов, опять первой. Определение того, как долго ресурс будет использоваться во времени, кто будет следующим и на какое время ему предоставляется ресурс — это задача операционной системы. Еще один пример временного мультиплексирования — распределение заданий, посылаемых для печати на принтер. Когда задания выстраиваются в очередь для печати на одном принтере, операционной системе каждый раз нужно принимать решение о том, которое из них будет печататься следующим.

3. Основные понятия операционной системы

Для каждой операционной системы существует набор базовых понятий, например, процессы, память и файлы, которые являются самыми важными для понимания общей идеи. Рассмотрим некоторые основные понятия, иллюстрируя их в основном на примере ОС UNIX.

3.1. Процессы и потоки

Ключевое понятие операционной системы — *процесс*. Содержательно процесс — это программа в момент её выполнения. Отличие процесса от программы, записанной, но не исполняющейся в данный момент, заключается в следующем. С каждым процессом связывается некий набор регистров, в том числе счетчик команд, указатель стека и другие аппаратные регистры, а также вся остальная информация, необходимая для запуска процесса.

Большинство современных систем может выполнять несколько процессов одновременно. Например, пользователь может запустить программу проигрыватель музыки и включить свою рабочую программу, чтобы выполнять необходимые ему работы под музыку. Кроме того, во время работы пользователя происходит работа множества сервисных программ: антивирусы, программы резервного копирования, планировщики и т.п.

Следует понимать, что один исполнитель (процессор) одновременно может выполнять лишь одно действие. То есть, одновременно исполнение программ — это иллюзия. Выполнение программ на одном процессоре с иллюзией одновременного выполнения также называют *псевдопараллельным*. Эффект одновременности возможен благодаря тому, что процессор может выполнять большое количество операций в единицу времени[9]. Таким образом, если каждому процессу предоставлять какой-то небольшой интервал времени (часть секунды, к примеру), то за это время процесс успеет выполнить достаточную свою часть. Поскольку такие интервалы времени, на которые делится процессорное время (их ещё называют *квантами*) очень малы — пользователь не успевает заметить поочерёдность выполнения, у него складывается впечатление одновременного выполнения нескольких программ.

[9] Для современных процессоров показатель производительности имеет порядок ГГц, т.е. тысяч МГц (1МГц = 1 млн. операций в секунду).

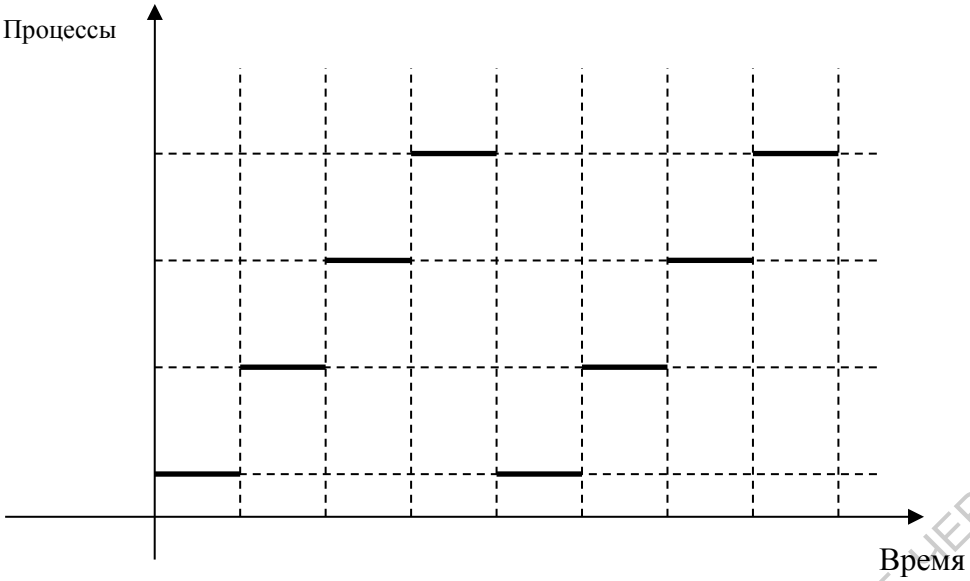


Рисунок 2. Псевдопараллельное выполнение четырех процессов

Процесс — в общем случае, это программа, находящаяся в памяти и получившая управление, выполняющаяся программа. Более точное определение *процесса* можно дать лишь для конкретной операционной системы.

- *Адресное пространство процесса* — это список адресов памяти от некоторого минимума (обычно нуля) до некоторого максимума, которые процесс может прочесть и в которые он может записать информацию. Область памяти, определяемая адресным пространством процесса, содержит код, данные и стек программы.
- *Контекст процесса (летучая среда процесса)* — это связанный с процессом набор значений регистров, счетчика команд, набор указателей на дескрипторы открытых файлов, информация о незавершенных операциях ввода-вывода, коды ошибок, выполняемых данным процессом системных вызовов, и прочие технические сведения о состоянии процесса в момент времени. Контекст процесса есть вектор-функция времени.

В процессе работы ОС осуществляет запуск, диспетчеризацию и завершение процессов. К диспетчеризации относится, в том числе, переключение процессов. Переключение процессов подразумевает приостановку одного процесса и передачу управления другому. Если процесс был приостановлен подобным образом, позже он должен быть запущен заново из того же состояния, в каком его остановили. Подобного рода возобновление возможно за счет сохранения в некоторой области памяти *летучей среды процесса* (контекста процесса), при его остановке.

Пример 3.1. Юлий Цезарь

Пусть Цезарь выполняет одновременно два процесса. Чтобы процессы выполнялись параллельно, Цезарь делает по небольшой части каждого из них, попеременно переключаясь. Т.е. сделав небольшую часть первого, переходит ко второму, сделав небольшую часть второго — вновь к первому и т.д. Процесс А — игра в шахматы. Процесс В — чтение трактата Платона.

Чтобы вернуться к выполнению процесса А с того же места, на котором он был прерван, Цезарю необходимо знать шахматную позицию на доске и очерёдность хода.

Чтобы вернуться к выполнению процесса В, необходимо помечать место, на котором Цезарь остановился при чтении, и возвращаясь, продолжать чтение с помеченного места.

В данном примере контекст процесса А — позиции фигур на шахматной доске и очерёдность хода, контекст процесса В — пометка места на котором остановлено чтение (абзац, строка, предложение). Аккуратно сохраняя контекст каждого из процессов, Цезарь сможет выполнять оба процесса параллельно

Во многих операционных системах вся информация о каждом процессе (естественно кроме содержимого адресного пространства процесса) хранится в таблице, организованной операционной системой. Такая таблица называется таблицей процессов и представляет собой массив (или связный список) структур, по одной на каждый существующий в данный момент времени процесс.

Таким образом, архитектурно, приостановленный процесс состоит из собственного адресного пространства, обычно называемого *образом памяти* (core image – «сердечник»), и компонентов таблицы процессов, содержащей, помимо других величин, его регистры.

Процесс может создавать несколько других процессов (они называются *дочерними процессами*, а породивший их процесс по отношению к ним называется *материнским*), а те в свою очередь могут создавать свои дочерние процессы. Таким образом, образуется *дерево процессов*. Как правило, дочерние процессы создаются материнскими для осуществления некоторой задачи, а значит процессам необходимо взаимодействовать. Такая связь называется *межпроцессорным взаимодействием* (IPC – interprocess communication) и состоит в передаче данных от одного процесса к другому, контроле деятельности процессов, синхронизации действий. При этом контроль деятельности процессов обеспечивает распределение ресурсов и управление доступом, а синхронизация подразумевает совмещение процессов во времени особым образом, и устранение возможных негативных эффектов, например, *эффекта гонок*.

- *Синхронизация* (от греч. synchronos – одновременный) — приведение двух или нескольких процессов к такому их протеканию, когда одинаковые или соответствующие элементы процессов совершаются с неизменным сдвигом во времени либо одновременно.
- *Эффект гонок* — эффект десинхронизации, проявляющийся в том, что процесс в своём выполнении доходит до этапа, требующего данных, получаемых от другого процесса, в то время как второй процесс ещё не выполнился до момента передачи данных. К примеру: интерфейсный процесс А готов вывести на печать результат работы вычислительного процесса В, а процесс В ещё не завершил вычисления.

С момента запуска процесс последовательно переживает определённый набор состояний (в той или иной очередности):

1. Выполнение — состояние, когда процесс непосредственно исполняется на процессоре.
2. Готовность — процесс временно приостановлен, чтобы позволить выполняться другим процессам (при это других объективных причин для невыполнения данного процесса может не существовать).
3. Ожидание — процесс не может выполняться до тех пор, пока не произойдёт некоторое внешнее относительно этого процесса событие (например, пока не освободится устройство ввода-вывода или пока от другого процесса не будут получены необходимые для выполнения данные).

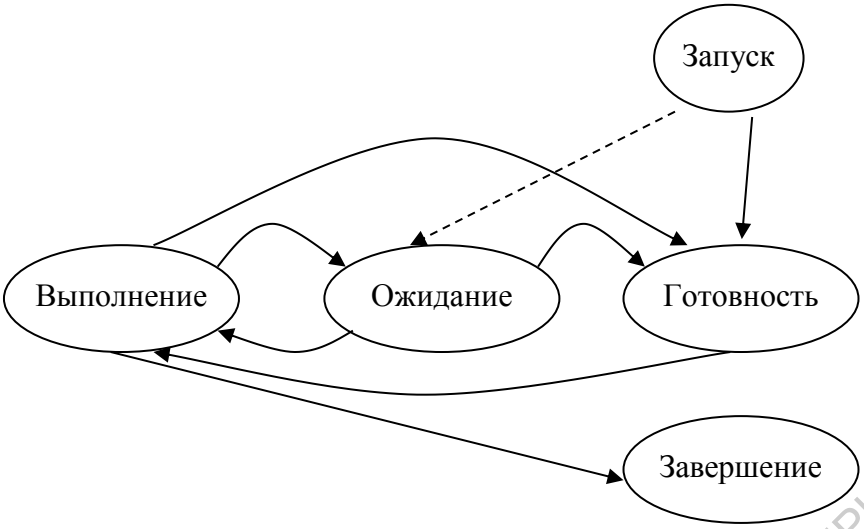


Рисунок 3. Смена состояний процесса

Потоки

В некоторых операционных системах каждому процессу соответствует адресное пространство и один поток команд (собственно программа), называемый *управляющим потоком*. По сути это и есть процесс. На деле, часто удобно иметь несколько параллельных (псевдопараллельных) управляющих потоков в одном и том же адресном пространстве.

Рассмотренное понятие процесса базируется на двух независимых концепциях: группировании ресурсов, необходимых программе (память, устройства и т.п.), и выполнении самой программы. Иногда полезно разделять эти концепции. В результате приходим к понятию *потока*.

def *Потоком (или управляющим потоком)* будем называть последовательность команд, со связанным с нею указателем команд.

Детально рассмотрим отличие понятий потока и процесса. Процесс подразумевает группировку ресурсов (при запуске процесса, он требует от системы какие-то ресурсы). Когда необходимые ресурсы (в том числе процессорное время) выделены процессу, запускается управляющий поток (то есть процесс непосредственно исполняется). Поток подразумевает лишь исполнение управляющего потока. При этом, по сути в рамках одного процесса могут выполняться несколько потоков. Объединение в процессе нескольких потоков обеспечивает всем потокам одни и те же ресурсы и совместную работу с ними.

