

Операционные системы

- Материалы: [Курс: Операционные системы 2 курс, ПИ \(bsu.by\)](#)
- Вопросы: trubach@bsu.by
- Кафедра технологий программирования

- 17 лекций \times 2 часа = 34 часа
- Экзамен по итогу семестра, без зачета
- Для допуска к экзамену необходимо иметь оценку ≥ 4 по лабораторным занятиям
 - Студенты с оценкой < 4 к экзамену не допускаются и сначала пересдают долги своему преподавателю в сентябре, а затем уже идут на экзамен

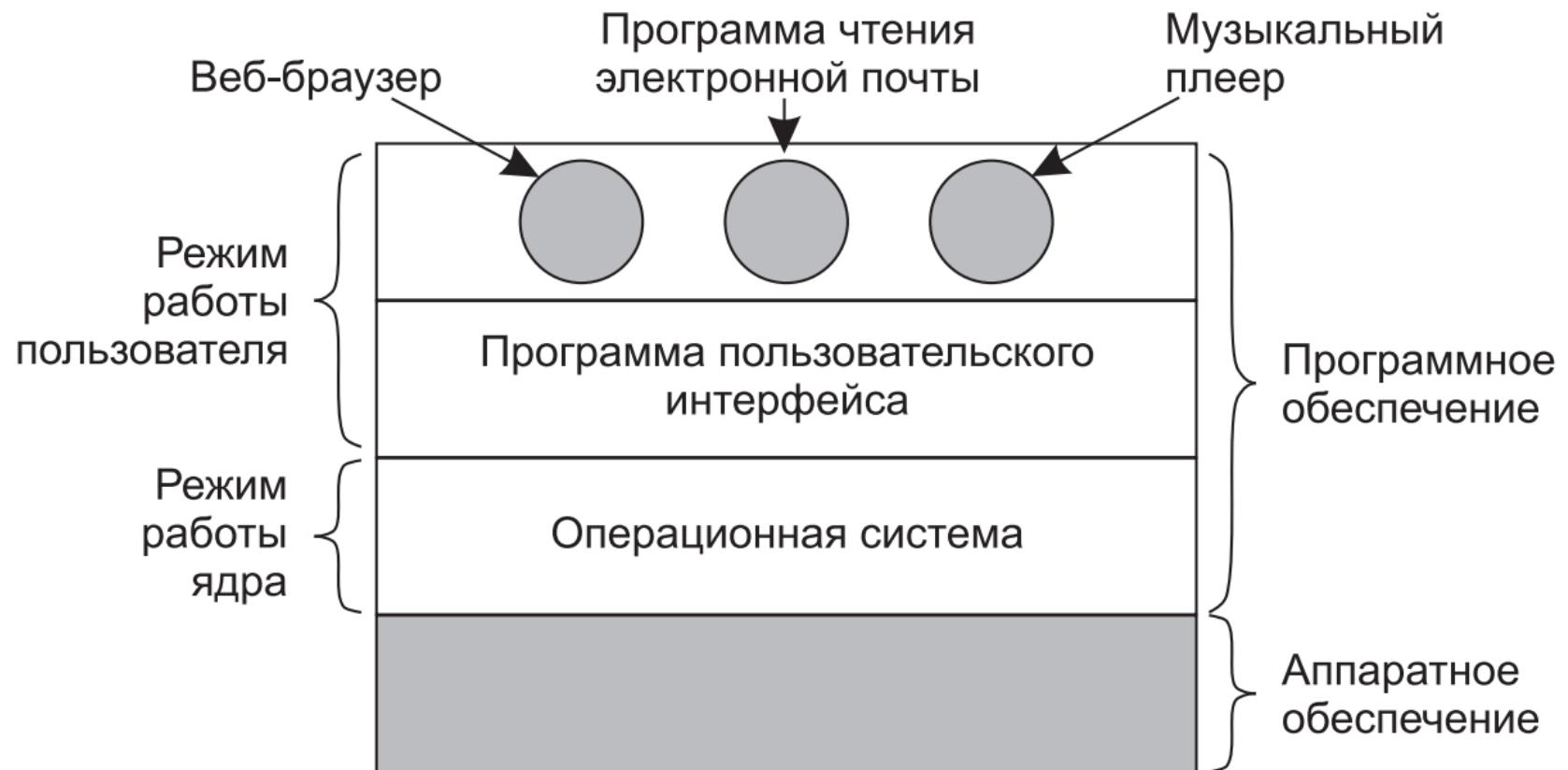
Основные вопросы курса

- Устройство операционной системы ОС Windows
- Процессы и потоки в ОС Windows
 - Понятие процессов и потоков
 - Планирование процессов и потоков
 - Синхронизация процессов и потоков
 - Коммуникация между процессами
- Управление памятью в ОС Windows
- Файловая система ОС Windows
- Устройство операционной системы ОС Linux

Литература

- MSDN <https://docs.microsoft.com/en-us/windows/win32/>
- Рихтер Дж., Windows для профессионалов. Создание эффективных Win32-приложений с учетом специфики 64-разрядной версии Windows
- Столингс В., Операционные системы
- Бэкон Д., Харрис Т., Операционные системы. Параллельные и распределенные системы.
- Танненбаум А., Операционные системы
- И многие другие...

Операционная система



Режимы работы

- Большинство компьютеров имеют два режима работы:
 - Режим ядра – в этом режиме имеется полный доступ ко всему аппаратному обеспечению и может задействоваться любая инструкцию, которую машина в состоянии выполнить. Операционная система всегда работает в режиме ядра.
 - Режим пользователя – в этом режиме доступно лишь подмножество машинных инструкций. В частности, программам, работающим в режиме пользователя, запрещено использование инструкций, управляющих машиной или осуществляющих операции ввода-вывода (Input/Output – I/O).

Операционная система

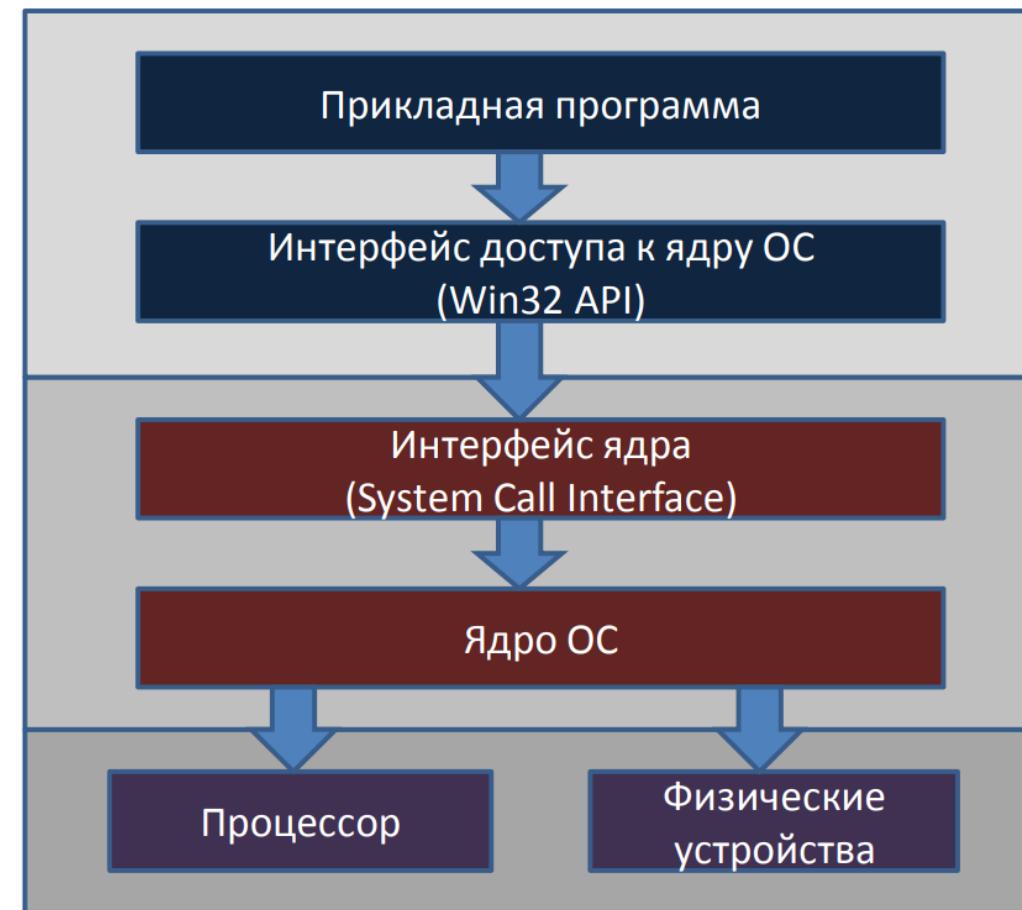
- ОС – это программное обеспечение (ПО), работающее в режиме ядра, и которое:
 - Предоставляет абстрактный набор ресурсов взамен неупорядоченного набора аппаратного обеспечения
 - Управляет этими ресурсами