Такой подход оказывается очень удобным. Например, рассмотрим текстовый редактор. Запущен один процесс — текстовый редактор. В рамках этого процесса запущено три потока: первый из них обеспечивает запоминание вводимого пользователем текста, второй поток обеспечивает отображение вводимых данных на экране так, как этот текст будет размещён на листе, третий поток проверяет орфографию во введённом тексте. Все потоки, запущенные в рамках процесса текстового редактора используют одни и те же ресурсы — введённые пользователем текст, при этом каждый по-своему обрабатывает этот текст. Такой подход очень удобен и на стадии проектирования. Однажды определив, каким образом потоки будут взаимодействовать между собой, можно проектировать соответствующие потоки независимо друг от друга, соблюдая лишь договорённости о взаимодействии.

При запуске многопоточного процесса в системе с одним процессором потоки работают поочередно. Пример работы процессов в многозадачном режиме был показана на рис. Рисунок 2**Ошибка! Источник ссылки не найден.**. Иллюзия параллельной работы нескольких различных последовательных процессов создается путем постоянного переключения системы между процессами. Многопоточность реализуется примерно так же. Процессор переключается между потоками, создавая впечатление параллельной работы потоков.

Часто потоки используются обработки возникающих в системе и пользовательском приложении событий. Такие процессы называются *всплывающими*. После запуска в приложении выполняется лишь один поток. В момент возникновения какого-либо события "всплывает" поток, основной задачей которого является обработка произошедшего события.

Пример 3.2. Длительная обработка события

Допустим, программа (написанная, скажем, на Delphi) должна по нажатию кнопки "Копировать" на форме приложения, программа должна произвести резервное копирование большого количества файлов, при этом отображая ход этого резервного копирования на визуальной шкале (progress bar).

Когда программа запущена, и пользователь нажал кнопку "Копировать", происходит обработка события "Нажатие на кнопку". Пока это событие не обработано до конца, приложение не будет реагировать ни на одно другое внешнее событие. Соответственно, если копирование происходит длительное время (больше нескольких секунд), операционная система сочтёт приложение зависшим. В частности, не будет осуществляться перерисовка окна приложения (а там ведь должна отображаться визуальная шкала).

Чтобы избежать такой ситуации, необходимо поступить следующим образом. При нажатии на кнопку "Копировать", приложение создаст новый управляющий поток, который должен будет осуществлять резервное копирование. При этом обработка события "Нажатие на кнопку" будет завершена, как только поток создан. После этого приложение готово реагировать на любые другие события. Созданный поток осуществляет резервное копирование. Поскольку оба потока работают в одном адресном пространстве, поток копирования может обращаться к элементу формы "визуальная шкала" и менять на ней значение. При этом приложение будет верно функционировать.

Межпроцессное взаимодействие

Процессам (и потокам) необходимо взаимодействовать друг с другом. При этом возникает ряд ситуаций, требующих дополнительного регулирования. Например, если несколько процессов используют один и тот же ресурс, необходимо контролировать последовательность получения доступа, чтобы процессы работали корректно. Рассмотрим способы организации межпроцессорных взаимодействий.

Пример 3.3. Спулер (spooler)

Пусть процессу необходимо вывести страницу (или несколько страниц) на печать, он помещает данные для печати в спулер (в зависимости от операционной системы это может быть каталог, файл или область памяти). Другой процесс, отвечающий за печать, по очереди берёт переданные задания и выводит их на принтер.

Тем самым снимается конкуренция за использование принтера различными процессами. Кроме того, процесс печати может решать по каким-либо определённым правилам в какой очерёдности следует пускать задания на печать.

Заметим, что спулер в данном примере реализует межпроцессное взаимодействие, в котором множество процессов, желающих вывести данные на принтер взаимодействуют через общий ресурс (спулер) с печатающим процессом. При этом ресурс "принтер" по сути монопольно занят один единственным печатающим процессом.

В случае взаимодействия двух произвольных процессов, не всегда возможно организовать в операционной системе специальный процесс для регулирования этих взаимодействий. Чтобы решить задачи взаимодействий на совместно используемых ресурсах вводят некоторые специальные понятия.

def *Критическая секция (или критическая область)* — это часть программы, в которой происходит обращение к совместно используемым ресурсам.

Критерий отсутствия состязательности. Два и более процессов, использующих один и тот же общий ресурс не состязаются за этот ресурс тогда и только когда, когда в критической секции, связанной с этим ресурсом одновременно находится не более чем один из этих процессов.

В самом деле, часть времени процесс занимается внутренними расчётами и не использует общий ресурс. Как только этот процесс входит в критическую секцию, т.е. происходит работа с общим ресурсом, об этом особым образом становится известно. Если в это время (пока первый процесс не вышел из критической секции) какой-либо другой процесс попробует войти в критическую секцию (т.е. начать работать с общим ресурсом), ему будет в этом отказано. Точнее, второй процесс будет приостановлен до тех пор, пока первый не выйдет из критической секции. Это можно проиллюстрировать на рисунке.

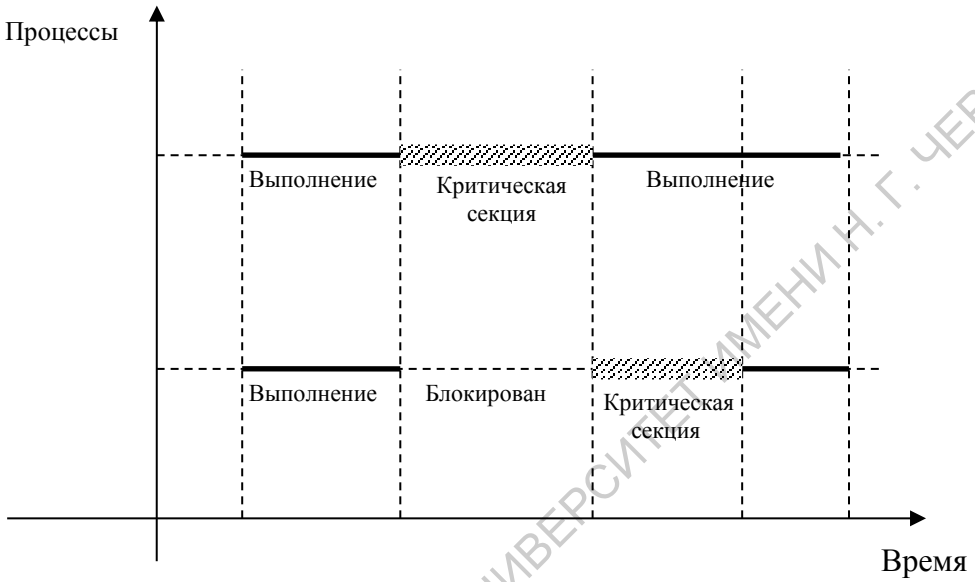


Рисунок 4. Исключение состязательности с использованием критических областей

Теоретическая концепция критических областей имеет несколько стандартных реализаций, применяемых в различных операционных системах. Подробно рассмотрим лишь некоторые из них.

1. *Запрет прерываний.*

   Если прерывания запрещены, то невозможно и переключение на другой процесс, который может состязаться за какие-либо ресурсы. Однако такой подход весьма неразумен, поскольку заранее не известно время, которое процесс будет находиться в критической области. Таким образом, пока пользовательский процесс в критической секции, не сможет произойти ни одна обработка, в том числе системных и неотложных событий.

2. *Переменные блокировки.*

   Если процессы используют один и тот же ресурс, разумно использовать некоторую общую переменную — *переменную блокировки* — которую изначально положить равной 0, а когда процесс будет входить в критическую область, он будет менять значение этой переменной на 1. Таким образом, если некоторый процесс хочет войти в критическую секцию, а переменная блокировки равна 1, процесс будет ожидать до тех пор, пока переменная блокировки не обратится в 0, что будет означать в критической секции не находится ни одного процесса.

Кроме рассмотренных можно назвать распространённые реализации: строгое чередование, алгоритм Петерсона, флаги готовности, алгоритм булочной (Bakery algorithm).

В простейших случаях концепция критических секций отлично срабатывает. Но существует ряд ситуаций, когда критических секций не достаточно. Рассмотрим на примере.

Проблема производителя и потребителя. Пусть два процесса совместно используют буфер ограниченного размера. Один из процессов помещает в буфер информацию (назовём этот процесс производителем), а другой читает информацию из буфера (назовём этот процесс потребителем). Трудность возникнет в тот момент, когда производитель заполнит буфер целиком. Решение очевидно, производитель должен ожидать пока потребитель прочтёт частично или полностью информацию из буфера. Аналогичная трудность возникнет, когда потребитель обратится к буферу для чтения и обнаружит, что буфер пуст. В этом случае потребитель должен ждать, пока производитель не поместит информацию в буфер.

Решение кажется достаточно простым, но приводит к состязательному состоянию двух процессов, даже при использовании критических секций в реализации производителя и потребителя. Дело в том, что для учёта заполненности буфера необходимо использовать какую-то общую для производителя и потребителя переменную. И как раз за эту переменную процессы будут состязаться. Можно ввести вспомогательный механизм, решающий задачу, например, установить бит активации, указывающий можно ли получать доступ к счётчику заполненности буфера. Однако можно смоделировать ситуацию с несколькими процессами, когда это решение не будет работать. Требуется сформулировать более общий подход.

В 1965г. Дейкстра (E.W. Dijkstra) предложил использовать специальную переменную целого типа, получившую название — *семафор*. Семафор связывается с совместно используемым ресурсом. Каждое обращение к ресурсу абстрактно будем называть *сигналом активизации*.

- *Семафор* — это неотрицательная целочисленная переменная, связанная с совместно используемым ресурсом, которая может быть нулём (в случае отсутствия сохранённых сигналов активизации) или некоторым положительным числом, соответствующим количеству отложенных сигналов активизации.

  Над семафорами определены две операции:

  1. down(sem) — сравнивает значение семафора с нулём, если значение больше нуля, то уменьшает его на 1 (то есть расходует один из сохраненных сигналов активизации) и возвращает управление. Если значение семафора равно нулю, процедура down() не возвращает управление процессу, а процесс переводится в состояние ожидания.
  2. up(sem) — увеличивает значение семафора на 1. При этом если с этим семафором связаны один или более ожидающих процессов, которые не могут завершить более раннюю операцию down(), а это означает что значение семафора равно 0, один из ожидающих процессов будет выбран системой и ему будет разрешено завершить down().

- *Мьютекс* —это семафор, находящийся в одном из двух возможных состояний: 0 — блокирован, либо 1 — не блокирован. Если процесс должен войти в критическую секцию, и мьютекс не заблокирован, процесс входит в критическую секцию, при этом заблокировав мьютекс. Если мьютекс заблокирован, вызывающий процесс блокируется до тех пор, пока процесс, работающий в критической области, не выйдет из неё.

Пример 3.4. Решение проблемы производителя и потребителя с помощью семафоров

```
#define N 100;                    /* Определяем размер буфера(в ячейки) */
typedef int semaphore;            /* Задаём тип "семафор" как целый */
semaphore mutex = 1;              /* Контроль входа в критическую область */
semaphore empty = N;              /* Число свободных ячеек буфера */
semaphore full = 0;               /* Число занятых ячеек буфера */
```

```
void producer(){
      /* Переменная "элемент", в неё
будем класть вновь созданный элемент,
для дальнейшей отправки в буфер */
            int item;
      /* Бесконечный цикл */
      while (TRUE){
            /* Создать элемент */
            item=produce_item();
            /* Уменьшить empty */
            down(empty);
            /* Войти в критическую
область*/
            down(mutex);
            /* Поместить в буфер
созданный элемент */
            put_item(item);
            /* Выход из критической
области */
            up(mutex);
            /* Увеличить full */
            up(full);
      }
}
```

```
      void consumer(){
      /* Переменная "элемент", в неё
будем класть взятый из буфера для
обработки элемент */
            int item;
      /* Бесконечный цикл */
      while (TRUE){
            /* Уменьшить full */
            down(full);
            /* Войти в критическую
область*/
            down(mutex);
            /* Поместить в буфер
созданный элемент */
            item=get_item();
            /* Выход из критической
области */
            up(mutex);
            /* Увеличить empty */
            up(empty);
            /* Обработка элемента */
            consum_item(item);
      }
}
```

В представленном в примере решении используются три семафора: один для подсчета заполненных сегментов буфера (full), другой для подсчета пустых сегментов (empty), а третий предназначен для исключения одновременного доступа к буферу производителя и потребителя (mutex). Значение счетчика full исходно равно нулю, счетчик empty равен числу сегментов в буфере, а mutex равен 1. Рассмотрим действия обоих процессов пошагово.

Процесс производитель (producer) создаёт новый элемент, чтобы потом поместить его в буфер. Следующим шагом семафор, указывающий количество свободных элементов буфера уменьшается. При этом если уменьшить этот семафор нельзя (в случае, когда он равен нулю), производитель будет ожидать, пока не появится хоть один свободный элемент. Если же уменьшение прошло удачно, производитель сообщает о том, что он входит в критическую секцию (вход будет произведён лишь в том случае, если потребитель не находится в критической секции). Исполняя критическую секцию, производитель помещает созданный элемент в буфер, после чего сообщает о выходе из критической секции и увеличивает семафор, отражающий число заполненных элементов буфера.

Процесс потребитель (consumer) уменьшает значение семафора, указывающее количество занятых элементов буфера. При этом, если семафор нельзя уменьшить (он равен нулю), потребитель будет ожидать, пока в буфере не появится хоть один заполненный элемент. Если уменьшение прошло успешно, потребитель уменьшает мьютекс, тем самым сообщая о входе в критическую секцию. Вход в критическую секцию произойдёт только если производитель не находится в критической секции, в противном случае потребитель будет ожидать, пока производитель не выйдет из критической секции. Исполняя критическую секцию, потребитель забирает из буфера элемент, после чего сообщает о выходе из критической секции и увеличивает семафор, отражающий число свободных элементов буфера. Завершающим этапом, потребитель обрабатывает полученный из буфера элемент.

В примере семафоры использовались двумя различными способами. Это различие достаточно значимо, чтобы сказать о нем особо. Семафор mutex используется для реализации взаимного исключения, то есть для исключения одновременного обращения к буферу и связанным переменным двух процессов.

3.2. Взаимоблокировка

В компьютерных системах существует большое количество ресурсов, каждый из которых в конкретный момент времени может использоваться только одним процессом. В качестве таких примеров можно привести принтеры, накопители на магнитной ленте и элементы внутренних таблиц системы. Появление двух процессов, одновременно передающих данные на принтер, приведет к печати бессмысленного набора символов. Наличие двух процессов, использующих один и тот же элемент таблицы файловой системы, обязательно станет причиной разрушения файловой системы. Поэтому все операционные системы обладают способностью предоставлять процессу эксклюзивный доступ (по крайней мере, временный) к определенным ресурсам.

Часто для выполнения прикладных задач процесс нуждается в исключительном доступе не к одному, а к нескольким ресурсам.

Пример 3.5. Взаимоблокировка

Предположим, что каждый из двух процессов хочет записать отсканированный документ на компакт-диск. Процесс А запрашивает разрешение на использование сканера и получает его. Процесс В запрограммирован по-другому, поэтому сначала запрашивает устройство для записи компакт-дисков и также получает его. Затем процесс А обращается к устройству для записи компакт-дисков, но запрос отклоняется до тех пор, пока это устройство занято процессом В. К сожалению, вместо того чтобы освободить устройство для записи компакт-дисков, В запрашивает сканер. В этот момент процессы заблокированы и будут вечно оставаться в этом состоянии. Такая ситуация называется тупиком, тупиковой ситуацией или взаимоблокировкой.

В сложно организованной системе с большим количеством ресурсов и одновременно исполняемых процессов взаимоблокировка весьма вероятна. При этом под ресурсами (понятие ресурса было введено в первом параграфе) можно понимать, в том числе, и сами процессы. Обобщая, ресурсом можно назвать любой объект, к которому может получить доступ процесс.

Ресурсы можно разбить на два класса:

1. *Выгружаемым* назовём такой ресурс, который можно безболезненно забирать у владеющего им процесса. К такому ресурсу можно отнести, например, память. Пока процесс приостановлен, можно безболезненно выгрузить содержимое памяти на диск, при этом отдав освободившийся объём памяти другому процессу. Когда настанет момент восстановить процесс, память которого была выгружена, выгруженные данные с диска прочитываются и помещаются в память, после чего процесс запускается вновь.
2. *Невыгружаемым* назовём такой ресурс, который нельзя забрать у процесса, не потеряв результатов работы этого процесса. К примеру, если в момент записи отнять у процесса записывающее устройство и передать его в пользование другому процессу, все данные первого процесса будут потеряны безвозвратно.

Справедливости ради заметим, что взаимная блокировка может возникнуть лишь на втором классе ресурсов, поскольку если в тупиковой ситуации задействованы ресурсы первого класса, ситуацию можно разрешить, принудительно выгрузив ресурс, позаботившись при этом о его дальнейшем восстановлении.

Последовательность событий, необходимых для использования ресурса, представлена ниже в абстрактной форме.

1. Запрос ресурса.
2. Использование ресурса.
3. Возврат ресурса.

Если ресурс недоступен, когда он требуется, то запрашивающий его процесс вынужден ждать. В некоторых операционных системах при неудачном обращении к ресурсу процесс автоматически блокируется и возобновляется только после того, как ресурс становится доступным. В других системах запрос ресурса, получивший отказ, возвращает код ошибки, тогда вызывающий процесс может подождать немного и повторить попытку заново.

Процесс, чье обращение к ресурсу оказалось неудачным, обычно дальше попадает в короткий цикл: запрос ресурса, затем режим ожидания, потом очередная попытка. Хотя этот процесс не блокирован, он во всех смыслах ведет себя, как заблокированный, поскольку не может выполнить никакой полезной работы. В дальнейшем мы будем предполагать, что когда процессу отказывается в предоставлении ресурса, он переходит в режим ожидания.

Истинная природа запросов ресурсов сильно зависит от системы. В некоторых системах существует системный вызов request, позволяющий процессам запрашивать ресурсы явно. В других случаях единственный вид ресурсов, известных операционной системе, — это специальные файлы, которые в каждый данный момент времени могут открыть только один процесс. Они открываются с помощью обычного вызова open. Если файл уже используется, вызывающая программа блокируется до тех пор, пока текущий владелец файла не закроет его.

Для некоторых видов ресурсов, таких как записи в базе данных, управление использованием ресурсов зависит от самих пользовательских процессов. Легко заметить, что последовательность событий, необходимых для использования ресурса очень напоминает последовательность действий для использования ячеек буфера и китайских палочек в рассмотренных выше задачах «Проблема производителя и потребителя» и «Проблема обедающих философов», соответственно. Очевидно, что один из способов, делающих возможным пользовательское управление, заключается в присоединении семафора к каждому из ресурсов. Рассмотрим теперь примеры использования ресурсов и взаимоблокировки в терминологии семафоров. Все семафоры в исходном состоянии равны 1. С тем же успехом можно использовать мьютексы. Три шага, перечисленные выше, выполняются следующим образом: сначала для запроса ресурса используется вызов down(), примененный к семафору, затем программа использует ресурс и, наконец, используется вызов up() для его освобождения. Эти шаги представлены в листинге 3.2.1(а).

Листинг 3.2.1 (а). Использование семафора для защиты ресурсов (один ресурс)

```
typedef int semaphore;
semaphore resource_1;

void process_A(void) {
      down(&resource_1);
      use_resource_1();
      up(&resource_1);
}
```

Листинг 3.2.1 (б). Использование семафора для защиты ресурсов (два ресурса)

```
typedef int semaphore;
semaphore resource_1;
semaphore resource_2;

void process_A(void) {
      down(&resource_1);
      down(&resource_2);
      use_both_resources();
      up(&resource_2);
      up(&resource_1);
}
```

Иногда процессы нуждаются в двух и более ресурсах. Их можно получать последовательно, как показано в листинге 3.2.1(б). Если требуется больше двух ресурсов, их запрашивают непосредственно один за другим.

Пока все хорошо. Эта схема работает прекрасно до тех пор, пока она касается только одного процесса. Конечно, при наличии всего лишь одного процесса отсутствует необходимость формального приобретения ресурсов, поскольку не возникает соперничества за их использование.

Теперь рассмотрим ситуацию с двумя процессами A и B и двумя ресурсами. В листинге 3.2.2 показаны два сценария: а — оба процесса получают ресурсы в одном и том же порядке; б — они запрашивают ресурсы в разном порядке. Разница может показаться несущественной, но это не так.

В листинге 3.2.2(а) один из процессов запрашивает первый ресурс, опережая второй процесс. Затем этот же процесс успешно получает второй ресурс и выполняет свою работу. Если второй процесс попытается получить ресурс 1 до его освобождения, второй процесс просто будет заблокирован до тех пор, пока не станет доступен ресурс.

Листинг 3.2.2. Код, не приводящий к тупику (а); коде потенциальной взаимоблокировкой (б)

а

```
typedef int semaphore;
semaphore resource_1;
semaphore resource_2;

void process_A(void) {
      down(&resource_1);
      down(&resource_2);
      use_both_resources();
      up(&resource_2);
      up(&resource_1);
}

void process_B(void) {
      down(&resource_1);
      down(&resource_2);
      use_both_resources();
      up(&resource_2);
      up(&resource_1);
}
```

б

```
typedef int semaphore;
semaphore resource_1;
semaphore resource_2;

void process_A(void) {
      down(&resource_1);
      down(&resource_2);
      use_both_resources();
      up(&resource_2);
      up(&resource_1);
}

void process_B(void) {
      down(&resource_2);
      down(&resource_1);
      use_both_resources();
      up(&resource_1);
      up(&resource_2);
}
```

Другая ситуация представлена в листинге 3.2.2(б). Может случиться так, что один из процессов получит оба ресурса и эффективно блокирует другой процесс до завершения своей работы. Однако также может произойти и то, что процесс A займет ресурс 1, а процесс B получит ресурс 2. Теперь, когда они попытаются запросить еще по одному ресурсу, каждый из них будет заблокирован. Ни один из двух процессов не сможет когда-либо заработать снова. Это ситуация взаимоблокировки.

Здесь мы видим, как проявляется несущественное различие в коде программы — последовательность запросов ресурсов, — которое вызывает разницу между работающей программой и программой, завершающейся аварийно, причем с трудно обнаруживаемой причиной ошибки. Поскольку взаимоблокировки могут происходить так просто, на поиски методов борьбы с ними было направлено большое количество исследований.

def *Взаимоблокировка* — это тупиковая ситуация, характеризующаяся тем, что группа процессов ожидает события, которое может вызвать только другой процесс из этой же группы.

Так как все процессы находятся в состоянии ожидания, ни один из них не будет причиной какого-либо события, которое могло бы активировать любой другой процесс в группе, и все процессы продолжают ждать до бесконечности. В этой модели мы предполагаем, что процессы имеют только один поток и что нет прерываний, способных активизировать заблокированный процесс. Условие отсутствия прерываний необходимо, чтобы предотвратить ситуацию, когда тот или иной заблокированный процесс активизируется, скажем, по сигналу тревоги и затем приводит к событию, которое освободит другие процессы в группе.

В большинстве случаев событием, которого ждет каждый процесс, является освобождение какого-либо ресурса, в данный момент занятого другим участником группы. Другими словами, каждый участник в группе процессов, зашедших в тупик, ждет доступа к ресурсу, принадлежащему заблокированному процессу. Ни один из процессов не может работать, ни один из них не может освободить какой-либо ресурс и ни один из них не может возобновиться. Количество процессов и количество и вид ресурсов, имеющихся и запрашиваемых, здесь не важны. Результат остается тем же самым для любого вида ресурсов, аппаратных и программных.