Операционная система

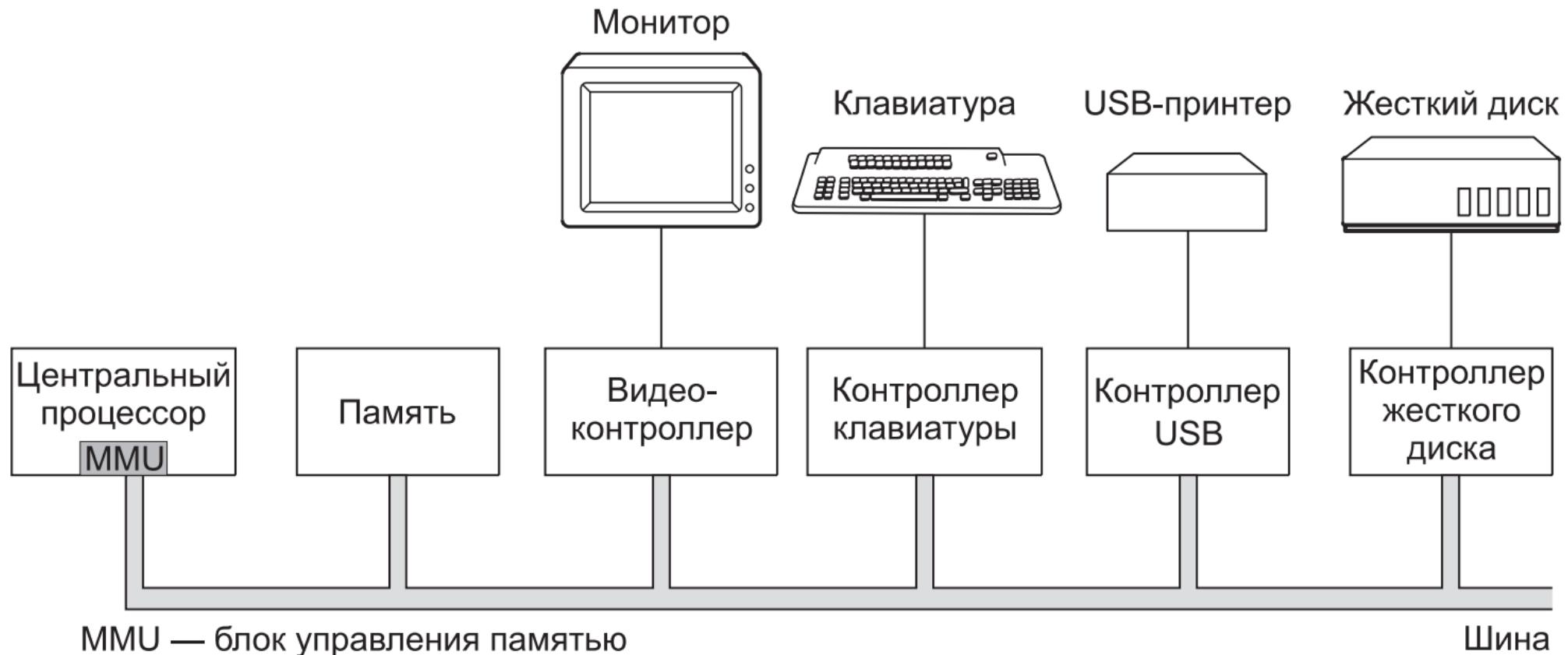
Пространство
пользователя

Пространство
ядра ОС

Аппаратное
обеспечение



Аппаратное обеспечение компьютера



Процессор

- Центральный процессор — это «мозг» компьютера. Его основная задача — выполнение инструкций
- По сути он занят бесконечным циклом:
 - Взять инструкцию из памяти
 - Выполнить эту инструкцию
- Процессоры бывают разных архитектур, например, x86_64, armv7, armv8, arm64
- Каждая архитектура отличается друг от друга, например, имеет свой набор инструкций

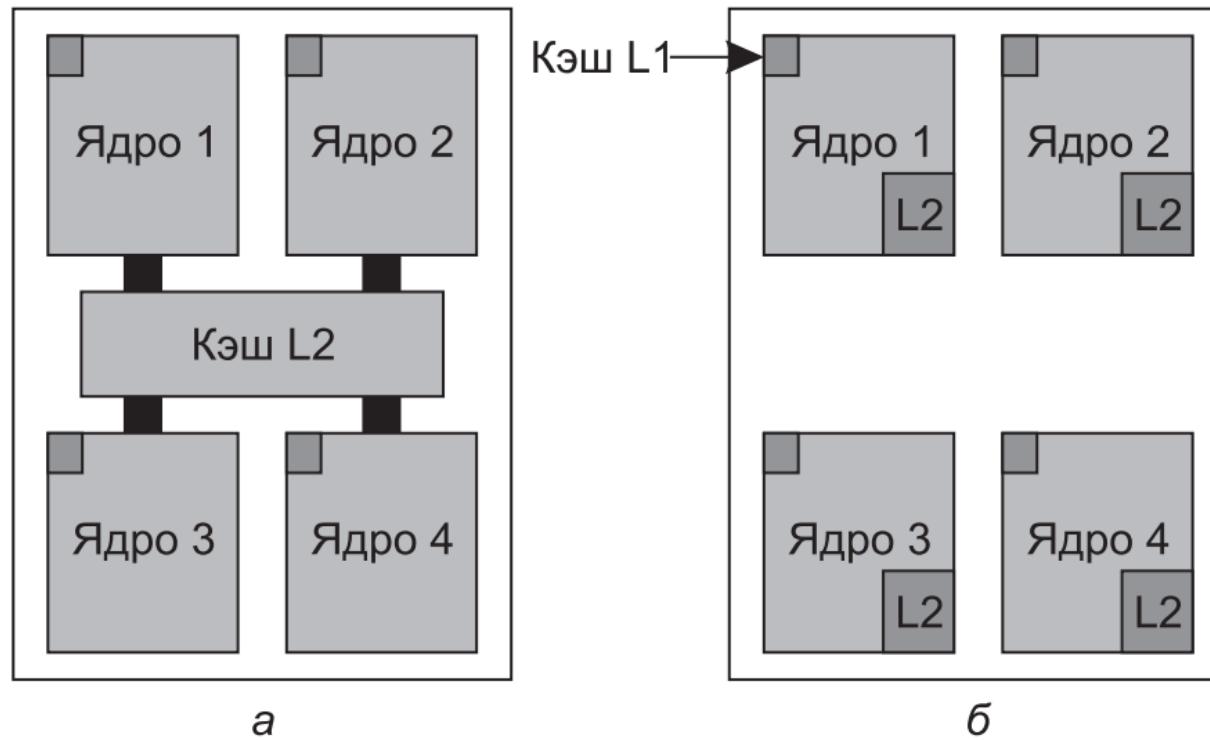
Многопоточные процессоры

- Многопоточность позволяет процессору сохранять состояние двух различных потоков и переключаться между ними за наносекунды.
- Например, если одному из процессов нужно прочитать слово из памяти (что занимает несколько тактов), многопоточный процессор может переключиться на другой поток.
- Многопоточность не предлагает настоящей параллельной обработки данных. Одновременно работает только один процесс, но время переключения между процессами сведено до наносекунд

Многоядерные процессоры

- Многоядерными являются процессоры, имеющие на одном кристалле четыре, восемь и более полноценных процессоров, или ядер.
- Например, четырехъядерные процессоры фактически имеют в своем составе четыре мини-чипа, каждый из которых представляет собой независимый процессор

Многоядерные процессоры



- Четырехъядерный процессор:
 - а* — с общей кэш-памятью второго уровня (L2);
 - б* — с отдельными блоками кэш-памяти L2

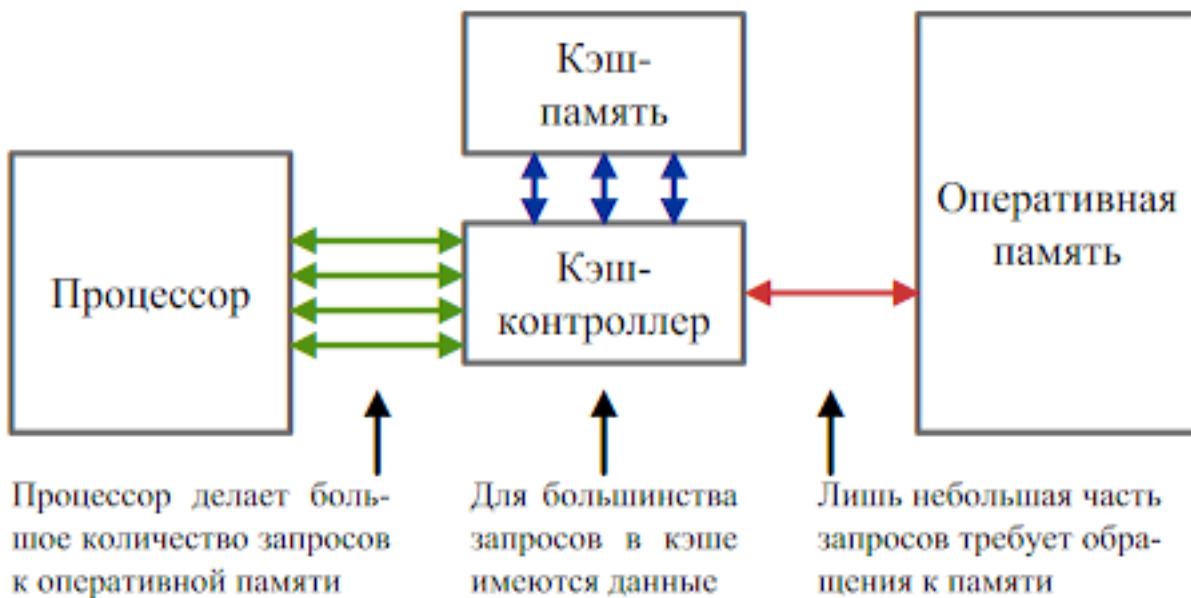
Память



Память: Регистры

- Регистры являются наиболее быстрой но и наиболее ограниченной памятью
- Внутренние регистры обычно предоставляют возможность для хранения 32×32 бит для 32-разрядного процессора или 64×64 бит — для 64-разрядного. В обоих случаях этот объем не превышает 1 Кбайт.