Необходимые условия возникновения взаимоблокировки — условия Коффмана (Coffman):

1. *Условие взаимного исключения.* Каждый ресурс в данный момент или отдан ровно одному процессу, или доступен.
2. *Условие удержания и ожидания.* Процессы, в данный момент удерживающие полученные ранее ресурсы, могут запрашивать новые ресурсы.
3. *Условие отсутствия принудительной выгрузки ресурса.* У процесса нельзя принудительным образом забрать ранее полученные ресурсы. Процесс, владеющий ими, должен сам освободить ресурсы.
4. *Условие циклического ожидания.* Должна существовать круговая последовательность из двух и более процессов, каждый из которых ждет доступа к ресурсу, удерживаемому следующим членом последовательности.

Указанные условия являются необходимыми. То есть, если хоть одно из них не выполняется, то взаимоблокировка никогда не возникнет. Достаточность не имеет места быть: если выполняются все четыре условия, взаимоблокировка может и не произойти, например, если в системе нет процессов, претендующих на одновременное использование одних и тех же ресурсов.

Следует отметить, что каждое условие относится к правилам, установленным в данной конкретной операционной системе. К сожалению, практически невозможно спроектировать операционную систему так, чтобы полностью исключить хоть одно из условий Коффмана заранее. Однако существуют варианты разрушения взаимоблокировок, основанные на принудительном исключении какого-либо из этих условий.

Отслеживать возникновение взаимоблокировок удобно на диаграммах Холта (Holt). Диаграмма Холта представляет собой направленный граф[10], имеющий два типа узлов: процессы (показываются кружочками) и ресурсы (показываются квадратиками). Тот факт, что ресурс получен процессом и в данный момент занят этим процессом, указывается ребром (стрелкой) от ресурса к процессу. Ребро, направленное от процесса, к ресурсу, означает, что процесс в данный момент блокирован и находится в состоянии ожидания доступа к соответствующему ресурсу.

[10] Граф — это пара $< V, R >$, где V — это множество вершин $V = \{v_i\}_{i=1}^{N}$, а R — множество рёбер $R = \{(v_{i_k}, v_{j_k})\}_{k=1}^{M}$, $i_k = \overline{1,M}$, $j_k = \overline{1,M}, k = \overline{1,N}$. Граф называется ориентированным или направленным, если $(v_{i_k}, v_{j_k}) \neq (v_{j_k}, v_{i_k})$. Ребра направленного графа на диаграмме обозначаются стрелками.

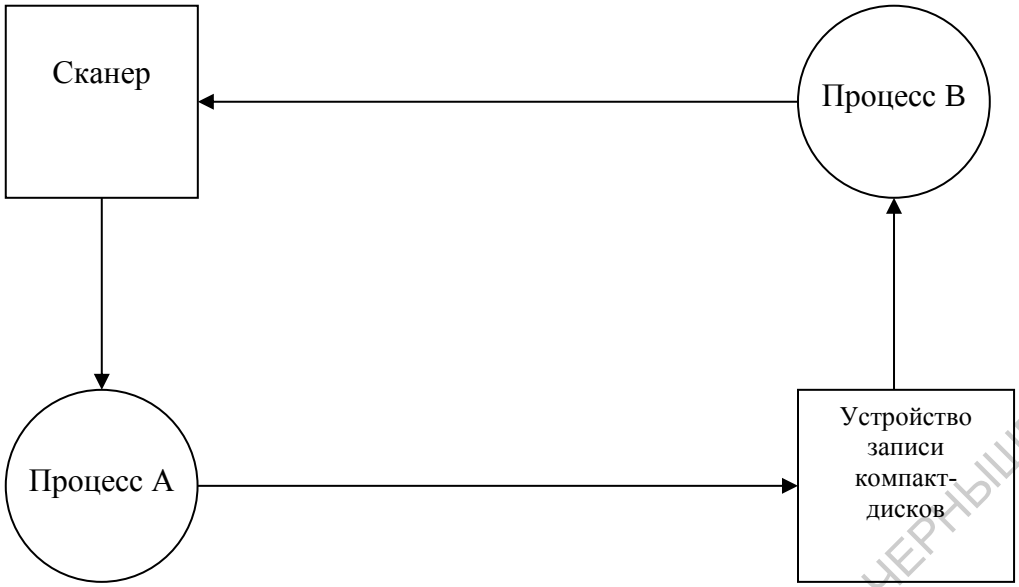


Рисунок 5. Диаграмма Холта для примера 3.5 взаимоблокировки

Критерий взаимоблокировки. Взаимоблокировка имеет место быть, тогда и только тогда, когда диаграмма Холта, отражающая состояния процессов и ресурсов, содержит цикл[11].

В операционных системах реализуются следующие стратегии обработки взаимоблокировок (возможных или произошедших):

1. *Игнорирование.* Не предполагается что-либо делать для предотвращения взаимоблокировки или её разрушения, если она возникла.
2. *Обнаружение и восстановление.* Если взаимоблокировка произошла, обнаружить её и предпринять какие-либо действия.
3. *Динамическое избежание.* Распределять ресурсы системы так, чтобы избежать взаимоблокировок. Всякий раз решение о распределении принимается исходя из текущего состояния системы.
4. *Предотвращение.* Предполагает структурное опровержение одного из четырёх правил Коффмана.

Рассмотрим их более подробно.

Пренебрежением проблемой в целом (страусовый алгоритм)

Если вероятность взаимоблокировки очень мала, то ею легче пренебречь, т.к. код исключения может очень усложнить ОС и привести к большим ошибкам. Также многие взаимоблокировки тяжело обнаружить. Поэтому (и не только) на в далёкой древности серверах часто устанавливают автоматическую перезагрузку (раз в сутки, как правило ночью), если возникнет взаимоблокировка, то после перезагрузки ее не будет.

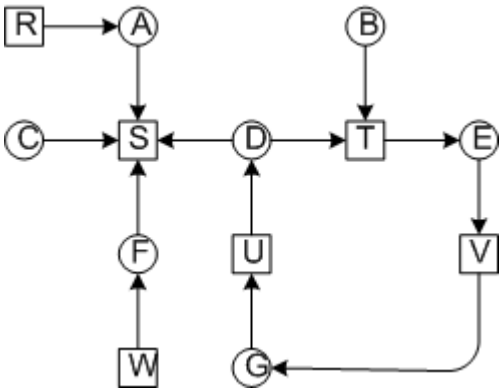
Обнаружение и устранение взаимоблокировок

Система не пытается предотвратить взаимоблокировку, а пытается обнаружить ее и устранить.

Обнаружение взаимоблокировки при наличии одного ресурса каждого типа

Под одним ресурсом каждого типа, подразумевается один принтер, один сканер и один плоттер и т.д. Рассмотрим систему из 7-ми процессов и 6-ти ресурсов.

[11] Говорят что граф содержит цикл, если начиная с некоторой вершины, переходя по рёбрам, можно каким-либо образом вернуться в эту же самую вершину.



Визуально хорошо видна взаимоблокировка, но нам нужно чтобы ОС сама определяла взаимоблокировку. Для этого нужен алгоритм. Рассмотрим один из алгоритмов. Для каждого узла N в графе выполняется пять шагов.

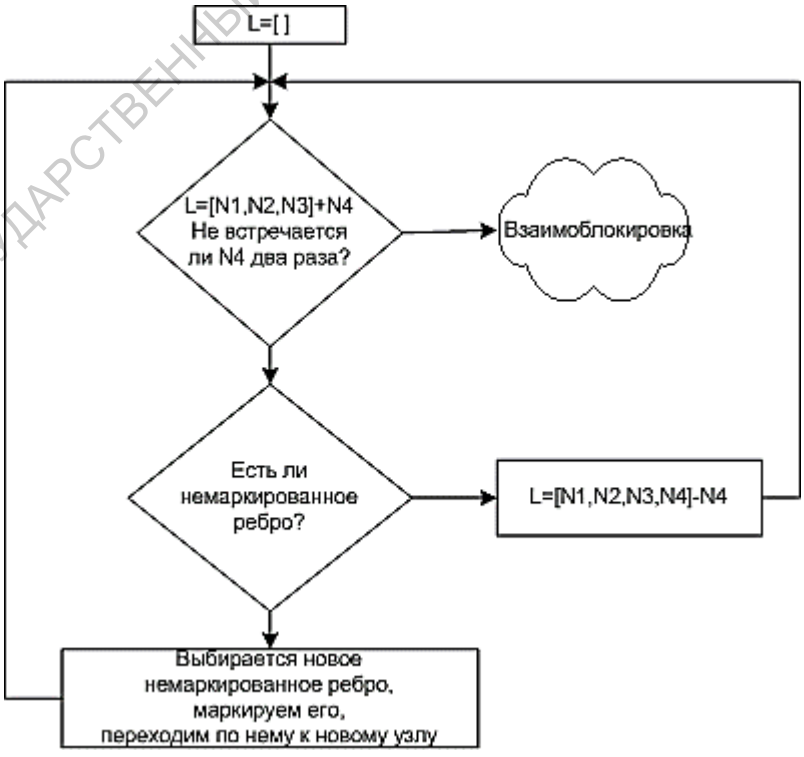
1. Задаются начальные условия: L-пустой список, все ребра не маркированы.

2. Текущий узел добавляем вконец списка L и проверяем количество появления узла в списке. Если он встречается два раза, значит цикл и взаимоблокировка.

3. Для заданного узла смотрим, выходит ли из него хотя бы одно немаркированное ребро. Если да, то переходим к шагу 4, если нет, то переходим к шагу 5.

4. Выбираем новое немаркированное исходящее ребро и маркируем его. И переходим по нему к новому узлу и возвращаемся к шагу 3.

5. Зашли в тупик. Удаляем последний узел из списка и возвращаемся к предыдущему узлу. Возвращаемся к шагу 3. Если это первоначальный узел, значит, циклов нет, и алгоритм завершается.

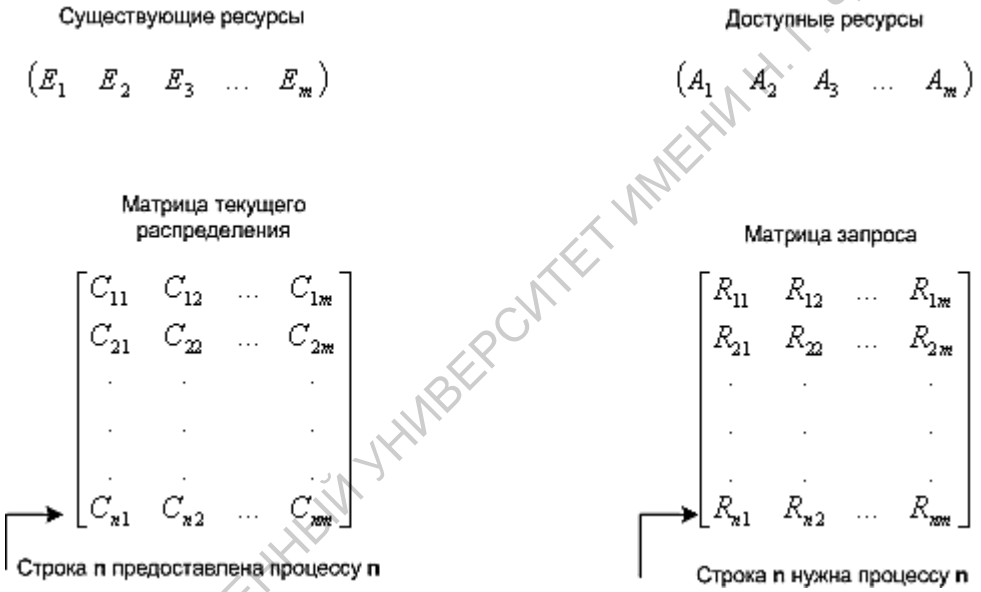


Для нашего случая тупик обнаруживается в списке L=[B,T,E,V,G,U,D,T]

Обнаружение взаимоблокировки при наличии нескольких ресурсов каждого типа

Рассмотрим систему.