Память: Кэши



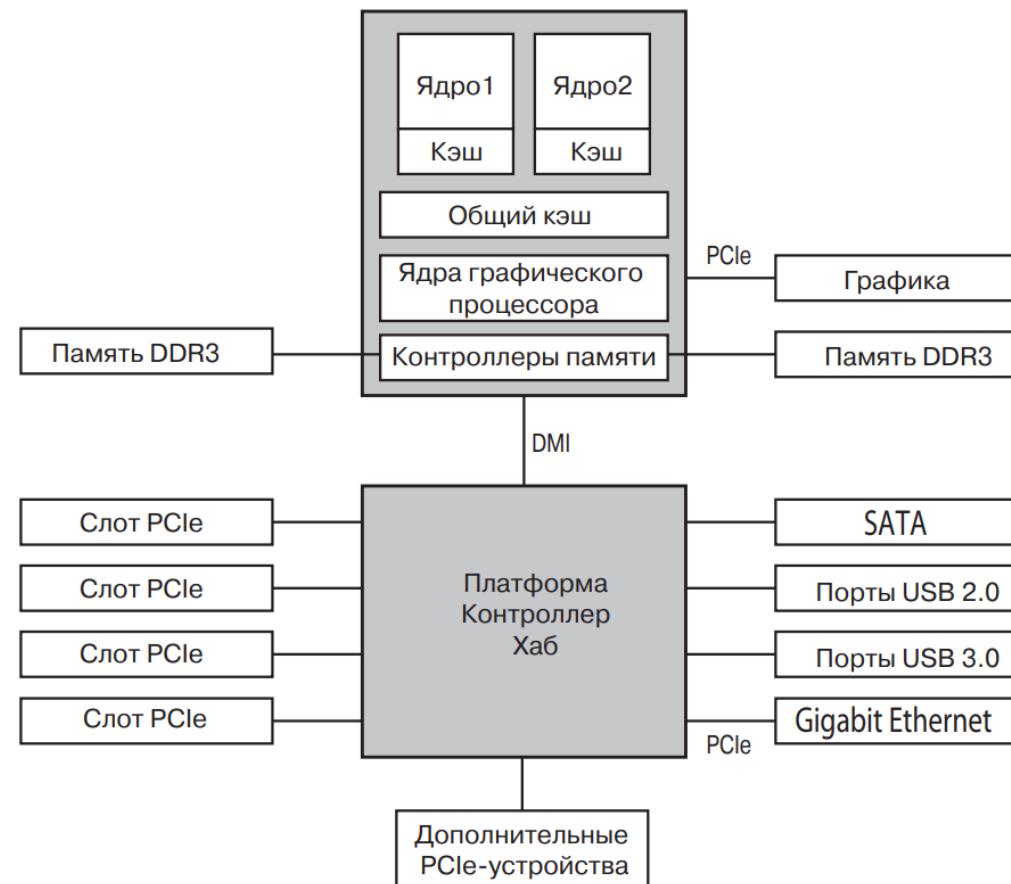
Память: Кэши

- Различают кэш память 1-, 2- и 3-го уровней (маркируются L1, L2 и L3).
 - Кэш память первого уровня (L1) – самый быстрый, но по объему меньший, чем у остальных. Например, в Intel Skylake он составляет 32Кб.
 - Кэш память второго уровня (L2) – объем этой памяти значительно больше, чем L1, но в то же время является более медленной памятью. В Intel Skylake он составляет 1Мб.
 - Кэш память третьего уровня (L3) – с большим объемом, но более медленная, чем L2. В Intel Skylake он составляет 27.5Мб.

Оперативная память

- Является быстрой памятью, но ограничена в размерах по сравнению с жестким диском.
- Для решения проблемы с малым размером существуют решения в виде сохранения части данных на диске
 - В Linux используются swap-файлы
 - В Windows - hiberfil.sys
- Является энергозависимой
 - В режиме гибернации (hibernation) питание на ОЗУ не подается. Перед переходом в этот режим, содержимое ОЗУ сохраняется на диск

Шина



Шина

- Используется для коммуникации между функциональными блоками компьютера
- Примерами являются:
 - PCI – шина ввода-вывода для подключения периферийных устройств к материнской плате компьютера
 - SATA – последовательный интерфейс обмена данными с накопителями информации
 - USB – последовательный интерфейс для подключения периферийных устройств к вычислительной технике.
 - SCSI – интерфейс для физического подключения и передачи данных между компьютерами и периферийными устройствами. Разработан для объединения на однойшине различных по своему назначению устройств, таких, как жёсткие диски, накопители на магнитооптических дисках, приводы CD, DVD, стримеры, сканеры, принтеры и т. д.

Загрузка компьютера

- У каждого персонального компьютера есть материнская плата.
- На материнской плате находится программа, которая называется базовой системой ввода-вывода — BIOS (Basic Input Output System).
- BIOS содержит низкоуровневое программное обеспечение ввода-вывода, включая процедуры считывания состояния клавиатуры, вывода информации на экран и осуществления, ко всему прочему, дискового ввода-вывода.

Загрузка компьютера

- При начальной загрузке компьютера BIOS начинает работать первой. Сначала она проверяет объем установленной на компьютере оперативной памяти и наличие клавиатуры, а также установку и нормальную реакцию других основных устройств.
- Все начинается со сканирования шин PCIe и PCI с целью определения всех подключенных к ним устройств. Эти устройства регистрируются.

Загрузка компьютера

- Затем BIOS определяет устройство, с которого будет вестись загрузка, по очереди проверив устройства из списка, сохраненного в CMOS-памяти. Пользователь может внести в этот список изменения, войдя сразу после начальной загрузки в программу конфигурации BIOS.
- После этого операционная система запрашивает BIOS, чтобы получить информацию о конфигурации компьютера. Она проверяет наличие драйвера для каждого устройства.

Загрузка компьютера

- Как только в ее распоряжении окажутся все драйверы устройств, операционная система загружает их в ядро.
- Затем она инициализирует свои таблицы, создает все необходимые ей фоновые процессы и запускает программу входа в систему или графический интерфейс пользователя.

Виды операционных систем

- Операционные системы мейнфреймов: OS/390, Linux
- Серверные операционные системы: FreeBSD, Linux, Windows Server
- Многопроцессорные операционные системы: Windows, Linux
- Операционные системы персональных компьютеров: Windows, Linux
- Операционные системы карманных персональных компьютеров: Android, iOS (по сути тоже Linux)
- Встроенные операционные системы: Embedded Linux, QNX и VxWorks
- Операционные системы сенсорных узлов: TinyOS
- Операционные системы реального времени
- Операционные системы смарт-карт

Процесс

- Процессом, по существу, является программа во время ее выполнения.
- С каждым процессом связано его **адресное пространство** – список адресов ячеек памяти от нуля до некоторого максимума, откуда процесс может считывать данные и куда может записывать их. Адресное пространство содержит выполняемую программу, данные этой программы и ее стек.

Процесс

- Кроме этого, с каждым процессом связан набор ресурсов, который обычно включает регистры (в том числе счетчик команд и указатель стека), список открытых файлов, необработанные предупреждения, список связанных процессов и всю остальную информацию, необходимую в процессе работы программы.
- Таким образом, процесс – это контейнер, в котором содержится вся информация, необходимая для работы программы.

Процесс

- Процесс способен создавать несколько других процессов, называющихся **дочерними процессами**
- Эти процессы в свою очередь могут создавать собственные дочерние процессы
- Связанные процессы, совместно работающие над выполнением какой-нибудь задачи, зачастую нуждаются в обмене данными друг с другом и синхронизации своих действий. Такая связь называется **межпроцессным взаимодействием**

Адресное пространство

- Представляет собой блок адресов в **виртуальной памяти**, доступных для программы
- Адресное пространство изолировано для каждой программы. По умолчанию невозможно из одной программы обратиться к памяти другой
- Виртуальная память в свою очередь является абстракцией над оперативной памятью и вторичным хранилищем (диском)
- Основной принцип работы виртуальной памяти заключается в том, что данные хранятся в ОЗУ, пока там есть место. Когда место заканчивается, часть данных переносится на вторичное хранилище

Ядро операционной системы

- Ядро операционной системы (Kernel) - часть операционной системы:
 - постоянно находящаяся в оперативной памяти,
 - управляющая всей операционной системой,
 - содержащая: драйверы устройств, подпрограммы управления памятью, планировщик заданий,
 - реализующая системные вызовы

Объект ядра в ОС Windows

- Объекты ядра создаются Windows-функциями ядра.
- Каждый объект ядра — на самом деле просто блок памяти, выделенный ядром и доступный только ему. Этот блок представляет собой структуру данных, в элементах которой содержится информация об объекте.
- Объекты ядра доступны только ядру, приложение не может самостоятельно найти эти структуры в памяти и напрямую модифицировать их содержимое.
- Такое ограничение Microsoft ввела намеренно, чтобы ни одна программа не нарушила целостность структур объектов ядра. Это же ограничение позволяет Microsoft вводить, убирать или изменять элементы структур, не нарушая работы каких-либо приложений.

Дескриптор (HANDLE) объекта ядра ОС Windows

- В ОС Windows каждому объекту операционной системы ставится в соответствие дескриптор (описатель, handle).
 - Дескриптор объекта представляет собой указатель на запись в таблице, которая поддерживается системой и содержит адрес объекта и средства для идентификации типа объекта.
- В Win32 API дескриптор имеет тип HANDLE. Определен как *void**.
- Приложение не имеет прямого доступа к объектам, а обращается к ним косвенно через дескрипторы.

Дескриптор vs идентификатор

- Дескриптор не имеет прямого отношения к другому свойству объектов ядра – идентификатору.
 - Дескриптор – это указатель (`void*`), идентификатор – это число (`DWORD`)
 - Идентификатор уникально идентифицирует объект, но не позволяет им управлять. Управлять объектом можно только с помощью дескриптора.
 - Например, человеком нельзя управлять используя его личный номер, указанный в паспорте (который по сути является идентификатором человека)

Учет пользователей объектов ядра ОС Windows

- Объекты ядра принадлежат ядру, а не процессу. Иначе говоря, если процесс вызывает функцию, создающую объект ядра, а затем завершается, объект ядра может быть не разрушен.
- В большинстве случаев такой объект все же разрушается; но если созданный объект ядра используется другим процессом, ядро запретит разрушение объекта до тех пор, пока от него не откажется и тот процесс.
- Ядру известно, сколько процессов использует конкретный объект ядра, поскольку в каждом объекте есть счетчик числа его пользователей.

Учет пользователей объектов ядра ОС Windows

- Этот счетчик – один из элементов данных, общих для всех типов объектов ядра.
- В момент создания объекта счетчику присваивается 1.
- Когда к существующему объекту ядра обращается другой процесс, счетчик увеличивается на 1.
- А когда какой-то процесс завершается, счетчики всех используемых им объектов ядра автоматически уменьшаются на 1.
- Как только счетчик какого-либо объекта обнуляется, ядро уничтожает этот объект.

Таблица дескрипторов ОС Windows

- При инициализации процесса система создает в нем таблицу дескрипторов, используемую только для объектов ядра.
- Сведения о структуре этой таблицы и управлении ею незадокументированы.

Таблица дескрипторов ОС Windows

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

Создание объекта ядра ОС Windows

- Когда процесс инициализируется в первый раз, таблица описателей еще пуста.
- При вызове функции, создающей объект ядра (например, `CreateFileMapping`), ядро
 - выделяет для этого объекта блок памяти и инициализирует его;
 - далее ядро просматривает таблицу описателей, принадлежащую данному процессу, и отыскивает свободную запись.
- Поскольку таблица еще пуста, ядро обнаруживает структуру с индексом 1 и инициализирует ее.
- Указатель устанавливается на внутренний адрес структуры данных объекта, маска доступа — на доступ без ограничений и, наконец, определяется последний компонент — флаги.

Закрытие объекта ядра ОС Windows

- Независимо от того, как был создан объект ядра, по окончании работы с ним его нужно закрыть вызовом **CloseHandle**
- Эта функция сначала проверяет таблицу описателей, принадлежащую вызывающему процессу, чтобы убедиться, идентифицирует ли переданный ей индекс объект, к которому этот процесс действительно имеет доступ.
- Если переданный индекс правilen, система получает адрес структуры данных объекта и уменьшает в этой структуре счетчик числа пользователей; как только счетчик обнулится, ядро удалит объект из памяти.

Закрытие объекта ядра ОС Windows

- Перед самым возвратом управления CloseHandle удаляет соответствующую запись из таблицы описателей:
 - данный описатель теперь недействителен в процессе и использовать его нельзя.
- При этом запись удаляется независимо от того, разрушен объект ядра или нет
- После вызова CloseHandle доступ к этому объекту ядра из данного процесса отсутствует; но, если его счетчик не обнулен, объект остается в памяти
- Когда и остальные процессы завершат свою работу с этим объектом (тоже вызвав CloseHandle), он будет разрушен

Процессы

Процессы

- Процесс — это экземпляр выполняемой программы, включая текущие значения счетчика команд, регистров и переменных.
- С каждым процессом связано его **адресное пространство** — список адресов ячеек памяти от нуля до некоторого максимума, откуда процесс может считывать данные и куда может записывать их.

Процессы

Task Manager

File Options View

Processes Performance App history Startup Users Details Services

Name	PID	Status	User name	CPU	CPU time	Memor...	I/O reads	Description
System Idle Process	0	Running	SYSTEM	93	334:30:49	4 K	0	Percentage of time the processor is idle
audiodg.exe	11824	Running	LOCAL SE...	02	1:39:09	7,692 K	0	Windows Audio Device Graph Isolation
chrome.exe	6592	Running	Tim	02	2:49:00	376,412 K	1,510,731	Google Chrome
dwm.exe	804	Running	DWM-1	01	2:05:41	285,828 K	175	Desktop Window Manager
Photoshop.exe	5116	Running	Tim	01	0:05:13	324,988 K	38,403	Adobe Photoshop CC 2015
Taskmgr.exe	7832	Running	Tim	00	0:02:54	28,688 K	1,830	Task Manager
svchost.exe	1316	Running	LOCAL SE...	00	0:05:25	13,780 K	772	Host Process for Windows Services
chrome.exe	5004	Running	Tim	00	0:05:25	81,700 K	24,307	Google Chrome
chrome.exe	7272	Running	Tim	00	1:12:24	170,884 K	9,686,964	Google Chrome
chrome.exe	4712	Running	Tim	00	0:45:46	235,532 K	19,963,479	Google Chrome
System	4	Running	SYSTEM	00	1:37:35	4 K	10,407	NT Kernel & System
chrome.exe	1504	Running	Tim	00	0:08:07	59,688 K	148,065	Google Chrome
explorer.exe	10768	Running	Tim	00	0:00:36	21,744 K	53,208	Windows Explorer
chrome.exe	4356	Running	Tim	00	0:10:01	205,032 K	2,337,673	Google Chrome
chrome.exe	11876	Running	Tim	00	0:03:36	243,400 K	38,710	Google Chrome
System interrupts	-	Running	SYSTEM	00	0:00:00	0 K	0	Deferred procedure calls and interrupt serv...
CEPHTMLEngine.exe	5224	Running	Tim	00	0:02:37	19,516 K	337,581	Adobe CEP HTML Engine
chrome.exe	9200	Running	Tim	00	0:01:41	57,992 K	82,263	Google Chrome
MsMpEng.exe	2992	Running	SYSTEM	00	2:14:22	109,300 K	6,666,059	Antimalware Service Executable
chrome.exe	6616	Running	Tim	00	0:02:06	167,048 K	101,496	Google Chrome
Adobe CEF Helper.exe	9320	Running	Tim	00	0:11:04	2,404 K	1,209,577	Adobe CEF Helper
svchost.exe	840	Running	SYSTEM	00	0:02:28	6,276 K	164	Host Process for Windows Services
OneDrive.exe	7420	Running	Tim	00	1:22:22	4,704 K	15,179	Microsoft OneDrive
CEPHTMLEngine.exe	11112	Running	Tim	00	0:00:36	1,692 K	1,390	Adobe CEP HTML Engine

Fewer details End task

Создание процессов

- Существуют четыре основных события, приводящих к созданию процессов:
 1. Инициализация системы: пользовательские и фоновые процессы
 2. Выполнение работающим процессом системного вызова, предназначенного для создания процесса: с помощью *fork* в UNIX или *CreateProcess* в Windows
 3. Запрос пользователя на создание нового процесса: при запуске пользователем приложения
 4. Инициация пакетного задания.