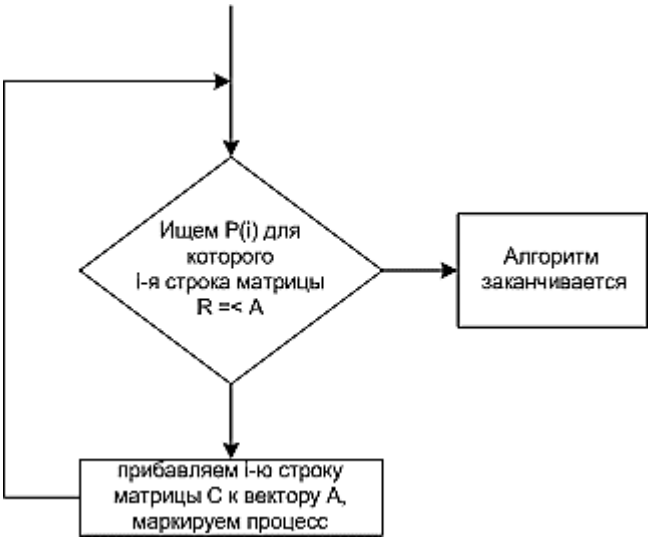
m – число классов ресурсов (например: принтеры это один класс)
n – количество процессов
P(n) – процессы
E – вектор существующих ресурсов
E(i) – количество ресурсов класса i
A – вектор доступных (свободных) ресурсов
A(i) –количество доступных ресурсов класса i
C – матрица текущего распределения (какому процессу, какие ресурсы принадлежат)
R – матрица запросов (какой процесс, какой ресурс запросил)
C(ij) – количество экземпляров ресурса j, которое занимает процесс P(i).
R(ij) – количество экземпляров ресурса j, которое хочет получить процесс P(i).



$$\sum_{i=1}^{n} C_{ij} + A_j = E_j$$

Общее количество ресурсов равно сумме занятых и свободных ресурсов

Рассмотрим алгоритм поиска тупиков при наличии нескольких ресурсов каждого типа.



Если остаются не маркированные процессы, значит, есть тупик. Рассмотрим работу алгоритма на реальном примере.

$$E = \begin{matrix}(\text{принтеры} & \text{плоттеры} & \text{сканеры} & \text{компакт диски}\end{matrix}$$
$$\begin{pmatrix}4 & 2 & 3 & 1\end{pmatrix}$$

$$A = \begin{matrix}(\text{принтеры} & \text{плоттеры} & \text{сканеры} & \text{компакт диски}\end{matrix}$$
$$\begin{pmatrix}2 & 1 & 0 & 0\end{pmatrix}$$

Матрица текущего распределения

$$C = \begin{bmatrix}0 & 0 & 1 & 0\\ 2 & 0 & 0 & 1\\ 0 & 1 & 2 & 0\end{bmatrix}$$

Матрица запросов

$$R = \begin{bmatrix}2 & 0 & 0 & 1\\ 1 & 0 & 1 & 0\\ 2 & 1 & 0 & 0\end{bmatrix}$$

Используем алгоритм:

1. Третий процесс может получить желаемые ресурсы, т.к. R (2 1 0 0) = A (2 1 0 0)
2. Третий процесс освобождает ресурсы. Прибавляем их к **A**. A = (2 1 0 0) + (0 1 2 0) =(2 2 2 0). Маркируем процесс.
3. Может выполняться процесс **2**. По окончании A=(4 2 2 1).
4. Теперь может работать первый процесс.

Тупиков не обнаружено.

Если рассмотреть пример, когда второму процессу требуются ресурсы (1 0 3 0), то два процесса окажутся в тупике.

Когда можно искать тупики:

- Когда запрашивается очередной ресурс (очень загружает систему)
- Через какой то промежуток времени (в интерактивных системах пользователь это ощутит)
- Когда загрузка процессора слишком велика

Выход из взаимоблокировки

Восстановление при помощи принудительной выгрузки ресурса

Как правило, требует ручного вмешательства (например: принтер).

Восстановление через откат

Состояние процессов записывается в контрольных точках, и в случае тупика можно сделать откат процесса на более раннее состояние, после чего он продолжит работу снова с этой точки. С принтером опять будут проблемы.

Восстановление путем уничтожения процесса

Самый простой способ.

Но с принтером опять будут проблемы.

В реальных системах они не годятся.

Динамическое избежание взаимоблокировок

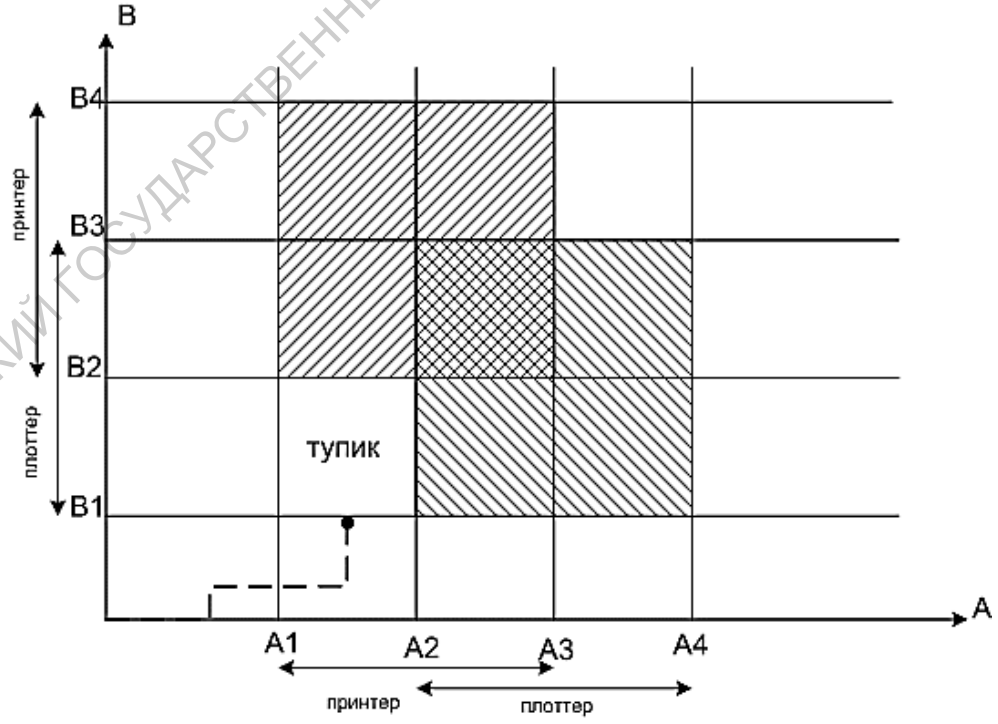
В этом способе ОС должна знать, является ли предоставление ресурса безопасным или нет.

Траектории ресурсов

Рассмотрим модель из двух процессов и двух ресурсов.

A1 – запрос принтера процессом A

A2 – запрос плоттера процессом A
A3 – освобождение принтера процессом A
A4 – освобождение плоттера процессом A
B1 – запрос плоттера процессом B
B2 – запрос принтера процессом B
B3 – освобождение плоттера процессом B
B4 – освобождение принтера процессом B



Динамическое избежание взаимоблокировок

Т.к. процессор предоставляется поочередно, траектория может продолжаться только параллельно осям.

Чтобы избежать тупика, процессам надо обойти прямоугольник, охватывающий всю заштрихованную область.

Безопасные и небезопасные состояния

В безопасном состоянии система может гарантировать, что все процессы закончат свою работу. Рассмотрим систему.

10 экземпляров ресурса
3 процесса

	имеет	max
A	3	9
B	2	4
C	2	7

Свободно:3

	имеет	max
A	3	9
B	4	4
C	2	7

Свободно:1

	имеет	max
A	3	9
B	0	-
C	2	7

Свободно:5

	имеет	max
A	3	9
B	0	-
C	5	7

Свободно:0

	имеет	max
A	3	9
B	0	-
C	0	-

Свободно:7

Процесс A занял 3 экземпляра, но ему необходимо 9.

В этой ситуации можно спланировать так, сначала запустить процесс B, потом C и потом A.

Процессы заканчивают работу без тупиковой ситуации.

Рассмотрим другую ситуацию.

Процесс A занял 4 экземпляра.

	имеет	max
A	3	9
B	2	4
C	2	7

Свободно:3

	имеет	max
A	4	9
B	2	4
C	2	7

Свободно:2

	имеет	max
A	4	9
B	4	4
C	2	7

Свободно:0

	имеет	max
A	4	9
B	0	-
C	2	7

Свободно:4

Возникает небезопасное состояние.

В принципе, процесс A может в какой-то момент ресурс освободить и тупика не возникнет. Видно, что в этом случае не стоило давать ресурс процессу A.

Алгоритм банкира для одного вида ресурсов

"Банкира", потому что аналогия такая, клиенты-процессы, кредиты-ресурсы.

Рассмотрим систему:

Банкир может дать 10 кредитов (ресурсы).

К нему попеременно обращаются 4-ре клиента.

	имеет	max
A	0	6
B	0	5
C	0	4
D	0	7

Свободно:10

	имеет	max
A	1	6
B	1	5
C	2	4
D	4	7

Свободно:2

	имеет	max
A	1	6
B	2	5
C	2	4
D	4	7

Свободно:1

Алгоритм банкира:

1. Банкиру поступает запрос от клиента на получение кредита
2. Банкир проверяет, приводит ли этот запрос к небезопасному состоянию.
3. Банкир в зависимости от этого дает или отказывает в кредите.

Алгоритм банкира

Алгоритм банкира для несколько видов ресурсов

Рассмотрим систему:

C

	принтеры	плоттеры	сканеры	устройства компакт-дисков
A	3	0	1	1
B	0	1	0	0
C	1	1	1	0
D	1	1	0	1
E	0	0	0	0

Распределенные ресурсы

	принтеры	плоттеры	сканеры	устройства компакт-дисков
A	1	1	0	0
B	0	1	1	2
C	3	1	0	0
D	0	0	1	0
E	2	1	1	0

Ресурсы, которые еще нужны

R

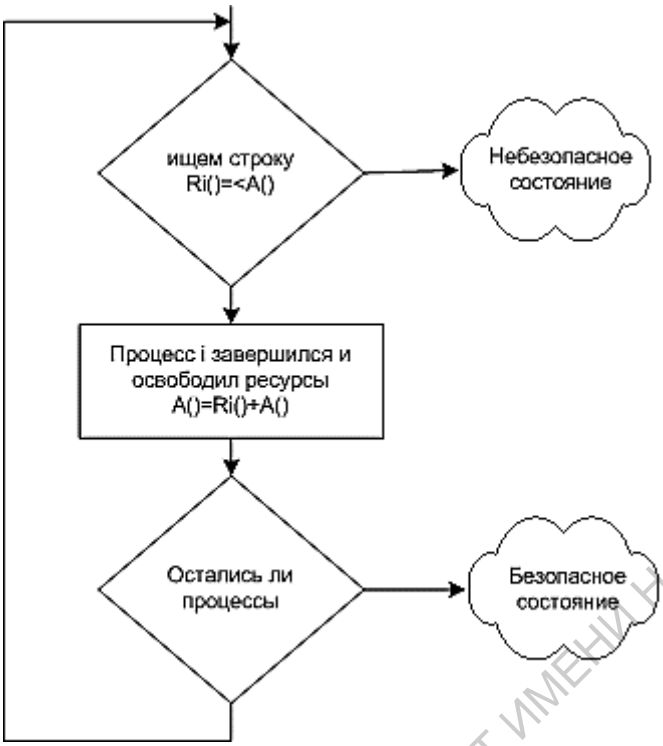
вектора:

E=(6342) - существующие ресурсы

P=(5322) - занятые ресурсы

A=(1020) - доступные ресурсы

Алгоритм поиска безопасного или небезопасного состояния:



Алгоритм банкира для несколько видов ресурсов

Если состояние безопасное, то ресурс дать можно, если нет то нельзя.