Завершение процессов

- Процессы могут быть завершены в случае:
 1. обычного выхода (добровольно): *exit* в UNIX и *ExitProcess* в Windows;
 2. выхода при возникновении ошибки (добровольно);
 3. возникновения фатальной ошибки (принудительно);
 4. уничтожения другим процессом (принудительно): *kill* в UNIX и *TerminateProcess* в Windows.

Иерархии процессов

- Процессы созданные другим процессов, называются **дочерними**.
- У процессов может быть несколько дочерних процессов.
- В то же время у процессов может быть лишь один родительский процесс.

Иерархии процессов в UNIX

- В UNIX все дочерние процессы и более отдаленные потомки образуют **группу процессов**.
- Когда пользователь отправляет сигнал с клавиатуры, тот достигает всех участников этой группы процессов, связанных на тот момент времени с клавиатурой (обычно это все действующие процессы, которые были созданы в текущем окне).
- Каждый процесс по отдельности может захватить сигнал, игнорировать его или совершить действие по умолчанию, которое должно быть уничтожено сигналом

Иерархии процессов в Windows

- В Windows не существует понятия иерархии процессов, и все процессы являются равнозначными.
- Единственным намеком на иерархию процессов можно считать присвоение родительскому процессу, создающему новый процесс, специального дескриптора, который может им использоваться для управления дочерним процессом.
- Но он может свободно передавать этот дескриптор другому процессу, нарушая тем самым иерархию. А в UNIX процессы не могут лишаться наследственной связи со своими дочерними процессами.

Состояния процесса

- Процесс может находиться в одном из трех состояний:
 - выполняемый (в данный момент использующий центральный процессор);
 - готовый (работоспособный, но временно приостановленный, чтобы дать возможность выполнения другому процессу);
 - заблокированный (неспособный выполнятся, пока не возникнет какое-нибудь внешнее событие)

Состояния процесса



1. Процесс заблокирован в ожидании ввода
2. Диспетчер выбрал другой процесс
3. Диспетчер выбрал данный процесс
4. Входные данные стали доступны

Реализация процессов

- Для реализации модели процессов операционная система ведет **таблицу процессов**, в которой каждая запись соответствует какому-нибудь процессу.
- Эти записи содержат важную информацию о состоянии процесса, включая счетчик команд, указатель стека, распределение памяти, состояние открытых им файлов, его учетную и планировочную информацию и все остальное, касающееся процесса, что должно быть сохранено, когда процесс переключается из состояния выполнения в состояние готовности или блокировки, чтобы позже он мог возобновить выполнение.

Реализация процессов

Таблица 2.1. Некоторые из полей типичной записи таблицы процессов

Управление процессом	Управление памятью	Управление файлами
Регистры	Указатель на информацию о текстовом сегменте	Корневой каталог
Счетчик команд	Указатель на информацию о сегменте данных	Рабочий каталог
Слово состояния программы	Указатель на информацию о сегменте стека	Дескрипторы файлов
Указатель стека		Идентификатор пользователя
Состояние процесса		Идентификатор группы
Приоритет		
Параметры планирования		
Идентификатор процесса		
Родительский процесс		
Группа процесса		
Сигналы		
Время запуска процесса		
Использованное время процессора		
Время процессора, использованное дочерними процессами		
Время следующего аварийного сигнала		

Процессы Windows

- В Windows под **процессом** понимается объект ядра, которому принадлежат системные ресурсы, используемые исполняемым приложением.
- С каждым процессом в Windows связаны:
 - уникальный дескриптор (HANDLE);
 - уникальный идентификатор.
- Дескрипторы используются программами пользователя для управления процессами.
- Идентификаторы используются служебными программами, которые позволяют пользователям системы отслеживать работу процессов.

Ресурсы процессов Windows

- Каждому процессу в Windows обязательно принадлежат следующие ресурсы:
 - виртуальное адресное пространство;
 - рабочее множество страниц в реальной памяти;
 - первичный поток;
 - маркер доступа, содержащий информацию для системы безопасности;
 - таблицу для хранения дескрипторов объектов ядра, которые используются процессом.

Создание процессов в Windows

- Процессы в Windows создаются с помощью функции Win32 API **CreateProcess**.
- [Создание процессов - Win32 apps | Microsoft Docs](#)

Завершение процессов в Windows

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

Завершение процессов: возврат управления входной функцией первичного потока

- Приложение следует проектировать так, чтобы его процесс завершался только после возврата управления входной функцией первичного потока.
- Это единственный способ, гарантирующий корректную очистку всех ресурсов, принадлежавших первичному потоку.
- При этом:
 - любые C++-объекты, созданные данным потоком, уничтожаются соответствующими деструкторами;
 - система освобождает память, которую занимал стек потока;
 - система устанавливает код завершения процесса (поддерживаемый объектом ядра «процесс») — его и возвращает Ваша входная функция;
 - счетчик пользователей данного объекта ядра «процесс» уменьшается на 1

Завершение процессов: функция ExitProcess

- Процесс завершается, когда один из его потоков вызывает функцию **ExitProcess**.
- Данная функция всегда вызывается неявно - когда входная функция возвращает управление, оно передается стартовому коду из библиотеки С/С++, и тот проводит очистку всех ресурсов, выделенных им процессу, а затем обращается к ExitProcess, передавая ей значение, возвращенное входной функцией

Завершение процессов: функция ExitProcess

- Но если вызывать ее явно, то можно получить некорректное освобождение памяти.
- Например, при запуске кода ниже, деструкторы не будут вызваны.

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

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

CSomeObj g_GlobalObj;

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

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

Завершение процессов: функция TerminateProcess

- Главное отличие этой функции от ExitProcess в том, что ее может вызвать любой поток и завершить любой процесс.
- TerminateProcess стоит использовать лишь в том случае, когда иным способом завершить процесс не удается:
 - процесс не получает абсолютно никаких уведомлений о том, что он завершается, и приложение не может ни выполнить очистку, ни предотвратить свое неожиданное завершение
 - при этом теряются все данные, которые процесс не успел переписать из памяти на диск.

Завершение процессов: функция TerminateProcess

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

Завершение процессов: не осталось активных потоков

- Если все потоки процесса завершили работу, операционная система больше не считает нужным «содержать» адресное пространство данного процесса.
- Обнаружив, что в процессе не исполняется ни один поток, она немедленно завершает его.
- При этом код завершения процесса приравнивается коду завершения последнего потока.

Завершение процессов

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

Наследование дескрипторов объектов в Windows

- Если дочерний процесс имеет доступ к объекту, созданному в родительском процессе, то этот объект называется **наследуемым**.
- Для того чтобы сделать объект наследуемым, нужно в родительском процессе установить свойство наследования в дескрипторе этого объекта.
- Причем это свойство должно быть установлено до создания дочернего процесса, который должен получить доступ к наследуемому объекту

Наследование дескрипторов объектов в Windows

- В Windows свойство наследования объекта устанавливается двумя способами:
 - функцией **Create** при создании объекта;
 - функцией **SetHandleInformation**.
- Для того чтобы узнать, является ли дескриптор наследуемым, нужно использовать функцию **GetHandleInformation**.

Наследование дескрипторов объектов в Windows

- Наследование позволяет дочерним процессам иметь доступ к дескрипторам родительского процесса.
- Альтернативой наследованию может быть использование именованных объектов и передачи имени с помощью средств межпроцессной коммуникации: каналы, почтовые ящики и др.

Потоки

Поток

- Поток (thread) определяет набор инструкций, который будет исполняться процессором во время работы программы.
- У каждого процесса в операционной системе есть
 - Адресное пространство.
 - Как минимум один поток выполнения (поток)

Зачем нужны потоки

- Во многих приложениях одновременно происходит несколько действий, часть которых может периодически быть заблокированной.
 - Модель программирования упрощается за счет разделения такого приложения на несколько последовательных потоков, выполняемых в квазипараллельном режиме.
- «Легкость» (то есть быстрота) их создания и ликвидации по сравнению с более «тяжеловесными» процессами.
 - Во многих системах создание потоков осуществляется в 10 – 100 раз быстрее, чем создание процессов.

Зачем нужны потоки

- Производительность:
 - Когда потоки работают в рамках одного центрального процессора, они не приносят никакого прироста производительности
 - Но когда выполняются значительные вычисления, а также значительная часть времени тратится на ожидание ввода-вывода, наличие потоков позволяет этим действиям перекрываться по времени, ускоряя работу приложения.
- Потоки весьма полезны для систем, имеющих несколько центральных процессоров, где есть реальная возможность параллельных вычислений.

Зачем нужны потоки – пример

- Рассмотрим в качестве первого примера текстовый процессор.
- Обычно эти программы отображают создаваемый документ на экране в том виде, в каком он будет выводиться на печать.
- Предположим, что пользователь пишет какую-то книгу.
- Вся книга хранится в едином файле и имеет порядка 1000 страниц.

Зачем нужны потоки – пример

- Проверив внесенные изменения, он хочет внести еще одну поправку на 600-й странице и набирает команду, предписывающую текстовому процессору перейти на эту страницу.
- Теперь текстовый процессор вынужден немедленно переформатировать всю книгу вплоть до 600-й страницы, поскольку он не знает, какой будет первая строка на 600-й странице, пока не обработает всех предыдущие страницы.
- Перед отображением 600-й страницы может произойти существенная задержка, вызывающая недовольство пользователя.

Зачем нужны потоки – пример

- Теперь предположим, что текстовый процессор написан как двухпоточная программа.
- Один из потоков взаимодействует с пользователем, а другой занимается переформатированием в фоновом режиме.
- Как только предложение с первой страницы будет удалено, поток, отвечающий за взаимодействие с пользователем, приказывает потоку, отвечающему за формат, переформатировать всю книгу.
- Пока взаимодействующий поток продолжает отслеживать события клавиатуры и мыши, реагируя на простые команды вроде прокрутки первой страницы, второй поток с большой скоростью выполняет вычисления.
- Если немного повезет, то переформатирование закончится как раз перед тем, как пользователь запросит просмотр 600-й страницы, которая тут же сможет быть отображена.

Зачем нужны потоки – пример

- Многие текстовые процессоры обладают свойством автоматического сохранения всего файла на диск каждые несколько минут, чтобы уберечь пользователя от утраты его дневной работы в случае программных или системных сбоев или отключения электропитания.
- Для этого можно добавить третий поток, который будет заниматься созданием резервных копий на диске, не мешая первым двум.

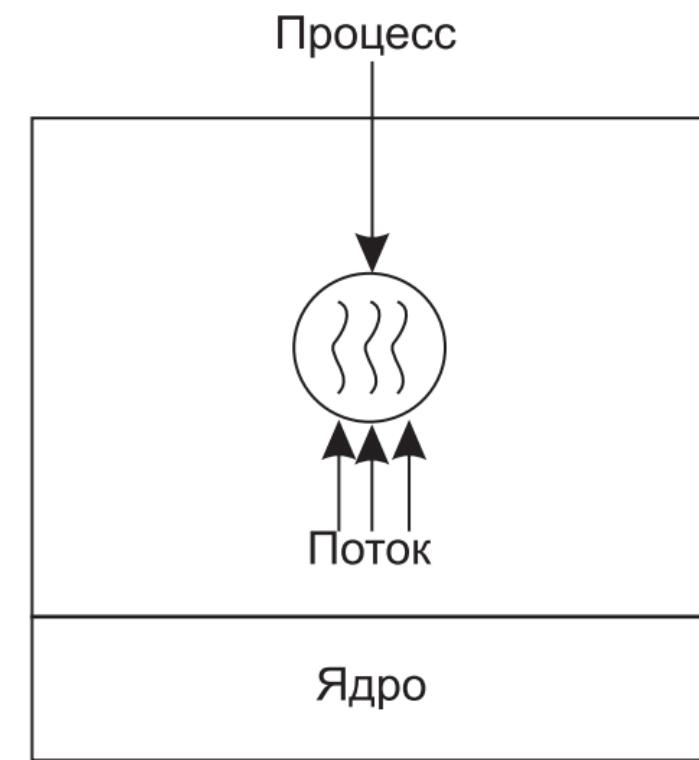
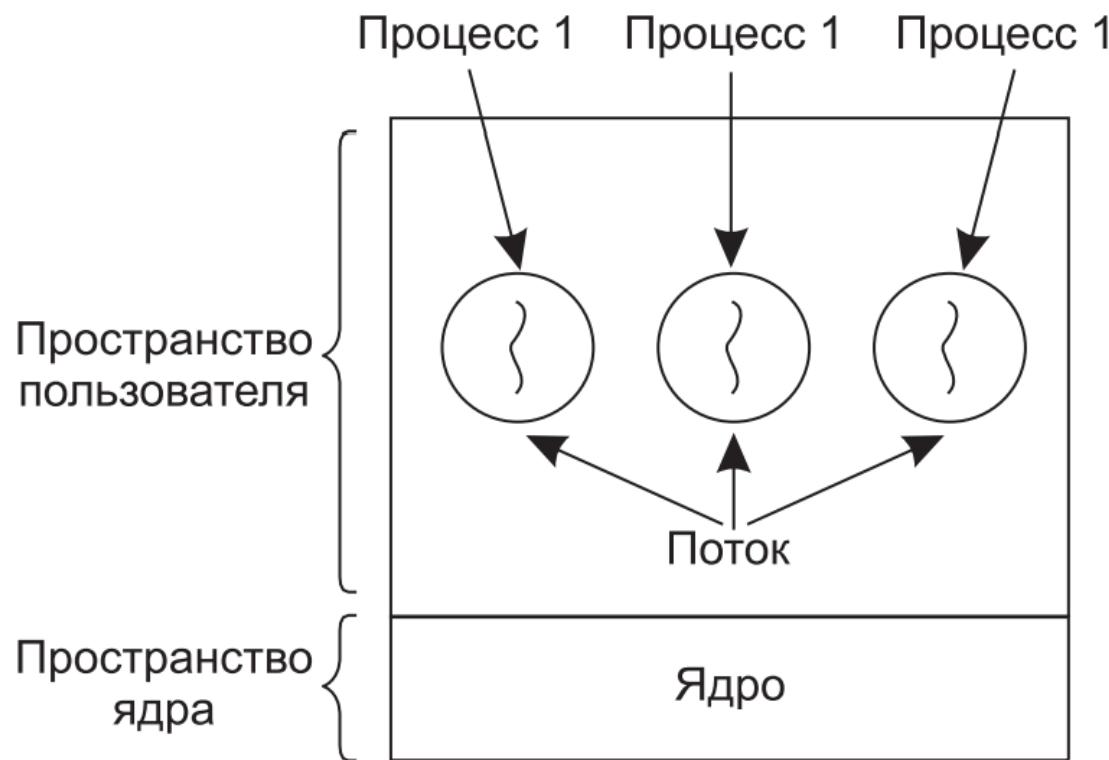
Зачем нужны потоки – пример

- Если бы программа была рассчитана на работу только одного потока, то с начала создания резервной копии на диске и до его завершения игнорировались бы команды и потоки с клавиатуры или мыши.
- Пользователь ощущал бы это как слабую производительность.
- Можно было бы сделать так, чтобы события клавиатуры или мыши прерывали создание резервной копии на диске, позволяя достичь более высокой производительности, но это привело бы к сложной модели программирования, основанной на применении прерываний.

Зачем нужны потоки – пример

- Три отдельных процесса так работать не будут, поскольку с документом необходимо работать всем трем потокам.
- Три потока вместо трех процессов используют общую память, таким образом, все они имеют доступ к редактируемому документу.
- При использовании трех процессов такое было бы сложно достичимо.

Процесс и поток



Процесс и поток

- Если потоки независимы друг от друга, то такие потоки могут располагаться в обособленных процессах.
- Если потоки являются частью одного и того же, например, используют общие данные и активно взаимодействуют друг с другом, то такие потоки могут находиться в одном и том же процессе.

Процесс VS ПОТОК

- Процесс инертен – он не имеет никакого отношения к выполнению инструкций процессором.
- Процесс служит для группировки ресурсов.
- У каждого процесса есть свое адресное пространство.
- Поток в свою очередь определяет инструкции, выполняемые процессором
- Потоки пользуются памятью процесса, в то же время они имеют свои регистры для сохранения рабочих переменных.
- Несколько потоков в одном процессе могут использовать общую память и общие переменные.

Процесс vs поток

Элементы, присущие каждому процессу	Элементы, присущие каждому потоку
Адресное пространство	Счетчик команд
Глобальные переменные	Регистры
Открытые файлы	Стек
Дочерние процессы	Состояние
Необработанные аварийные сигналы	
Сигналы и обработчики сигналов	
Учетная информация	

Счетчик команд потока

- У каждого потока есть счетчик команд, отслеживающий, какую очередную инструкцию нужно выполнять.

Регистры потока

- Каждый поток имеет свой набор регистров, в котором хранятся текущие рабочие переменные.

Состояние потока

- Подобно традиционному процессу (то есть процессу только с одним потоком), поток должен быть в одном из следующих состояний:
 - Выполняемый – занимает центральный процессор и является активным в данный момент.
 - Заблокированный – ожидает события, которое его разблокирует.
 - Готовый или завершенный - планируется к выполнению и будет выполнен, как только подойдет его очередь.
- Переходы между состояниями потока происходят аналогично процессам.