На практике все эти алгоритмы тяжело реализовать.

Предотвращение четырех условий, необходимых для взаимоблокировок

Предотвращение условия взаимного исключения

Можно минимизировать количество процессов, борющихся за ресурсы.

Например, с помощью спулинга для принтера, когда только демон принтера работает с принтером.

Предотвращение условия удержания и ожидания

Один из способов достижения этой цели, это когда процесс должен запрашивать все необходимые ресурсы до начала работы. Если хоть один ресурс недоступен, то процессу вообще ничего не предоставляется.

Предотвращение условия отсутствия принудительной выгрузки ресурса

Можно выгружать ресурсы, но могут быть проблемы с принтером.

Предотвращение условия циклического ожидания

Способы предотвращения:

- Процесс сначала должен освободить занятый ресурс, прежде чем занять новый.
- Можно пронумеровать все ресурсы (и упорядочить), и процессы должны запрашивать ресурсы только по возрастающему порядку.

3.3. Управление памятью

О памяти компьютерной системы

Память — важнейший ресурс вычислительной системы, требующий эффективного управления. Несмотря на то, что в наши дни память среднего домашнего компьютера в тысячи раз превышает память больших ЭВМ 70-х годов, программы увеличиваются в размере быстрее, чем память. Достаточно сказать, что только операционная система занимает сотни Мбайт, не говоря о прикладных программах и базах данных, которые могут занимать в вычислительных системах десятки и сотни Гбайт.

Со времен создания ЭВМ фон Неймана основная память в компьютерной системе организована как линейное (одномерное) адресное пространство, состоящее из последовательности слов, а позже байтов. Ячейки памяти занумерованы, начиная с нуля. Номер ячейки называется её физическим адресом.

В идеале программист хотел бы иметь неограниченную в размере и скорости память, которая была бы энергонезависимой, т.е. сохраняла бы своё содержимое при выключении питания, при этом недорого стоила.

В реальности пока такой памяти нет. В то же время на любом этапе развития технологии производства запоминающих устройств действуют следующие достаточно устойчивые соотношения:

- чем меньше время доступа, тем дороже бит;
- чем выше емкость, тем ниже стоимость бита;
- чем выше емкость, тем больше время доступа.

Чтобы найти выход из сложившейся ситуации, необходимо опираться не на отдельно взятые компоненты или технологию, а выстроить иерархию запоминающих устройств, показанную на рис. 3.1. При перемещении слева направо происходит следующее:

- снижается стоимость бита;
- возрастает емкость;
- возрастает время доступа;
- снижается частота обращений процессора к памяти.

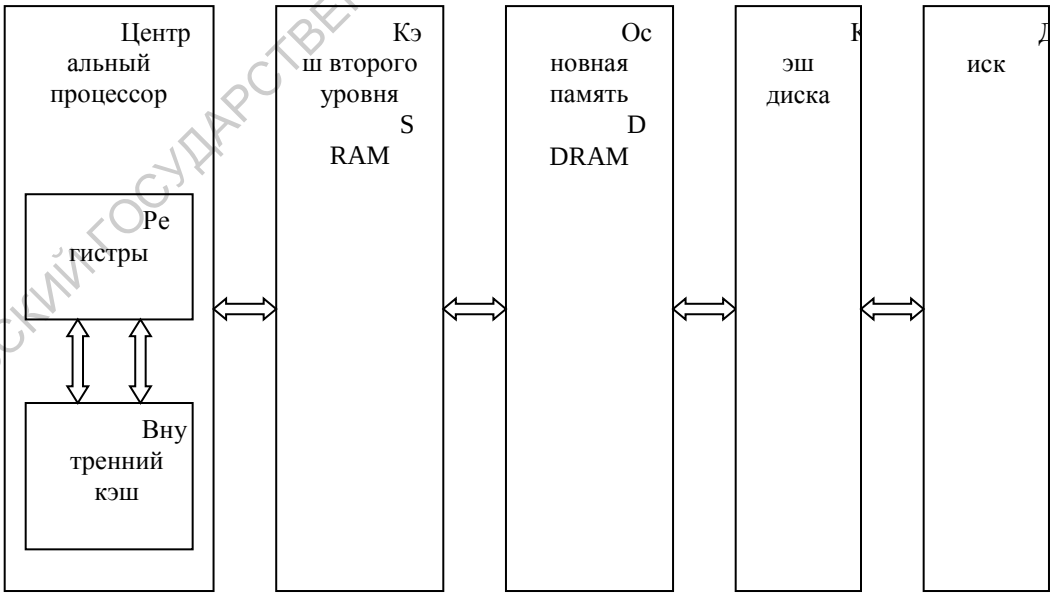


Рисунок 6. Иерархия запоминающих устройств

Такие закономерности организации и характеристики памяти накладывают свой отпечаток на протекающие вычислительные процессы. Некоторые вида специально введены как промежуточные (кэширующие) между различными видами памяти, сильно отличающимися по времени доступа. Самым наглядным примером в этом случае может служить кэш жесткого диска (обычный объём её 8-16Мб по данным на 2006г.). Доступ к информации на жестком диске происходит существенно медленнее, чем работа с оперативной памятью. Если пользователь посылает запрос на чтение части файла с жесткого диска, в кэш диска читается весь файл, а передаётся в оперативную память лишь запрошенная часть. При следующем запросе на оставшуюся часть файла, данные будут взяты из кэша. При этом доступ к кэш-памяти диска происходит быстрее, чем чтение с диска.

Виртуализация памяти

Объём оперативной памяти существенно сказывается на вычислительном процессе, так как ограничивает количество одновременно протекающих в системе процессов. Большое количество задач требует большого объёма оперативной памяти. В условиях недостатка памяти был предложен метод организации вычислительного процесса, при котором образы некоторых процессов целиком или частично временно выгружаются на диск.

Очевидно, что имеет смысл временно выгружать неактивные процессы, находящиеся в ожидании каких-либо ресурсов, в том числе очередного кванта времени центрального процессора. К моменту, когда пройдет очередь выполнения выгруженного процесса, его образ возвращается с диска в оперативную память. Если при этом обнаруживается, что свободного места в оперативной памяти не хватает, то на диск выгружается другой процесс.

Такая подмена (виртуализация) оперативной памяти дисковой памятью позволяет повысить уровень мультипрограммирования, поскольку объем оперативной памяти теперь не столь жестко ограничивает число одновременно выполняемых процессов. При этом суммарный объем оперативной памяти, занимаемой образами процессов, может существенно превосходить имеющийся объем оперативной памяти.

В данном случае в распоряжение прикладного программиста предоставляется виртуальная оперативная память, размер которой намного превосходит реальную память системы и ограничивается только возможностями адресации используемого процесса (в ПК на базе Pentium 232 можно адресовать 4 Гбайт памяти). Вообще виртуальным (кажущимся) называется ресурс, обладающий свойствами (в данном случае большой объем ОП), которых в действительности у него нет.

Виртуализация оперативной памяти осуществляется совокупностью аппаратных и программных средств вычислительной системы (схемами процессора и операционной системой) автоматически без участия программиста и не сказывается на логике работы приложения.

Виртуализация памяти возможна на основе двух возможных подходов:

1. *свопинг (swapping)* — образы процессов выгружаются на диск и возвращаются в оперативную память целиком;
2. *виртуальная память (virtual memory)* — между оперативной памятью и диском перемещаются части образов (сегменты, страницы, блоки и т. п.) процессов.

Недостатки свопинга:

- избыточность перемещаемых данных и отсюда замедление работы системы и неэффективное использование памяти;
- невозможность загрузить процесс, виртуальное пространство которого превышает имеющуюся в наличие свободную память.

Достоинство свопинга по сравнению с виртуальной памятью — меньшие затраты времени на преобразование адресов в кодах программ, поскольку оно делается один раз при загрузке с диска в память (однако это преимущество может быть незначительным, т.к. выполняется при очередной загрузке только часть кода и полностью преобразовывать код может быть и не надо).

Виртуальная память не имеет указанных недостатков, но ее ключевой проблемой является преобразование виртуальных адресов в физические. На это преобразование существенно затрачивается время, если не принять специальных мер.

Функции ОС по управлению памятью

Под памятью в данном случае подразумевается оперативная (основная) память компьютера. В ранних операционных системах функции по управлению памятью сводились к решению простых задачам: загрузить программу в память, выгрузить из памяти и т.п. С появлением многозадачных, многопоточных операционных систем, задачи качественно усложнились и число существенно возросло.

Функциями ОС по управлению памятью в многозадачных и многопоточных системах являются:

1. *Отслеживание (учет) свободной и занятой памяти*.

   С этой целью используют так называемые *карты памяти*, указывающие какие области памяти свободны, а какие заняты. Для отслеживания свободной и занятой памяти могут использоваться различные механизмы. Наиболее распространены *битовые массивы* и *списки свободных участков*.

   При учёте с помощью механизма битового массива, память разделяется на области определённой длины, например, 64Кб. Каждому блоку в массиве соответствует 1 бит. Если этот бит равен 0, это означает что блок свободен, если 1, то соответствующий блок занят.

   В случае учёта списком свободных участков, операционная система организует связный список, каждым элементом которого является структура, состоящая из двух чисел (Adr, Len), первое из которых указывает адрес начала свободного участка памяти, а второе — его длину в байтах.
2. *Первоначальное и динамическое выделение памяти* процессам приложений и самой операционной системе и освобождение памяти по завершении процессов.

   При запуске процесса операционная система выделяет свободную область памяти, куда будет загружен процесс, загружает в неё данные этого процесса. При завершении процесса, операционная система должна выгрузить его из памяти и пометить что область памяти, которую процесс занимал, теперь свободна.

   Набор действий кажется не столь сложным, но это ложная простота. Например, действие по выбору свободной области памяти, куда будет загружен процесс, на самом деле может происходить согласно различным стратегиям ("выделить любой участок нужной длины", "выделить участок, имеющий минимальный достаточный размер", "выделить участок, имеющий максимальный достаточный").
3. *Настройка адресов программы на конкретную область физической памяти.*

   Для идентификации переменных и команд на разных этапах жизненного цикла программы используются *символьные имена*, *виртуальные* (математические, условные, логические — все это синонимы) и *физические адреса*.

   Символьные имена присваивает пользователь при написании программ на алгоритмическом языке или ассемблере. Виртуальные адреса вырабатывают транслятор, переводящий программу на машинный язык. Поскольку во время трансляции не известно, в какое место оперативной памяти будет загружена программа, то транслятор присваивает переменным и командам виртуальные (условные) адреса, считая по умолчанию, что начальным адресом программы будет нулевой адрес.

   Физические адреса соответствуют номерам ячеек оперативной памяти, где в действительности будут расположены переменные и команды.

   Совокупность виртуальных адресов процесса называется *виртуальным адресным пространством*. Диапазон адресов виртуального пространства у всех процессов один и тот же и определяется разрядностью адреса процессора (для Pentium

адресное пространство составляет объем, равный 2^{32} байт, с диапазоном адресов от 00000000_{16} до $FFFFFFFF_{16}$).