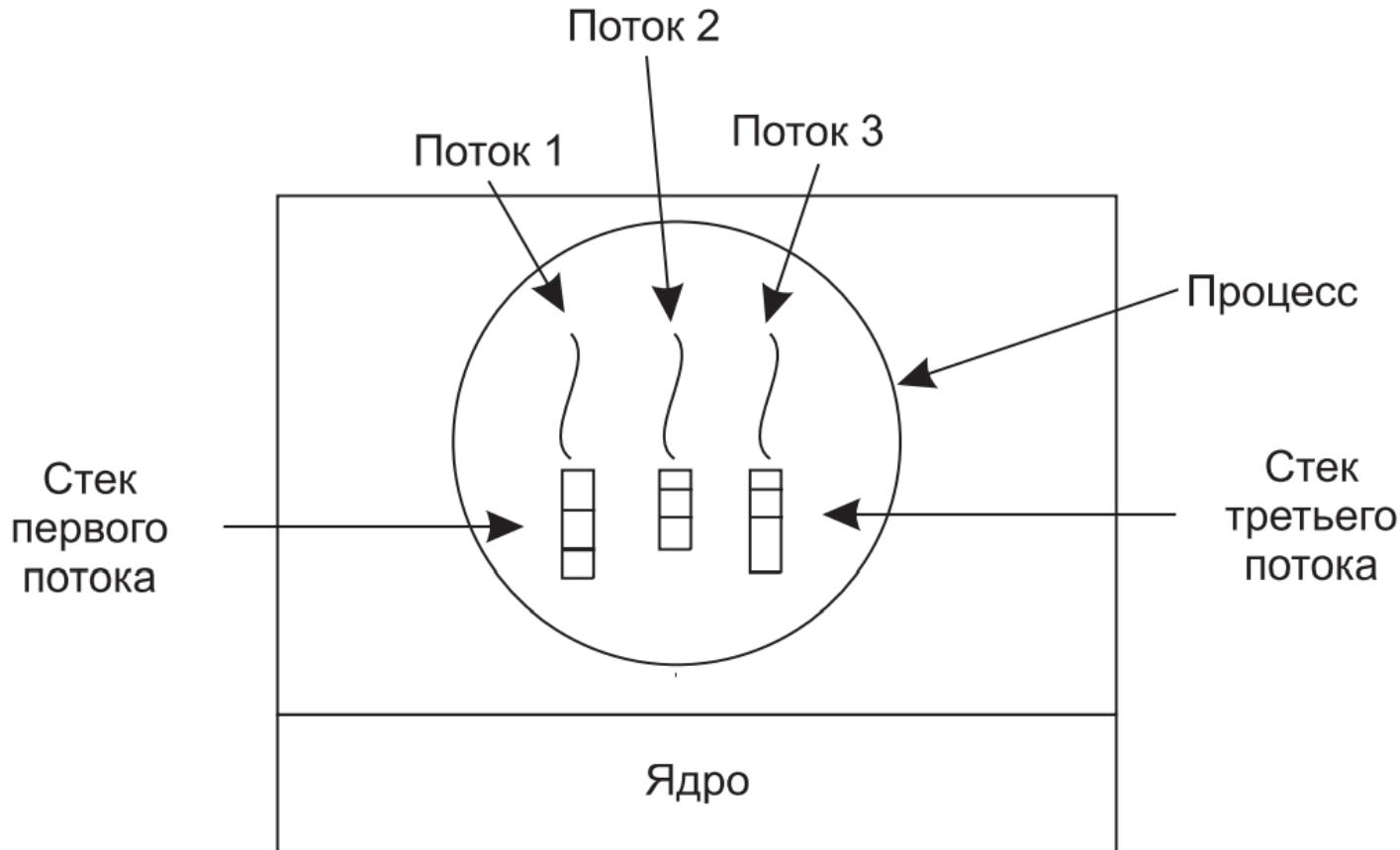
Состояние потока



Стек потока

- Каждый поток имеет собственный стек.
- Стек каждого потока содержит по одному фрейму для каждой уже вызванной, но еще не возвратившей управление процедуры.
- Такой фрейм содержит локальные переменные процедуры и адрес возврата управления по завершении ее вызова.
- Например, если процедура X вызывает процедуру Y, а Y вызывает процедуру Z, то при выполнении Z в стеке будут фреймы для X, Y и Z.

Стек потока



Основные функции работы с потоками - создание

- Когда используется многопоточность, процесс обычно начинается с использования одного потока.
- Этот поток может создавать новые потоки, вызвав библиотечную процедуру, к примеру **thread_create**.
- В параметре **thread_create** обычно указывается имя процедуры, запускаемой в новом потоке.
- Нет необходимости (или даже возможности) указывать для нового потока какое-нибудь адресное пространство, поскольку он автоматически запускается в адресном пространстве создающего потока.

Основные функции работы с потоками - завершение

- Когда поток завершает свою работу, выход из него может быть осуществлен за счет вызова библиотечной процедуры, к примеру **thread_exit**.
- После этого он прекращает свое существование и больше не фигурирует в работе планировщика.

Основные функции работы с потоками - завершение

- В некоторых использующих потоки системах какой-нибудь поток для выполнения выхода может ожидать выхода из какого-нибудь другого (указанного) потока после вызова процедуры, к примеру **thread_join**.
- Эта процедура блокирует вызывающий поток до тех пор, пока не будет осуществлен выход из другого (указанного) потока.

Основные функции работы с потоками – освобождение процессора

- Другой распространенной процедурой, вызываемой потоком, является **thread_yield**.
- Она позволяет потоку добровольно уступить центральный процессор для выполнения другого потока.
- Важность вызова такой процедуры обусловливается отсутствием прерывания по таймеру, которое есть у процессов и благодаря которому фактически задается режим многозадачности.

Потоки в Windows

- **Потоком в Windows** называется объект ядра, которому операционная система выделяет процессорное время для выполнения приложения.
- С каждым потоком в Windows связаны:
 - уникальный дескриптор (HANDLE);
 - уникальный идентификатор.
- Дескрипторы используются программами пользователя для управления потоками.
- Идентификаторы используются служебными программами, которые позволяют пользователям системы отслеживать работу потоков.

Контекст потока в Windows

- Каждому потоку в Windows принадлежат следующие ресурсы:
 - набор регистров микропроцессора;
 - стек для работы потока;
 - стек для работы ядра операционной системы;
 - маркер доступа, который содержит информацию для системы безопасности.
- Все эти ресурсы образуют **контекст потока в Windows**.

Типы потоков в Windows

- В Windows различаются потоки двух типов:
 - системные потоки;
 - пользовательские потоки.
- Системные потоки выполняют различные сервисы операционной системы и запускаются ядром операционной системы.
- Пользовательские потоки служат для решения задач пользователя и запускаются приложением.

Типы потоков в Windows

- В приложении различаются потоки двух типов:
 - рабочие потоки (working threads);
 - потоки интерфейса пользователя (user interface threads).
- Рабочие потоки выполняют различные задачи в приложении – функция main.
- Потоки интерфейса пользователя выполняют обработку сообщений к окнам, с которыми они связаны – функция WinMain.

ФУНКЦИИ УПРАВЛЕНИЯ ПОТОКАМИ В Windows

- **CreateThread** – один поток создает другой поток;
- **ExitThread** – поток завершает свою работу;
- **GetExitCodeThread** – получить код завершения потока;
- **TerminateThread** – один поток завершает работу другого потока;
- **SuspendThread** – один поток приостанавливает исполнение другого потока;
- **ResumeThread** – один поток возобновляет исполнение другого потока;
- **Sleep** – поток приостанавливает свое исполнение на заданный интервал времени

POSIX потоки

- **POSIX**-потоки: первая попытка стандартизировать АПИ операционных систем
 - POSIX == Portable Operating System Interface
 - Поддерживается по умолчанию системами UNIX
 - Также есть возможность использовать в Windows.
- Основные функции:
 - `pthread_create` – создание потока
 - `pthread_join` – ожидание потока
 - `pthread_exit` – завершение потока

POSIX потоки

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

void *print_message_function( void *ptr );

main()
{
    pthread_t thread1, thread2;
    char *message1 = "Thread 1";
    char *message2 = "Thread 2";
    int iret1, iret2;

    /* Create independent threads each of which will execute function */

    iret1 = pthread_create( &thread1, NULL, print_message_function, (void*) message1);
    iret2 = pthread_create( &thread2, NULL, print_message_function, (void*) message2);