Существует два принципиально отличающихся подхода к преобразованию виртуальных адресов в физические.

В первом случае такое преобразование выполняется один раз для каждого процесса во время начальной загрузки программы в память. Преобразование осуществляет перемещающий загрузчик на основании имеющихся у него данных о начальном адресе физической памяти, в которую предстоит загружать программу, а также информации, предоставляемой транслятором об адресно-зависимых элементах программы.

Второй способ заключается в том, что программа загружается в память в виртуальных адресах. Во время выполнения программы при каждом обращении к памяти операционная система преобразует виртуальные адреса в физические.

4. *Виртуализация памяти*.

   В случае, когда не хватает объёма реальной памяти для загрузки процесса, диспетчер памяти частично или полностью вытесняет из памяти одни из приостановленных процессов на диск. Затем, при возобновлении вытесненного процесса, диспетчер должен восстановить его образ в памяти, причём если памяти не хватает, вытеснить другой процесс.

   Такой механизм виртуализации памяти уже был описан ранее. Задачи по отслеживанию выгруженных образов памяти и корректному их восстановлению при возобновлении процесса — решаются диспетчером виртуальной памяти.

5. *Защита памяти*, выделенной процессу от возможных вмешательств со стороны других процессов.

   Каждый процесс должен работать в своём адресном пространстве, не вмешиваясь в адресное пространство других процессов. За этим следит операционная система. В случае обращения процесса по адресу за пределами его адресного пространства, операционная система не позволит выполнить обращение и остановит процесс, сообщив о произошедшей ошибке адресации.

Замечание 3.1.

Иногда процессам требуется производить чтение из областей памяти других процессов (например, при антивирусной проверке), или передавать данные другим процессам. Такие ситуации относятся к особому случаю межпроцессного взаимодействия и выполняются специальными системными вызовами.

6. *Дефрагментация памяти.*

   В процессе работы операционной системы множество программ загружаются в память и выгружаются из неё. Часто возникает ситуация, когда между диапазонами памяти занятыми процессами расположены свободные диапазоны. При это возникает негативный эффект: по сумме свободные диапазоны могут составлять достаточный для загрузки запускаемого вновь процесса объём памяти, но среди этих свободных диапазонов, может не найтись ни одного непрерывного диапазона памяти, куда этот загружаемый процесс полностью поместился бы. Операционная система для устранения этого эффекта перемещает процессы в памяти, при этом, перенастраивая адресно-зависимые части кода этих процессов на новые адреса, куда перемещён процесс.

3.4. Ввод-вывод

Одной из важнейших функций операционной системы является управление всеми устройствами ввода-вывода, подключенными к компьютерной системе. Операционная система должна передавать устройствам команды, перехватывать прерывания и обрабатывать ошибки, она также должна обеспечивать унифицированный абстрактный интерфейс этих устройств в системе.

Устройства ввода-вывода делятся на два типа:

1. *Блок-ориентированные устройства (block-oriented).* Это устройства прямого доступа, которые хранят информацию в блоках фиксированного размера, каждый из которых имеет свой адрес. Адресуемость блоков приводит к тому, что для дисков, являющихся устройствами прямого доступа, появляется возможность кэширования данных в оперативной памяти. Это обстоятельство значительно влияет на общую организацию ввода-вывода для блок-ориентированных драйверов.
2. *Байт-ориентированные устройства (character-oriented).* Устройства, обмен информацией с которыми производится побайтно. К таким устройствам относятся: терминалы, некоторые принтеры, сетевые адаптеры. При таком взаимодействии информация не адресуется и не позволяется производить операцию поиска. Работа с устройством происходит следующим образом: устройство посылает системе последовательности байт, либо, при отправке данных устройству, данный передаются побайтно.

Замечание 3.2.

Некоторые внешние устройства не относятся ни к одному из указанных типов, которые, не адресуемы, но и не порождают потока байтов. Примером может служить *системный таймер*. Это устройство только выдает сигнал прерывания в некоторые моменты времени.

С каждым внешним устройством, как правило, связан его контроллер, выполняющий функции управления устройством. Контроллер обычно выполняет простые функции, например, преобразует поток бит в блоки, состоящие из байтов, и осуществляет контроль и исправление ошибок. Каждый контроллер имеет несколько регистров, называемых портами.

Операционная система обычно имеет дело не с устройством, а с контроллером и выполняет ввод-вывод, записывая команды в регистры контроллера.

Например, контроллер гибкого диска принимает 15 команд, таких как READ, WRITE, SEEK, FORMAT и т.д. Когда команда принята, процессор оставляет контроллер и занимается другой работой. При завершении команды контроллер организует прерывание для того, чтобы передать управление процессором операционной системе, которая должна проверить результаты операции. Процессор получает результаты и статус устройства, читая информацию из регистров контроллера.

Программное обеспечение ввода-вывода обычно организуется в виде четырех уровней. У каждого уровня есть четко очерченная функция, которую он должен выполнять, и строго определенный интерфейс с соседними уровнями:

1. Первичный обработчик прерываний

   Это небольшая часть ядра операционной системы, тесно связанная с аппаратурой. При возникновении прерывания операционная система выполняет ряд действий по сохранению контекста текущего процесса, запуску процедуры-обработчика прерываний, а также выбор нового процесса для выполнения.

2. *Драйверы устройств*

   Драйвером устройства называется программа, которая управляет данным устройством и содержит весь специфический для данного устройства код.

   Драйвер обычно пишется производителем устройства и поставляется вместе с ним.

   Только драйвер устройства знает о его конкретных особенностях. Драйверы принимают от вышележащего уровня некоторые универсальные для всех устройств ввода-вывода команды и переводят их в последовательность команд, которые может выполнить контроллер данного устройства.

Типичным запросом, который может получить драйвер, является чтение n блоков данных. Драйвер жесткого диска должен, например, преобразовать номера блоков в номера цилиндров, головок, секторов, проверить, работает ли мотор, находится ли головка над нужным цилиндром и т.д.

Драйверы блок-ориентированных и бит-ориентированных устройств предоставляют вышележащему уровню различные интерфейсы.

3. *Независимый от устройств слой операционной системы.*

Задачей этого слоя (вместе с двумя нижележащими) является представление всех устройств ввода-вывода в виде единообразной абстракции. Ранее уже упоминались такие унифицированные абстракций, например, файловая абстракция (файловые потоки, специальные файлы).

Типичными функциями данного слоя являются:

- обеспечение общего интерфейса к драйверам устройств;

  Одним из аспектов единообразного интерфейса является именование устройств ввода-вывода. Независимо от устройств программное обеспечение занимается отображением символьных имен устройств на соответствующие драйверы.

  С именованием устройств тесно связан вопрос защиты устройств от работы с ними тех пользователей, у которых нет на это соответствующих полномочий.
- обеспечение независимого размера блока;

  Верхним слоям программного обеспечения неудобно работать с блоками разной величины, поэтому данный слой обеспечивает единый размер блока, например за счет объединения нескольких различных блоков в единый логический блок.
- уведомление об ошибках при работе с устройством.

4. *Пользовательский слой программного обеспечения.*

В этом слое присутствуют программные модули (или даже целостные операционные среды), которые обеспечивают высокоуровневое взаимодействие с соответствующим устройством.

Более детально остановимся на понятии *драйвер*. Первоначально термин «драйвер» применялся в достаточно узком смысле; под драйвером понимается программный модуль, который:

- входит в состав ядра ОС, работая в привилегированном режиме;
- непосредственно управляет внешним устройством, взаимодействуя с его контроллером с помощью команд ввода-вывода компьютера;
- обрабатывает прерывания от контроллера устройства;
- предоставляет прикладному программисту удобный логический интерфейс работы с устройством, абстрагированный от низкоуровневых деталей управления устройством и организации его данных;
- взаимодействует с другими модулями ядра ОС с помощью строго оговоренного интерфейса, описывающего формат передаваемых данных, структуру буферов, способы включения драйвера в состав ОС, способы вызова драйвера, набор общих процедур подсистемы ввода-вывода, которыми драйвер может пользоваться и т. п.

Согласно этому определению драйвер вместе с контроллером устройства и прикладной программой воплощали идею многослойного подхода к организации программного

обеспечения. Контроллер представлял низкий слой управления устройством, выполняющий операции в терминах блоков и агрегатов устройства (например, параметры, передаваемые жесткому диску из примера ранее). Драйвер выполнял более сложные операции, преобразуя данные, адресуемые в терминах номеров цилиндров, головок и секторов диска, в линейную последовательность блоков. В результате прикладная программа работала с данными, преобразованными в достаточно понятную форму — файлами, таблицами баз данных, текстовыми окнами на мониторе и т. п., не вдаваясь в детали представления этих данных в устройствах ввода-вывода.

В описанной схеме драйверы не делились на слои. Постепенно, по мере развития операционных систем и усложнения структуры подсистемы ввода-вывода, наряду с традиционными драйверами в ОС появились так называемые высокоуровневые драйверы, которые располагаются в общей модели подсистемы ввода-вывода над традиционными драйверами. Появление таких драйверов можно считать развитием идеи многоуровневой организации подсистемы ввода-вывода, когда ее функции декомпозируются между несколькими модулями в соседних слоях иерархии (таких примеров много, например, семиуровневая модель сетевых протоколов).

Традиционные драйверы, которые стали называть аппаратными, низкоуровневыми, или драйверами устройств, освобождаются от высокоуровневых функций и занимаются только низкоуровневыми операциями. Эти низкоуровневые операции составляют фундамент, на котором можно построить тот или иной набор операций в драйверах более высоких уровней.

При каком подходе повышается гибкость и расширяемость функции по управлению устройством. Например, если различным приложениям необходимо работать с различными логическими модулями одного и того же физического устройства, то для этого в системе достаточно установить несколько драйверов на одном уровне, работающих над одним аппаратным драйвером. Несколько драйверов, управляющих одним устройством, но на разных уровнях, можно рассматривать как один многоуровневый драйвер.

На практике используют от двух до пяти уровней драйверов, поскольку с увеличением числа уровней снижается скорость выполнения операций ввода-вывода.

Высокоуровневые драйверы оформляются по тем же правилам и придерживаются тех же внутренних интерфейсов, что и аппаратные драйверы. Как правило, высокоуровневые драйверы не вызываются по прерываниям, так как взаимодействуют с устройством через посредничество аппаратных драйверов.

В модулях подсистемы ввода-вывода кроме драйверов могут присутствовать и другие модули, например, дисковый кэш. Достаточно специфичные функции кэша делают нецелесообразным оформление его в виде драйвера, взаимодействующего с другими модулями ОС только с помощью услуг менеджера ввода-вывода. Другим примером модуля, который чаще всего не оформляется в виде драйвера, является диспетчер окон графического интерфейса. Иногда этот модуль вообще выносится из ядра ОС и реализуется в виде пользовательского интерфейса. Таким образом был реализован диспетчер окон в Windows NT 3.5 и 3.51, но этот микроядерный подход заметно замедляет графические операции, поэтому в Windows 4.0 диспетчер окон и высокоуровневые графические драйверы, а также графическая библиотека GDI были перенесены в пространство ядра.

Аппаратные драйверы после запуска операции ввода-вывода должны своевременно реагировать на завершение контроллером заданного действия путем взаимодействия с системой прерывания. Драйверы более высоких уровней вызываются не по прерываниям, а по инициативе аппаратных драйверов или драйверов вышележащего уровня. Не все процедуры аппаратного драйвера нужно вызывать по прерываниям, поэтому драйвер обычно имеет определенную структуру, в которой выделяется секция обработки прерываний (Interrupt Service Routine, ISR), которая и вызывается от соответствующего устройства диспетчером прерываний.

В унификацию драйверов большой вклад внесла ОС UNIX. Упомянутое ранее деление устройств на блок-ориентированные и байт-ориентированные было впервые введено именно в UNIX. Это более общее деление, чем деление на вертикальные подсистемы.

В свое время ОС UNIX сделала очень важный шаг по унификации операций и структуризации программного обеспечения ввода-вывода. В ОС UNIX все устройства рассматриваются как виртуальные (специальные) файлы, что дает возможность использовать общий набор базовых операций ввода-вывода для любых устройств независимо от их специфики. Подобная идея реализована позже в MS DOS, где последовательные устройства — монитор, принтер и клавиатура считаются файлами со специальными именами: con, prn, con.

3.5. Файловые системы

Любое компьютерное приложение получает, хранит и выводит данные. Во время работы процесс может хранить ограниченное количество данных в собственном адресном пространстве, поскольку его емкость ограничена рамками виртуального адресного пространства. Для некоторых приложений, например систем резервирования авиабилетов, систем банковского учета и др., одного только виртуального адресного пространства будет недостаточно.

Кроме того, после завершения работы процесса информация, хранящаяся в его адресном пространстве, теряется. В это же время для ряда приложений (например, баз данных) ее надо хранить длительное время, а иногда даже вечно. Исчезновение данных после завершения процесса для таких приложений неприемлемо. Информация должна сохраняться и при аварийном завершении процесса в случае сбоя компьютера.

Третья проблема состоит в том, что часто необходимо разным процессам одновременно получать доступ к одним и тем же данным (или части данных). Для решения этой проблемы необходимо отделить информацию от процесса.

Таким образом, необходимо хранить данные на устройствах компьютеров (диски, ленты и др.) с соблюдением следующих требований:

- Устройства должны позволять хранить очень большие объемы данных. К таким устройствам относятся жесткие магнитные диски, магнитные ленты, оптические и магнитооптические диски.
- Информация должна длительно и надежно сохраняться после прекращения работы процесса, использующего эту информацию. Долговременность хранения обеспечивается использованием запоминающих устройств, не зависящих от электропитания, а высокая надежность определяется соответствующей организацией операционной системы.
- Несколько процессов должны иметь возможность получения одновременного доступа к информации, т. е. должно быть обеспечено совместное использование данных.

Решение этих проблем состоит в хранении информации организованной в файлы.

def *Файл* — в общем случае, это именованная совокупность данных, хранящаяся на каком-либо носителе информации.

При рассмотрении отдельных файлов и их совокупностей используются следующие понятия:

def *Поле (field)* — элемент данных, содержащий некоторое значение и характеризующееся длиной (фиксированной или переменной) и типом данных. Параметры поля (имя, тип данных, длина) могут храниться в самом поле, в таком случае они будут называться *подполями*.

def *Запись (record)* — набор связанных между собой полей, которые могут быть обработаны как единое целое некоторой прикладной программой.

В рамках этой терминологии можно переопределить понятие *файл* как совокупность однородных записей. Файл рассматривается как единое целое приложениями и пользователем. Обращение к файлу осуществляется по его имени. Пользователь (программист) должен иметь

удобные средства работы с файлами, включая каталоги-справочники, объединяющие файлы в группы, средства поиска файла по различным признакам, набор команд для создания, модификации и удаления файлов. Файл может быть создан одним пользователем, а затем использоваться другим, при этом создатель файла или администратор могут определить права доступа к нему других пользователей. В некоторых системах управления доступом осуществляется на уровне записи, а иногда и на уровне поля.

def *База данных (database)* — набор связанных между собой данных, представленных совокупностью файлов одного или несколько типов. Обычно существует отдельная система управления базой данных (СУБД), независимая от операционной системы, но, тем не менее, она почти всегда использует некоторые программы управления файлами.

Обычно единственным способом работы с файлами является использование системы управления файлами или иначе — *файловой системы (ФС)*. Файловая система — это часть операционной системы, включающая:

- совокупность всех файлов на носителе информации (магнитном или оптическом диске, магнитной ленте и др.);
- наборы структур данных, используемых для управления файлами (каталоги и дескрипторы файлов, таблицы распределения свободного и занятого пространства на диске и др.);
- комплекс системных программных средств, реализующих различные операции над файлами (создание, уничтожение, чтение, запись и др.).

Задачи, решаемые файловой системой, во многом определяются способом организации вычислительного процесса (наиболее простые - в однопрограммных и однопользовательских ОС, наиболее сложные - в сетевых ОС).

В многопрограммных, многопользовательских ОС задачами файловой системы являются:

- соответствие требованиям управления данными и требованиям со стороны пользователей, включающим возможность хранения данных и выполнения операций с ними;
- гарантирование корректности данных, содержащихся в файле;
- оптимизация производительности, как с точки зрения системы (пропускная способность), так и с точки зрения пользователя (время отклика);
- поддержка ввода-вывода для различных типов устройств хранения информации;
- минимизация или полное исключение возможных потерь или повреждений данных;
- защита файлов от несанкционированного доступа;
- обеспечение поддержки совместного использования файлов несколькими пользователями (в том числе средства блокировки файла и его частей, исключение тупиков, согласование копий и т. п.);
- обеспечение стандартизированного набора подпрограмм интерфейса ввода-вывода.

Минимальным набором требований к файловой системе со стороны пользователя диалоговой системы общего назначения можно считать следующую совокупность возможностей, предоставляемую пользователю:

1. Создание, удаление, чтение и изменение файлов.
2. Контролируемый доступ к файлам других пользователей.
3. Управление доступом к своим файлам.
4. Реструктурирование файлов в соответствии с решаемой задачей.
5. Перемещение данных между файлами.
6. Резервирование и восстановление файлов в случае повреждения.
7. Доступ к файлам по символьным именам.

Объекты файловой системы

Файловые системы поддерживают несколько функционально различных типов файлов, в число которых входят обычные файлы, содержащие информацию произвольного характера

(текст, графика, звук и др.), файлы-каталоги, специальные файлы, именованные конвейеры, отображаемые в память файлы и др.

Обычные файлы, или просто файлы или регулярные файлы, содержат информацию, которую в них заносит пользователь или которая образуется в результате работы системных и пользовательских программ. Большинство ОС не контролируют содержимое и структуру регулярных файлов, которые в основном являются ASCII-файлами либо двоичными файлами. ASCII-файлы состоят из текстовых строк. Они могут отображаться на экране и выводиться на печать без какого-либо преобразования, и могут редактироваться практически любым текстовым редактором. Двоичные файлы имеют определенную внутреннюю структуру, которая известна программе, использующей данный файл. При выводе двоичного файла на принтер получается случайный набор символов.

Каталоги — это системные файлы, обеспечивающие поддержку структуры файловой системы. Они содержат системную справочную информацию о наборе файлов, сгруппированных пользователем по какому-либо неформальному признаку. Во многих ОС в каталог могут входить другие файлы, в том числе другие каталоги, за счет чего образуется древовидная структура, удобная для поиска требуемого файла. Каталоги устанавливают соответствие между именами файлов и их характеристиками, используемыми файловой системой для управления файлами. В число таких характеристик входит тип файла, права доступа к файлу, его распоряжение на диске, размер, дата и время создания и др.

Специальные файлы — это фиктивные файлы, ассоциированные с устройствами ввода-вывода, которые используются для унификации механизма доступа к последовательным устройствам ввода-вывода, таким как терминалы, принтеры и др. (например, MS DOS рассматривает монитор и клавиатуру как файлы со стандартным именем con — консоль, а принтер — как файл prn). Блочные специальные файлы используются для моделирования дисков.

Именованные конвейеры (каналы) — циклические буферы, позволяющие выходной файл одной программы соединить со входным файлом другой программы.

Отображаемые файлы — это обычные файлы, отображенные на адресное пространство процесса по указанному виртуальному адресу.

Файлы относятся к абстрактному механизму. Они предоставляют способ сохранять информацию на запоминающем устройстве и считывать ее позднее снова. При этом от пользователя должны скрываться такие детали, как способ и место хранения информации, а также детали работы устройства.

В зависимости от правил, установленных разработчиками файловой системы на имя файла накладываются ограничения по длине и используемым в имени символам. Кроме того, в файловой системе могут присутствовать зарезервированные имена, которые нельзя присваивать файлам (например, для файловой системы NTFS одним из таких имён будет являться $MFT). В некоторых файловых системах имя файла может состоять из нескольких полей (например, в MS DOS имя файла состоит из собственно имени и расширения).

Список литературы

1. Таненбаум Э. С., Херберт Б. Современные операционные системы. 4-е изд. – Издательский дом «Питер», 2015.
2. Таненбаум Э. С., Таненбаум Э. С. Компьютерные сети:[пер. с англ.]. – Издательский дом «Питер», 2012.
3. Таненбаум Э. С. Архитектура компьютера:[пер. с англ.]. – Издательский дом «Питер», 2011.
4. Cannon J. Command Line Kung Fu: Bash Scripting Tricks, Linux Shell Programming Tips, and Bash One-liners. – CreateSpace Independent Publishing Platform, 2014.
5. Cooper M. Advanced Bash Scripting Guide. – Рипол Классик, 2014.
6. Бессонов Л. В., Брагина И. Г. Операционные системы. Компьютерные сети: Пособие для студентов, обучающихся по дополнительной специальности «Компьютерная графика и веб-дизайн» // Саратов: Изд-во «Научная книга», 2009. – 44 с.
7. Брагина И.Г. О применении программных средств в обучении математике /И.Г. Брагина, А.В. Букушева // Наука, образование, общество: актуальные вопросы и перспективы развития: Сборник научных трудов по материалам Международной научно-практической конференции 30 мая 2015 г.: в 3 частях. Часть II. М.: «АР-Консалт», 2015. С. 109-110.
8. Брагина И.Г., Сергеева Н.В., Бессонов Л.В. Основы теории вероятностей: Учебное пособие//Саратов, 2014.
9. Бессонов Л. В., Брагина И. Г. Информационные технологии в профессиональной деятельности преподавателя: Учеб. пособие. – Саратов: Изд-во «Научная книга», 2014. – 28 с.
10. Дмитриев П.О. Практикум по веб-программированию. Часть 1. Теоретическое введение в язык PHP: Учебное пособие для студентов, обучающихся по направлению подготовки бакалавриата 09.03.03 «Прикладная информатика» — Саратов: ООО Издательский Центр «Наука», 2016. — 40 с.
11. Бессонов Л.В. Практикум по веб-программированию. Упражнения и практические задания: Учебно-методическое пособие для студентов, обучающихся по направлению подготовки бакалавриата 09.03.03 «Прикладная информатика» — Саратов: ООО Издательский Центр «Наука», 2016. — 40 с.
12. Донник А.М. Операционные системы. Практикум: Учебное пособие для студентов, обучающихся по направлениям подготовки бакалавриата 09.03.03 «Прикладная информатика», 38.03.05 «Бизнес-информатика», 02.03.01 «Математика и компьютерные науки» — Саратов: ООО Издательский Центр «Наука», 2016. — 40 с.

Учебное издание

Бессонов Л. В.

Операционные системы

Опорный конспект лекций

Учебное пособие для студентов, обучающихся по направлениям подготовки бакалавриата 09.03.03 «Прикладная информатика», 38.03.05 «Бизнес-информатика»

Подписано в печать 01.12.2016. Формат 60x84 1/16. Бумага офсетная.
Гарнитура Times New Roman. Печать RISO. Объем 2,75 печ. л.
Тираж 100 экз. Заказ № 207.

ООО Издательский Центр «Наука»
410012, г. Саратов, ул. Пугачевская, 117, оф. 50

Отпечатано с готового оригинал-макета
Центр полиграфических и копировальных услуг
Предприниматель Серман Ю.Б. Свидетельство № 3117
410012, Саратов, ул. Московская, д.152, офис 311, тел. 26-18-19