    /* Wait till threads are complete before main continues. Unless we */
    /* wait we run the risk of executing an exit which will terminate */
    /* the process and all threads before the threads have completed. */

    pthread_join( thread1, NULL);
    pthread_join( thread2, NULL);

    printf("Thread 1 returns: %d\n",iret1);
    printf("Thread 2 returns: %d\n",iret2);
    exit(0);
}

void *print_message_function( void *ptr )
{
    char *message;
    message = (char *) ptr;
    printf("%s \n", message);
}
```

C++ 11 ПОТОКИ

- В стандарте C++ 11 были введены интерфейсы для работы с потоками.
- Данный стандарт реализуется всеми популярными компиляторами, включая gcc, g++ и msvc (компилятор C/C++ для Windows – Microsoft Visual C Compiler).
- В Windows данный интерфейс реализован над Win32 API, т.е. внутри используются те же функции CreateThread(...)_beginthreadex(...)
- Основные функции:
 - Создание происходит с помощью **конструктора thread(...)**
 - Ожидание выполняется с помощью метода класса thread **join()**
 - Освобождение выполняется с помощью деструктора

Асинхронность в C++

- Также в современных стандартах C++ присутствуют API для асинхронной работы.
- Например, функция `std::async` возвращает объект `future`, с помощью которого можно получить результат работы (используя метод `get`).
- При вызове функции `async` создается поток, который будет выполнять функцию, переданную как аргумент `async`.
- При вызове метода `get` объекта `future`, происходит ожидание завершения работы потока и возвращается его результат.

Планирование процессов и потоков

Планирование процессов и потоков

- Когда компьютер работает в многозадачном режиме, на нем зачастую запускается сразу несколько процессов или потоков, претендующих на использование центрального процессора.
- Такая ситуация складывается в том случае, если в состоянии готовности одновременно находятся два или более процесса или потока.
- Если доступен только один центральный процессор, необходимо выбрать, какой из этих процессов будет выполняться следующим.
- Та часть операционной системы, на которую возложен этот выбор, называется **планировщиком**, а алгоритм, который ею используется, называется **алгоритмом планирования**.

Поведение процессов

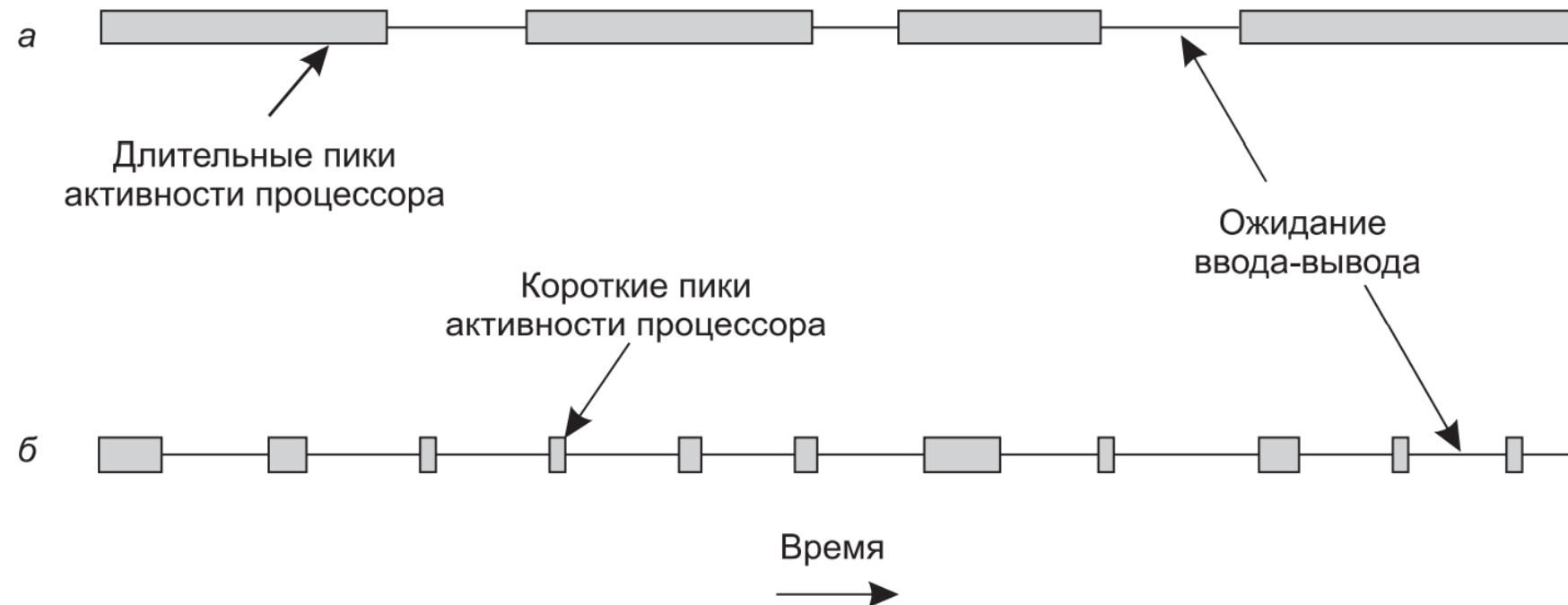


Рис. 2.20. Пики активного использования центрального процессора чередуются с периодами ожидания завершения операций ввода-вывода: а — процесс, ограниченный скоростью вычислений; б — процесс, ограниченный скоростью работы устройств ввода-вывода

Поведение процессов

- Некоторые процессы проводят основную часть своего времени за вычислениями. Такие процессы называются **процессами, ограниченными скоростью вычислений**.
- Другие основную часть своего времени ожидают завершения операций ввода-вывода. Такие процессы называются **процессами, ограниченными скоростью работы устройств ввода-вывода**.
- Чем быстрее становятся центральные процессоры, тем больше процессы ограничиваются скоростью работы устройства ввода-вывода

Когда необходимо планирование процессов

- При создании нового процесса необходимо принять решение, какой из процессов выполнять, родительский или дочерний.
 - Поскольку оба процесса находятся в состоянии готовности, вполне естественно, что планировщик должен принять решение, то есть вполне обоснованно выбрать выполнение либо родительского, либо дочернего процесса.
- Когда процесс завершает работу.
 - Процесс больше не может выполняться (поскольку он уже не существует), поэтому нужно выбрать какой-нибудь процесс из числа готовых к выполнению.
 - Если готовые к выполнению процессы отсутствуют, обычно запускается предоставляемый системой холостой процесс.

Когда необходимо планирование процессов

- Когда процесс блокируется в ожидании завершения операции ввода-вывода, на семафоре или по какой-нибудь другой причине, для выполнения должен быть выбран какой-то другой процесс.
 - Иногда в этом выборе играет роль причина блокирования.
 - Например, если процесс А играет важную роль и ожидает, пока процесс Б не выйдет из критической области, то предоставление очередности выполнения процессу Б позволит этому процессу выйти из его критической области, что даст возможность продолжить работу процессу А.
 - Но сложность в том, что планировщик обычно не обладает необходимой информацией, позволяющей учесть данную зависимость.

Когда необходимо планирование процессов

- При возникновении прерывания ввода-вывода.
 - Если прерывание пришло от устройства ввода-вывода, завершившего свою работу, то какой-то процесс, который был заблокирован в ожидании завершения операции ввода-вывода, теперь может быть готов к возобновлению работы.
 - Планировщик должен решить, какой процесс ему запускать: тот, что только что перешел в состояние готовности, тот, который был запущен за время прерывания, или какой-нибудь третий процесс.

Алгоритмы планирования по реакции на прерывания по таймеру

- **Неприоритетный** алгоритм планирования выбирает запускаемый процесс, а затем просто дает ему возможность выполняться до тех пор, пока он не заблокируется (в ожидании либо завершения операции ввода-вывода, либо другого процесса), или до тех пор, пока он добровольно не освободит центральный процессор.
- **Приоритетный** алгоритм планирования предусматривает выбор процесса и предоставление ему возможности работать до истечения некоторого строго определенного периода времени. Если до окончания этого периода он все еще будет работать, планировщик приостанавливает его работу и выбирает для запуска другой процесс (если есть доступный для этого процесс).

Алгоритмы планирования в зависимости от среды выполнения

- В **пакетных системах** не бывает пользователей, терпеливо ожидающих за своими терминалами быстрого ответа на свой короткий запрос. Поэтому для них зачастую приемлемы неприоритетные алгоритмы или приоритетные алгоритмы с длительными периодами для каждого процесса.
- В среде с пользователями, работающими в **интерактивном режиме**, приобретает важность приоритетность, удерживающая отдельный процесс от захвата центрального процессора, лишающего при этом доступа к службе всех других процессов.
 - Даже при отсутствии процессов, склонных к бесконечной работе, один из процессов в случае программной ошибки мог бы навсегда закрыть доступ к работе всем остальным процессам.
- В системах, ограниченных условиями **реального времени**, как ни странно, приоритетность иногда не требуется, поскольку процессы знают, что они могут запускаться только на непродолжительные периоды времени, и зачастую выполняют свою работу довольно быстро, а затем блокируются.
 - В отличие от интерактивных систем в системах реального времени запускаются лишь те программы, которые предназначены для содействия определенной прикладной задаче.

Задачи алгоритма планирования во всех системах

- равнодоступность – предоставление каждому процессу справедливой доли времени центрального процессора;
- принуждение к определенной политике – наблюдение за выполнением установленной политики;
- баланс – поддержка загруженности всех составных частей системы

Задачи алгоритма планирования в пакетных системах

- производительность – выполнение максимального количества задач в час;
- оборотное время — минимизация времени между представлением задачи и ее завершением;
- использование центрального процессора – поддержка постоянной загруженности процессора.

Задачи алгоритма планирования в интерактивных системах

- время отклика – быстрый ответ на запросы;
- пропорциональность – оправдание пользовательских надежд
 - Пользователям свойственно прикидывать продолжительность тех или иных событий.
 - Когда запрос, рассматриваемый как сложный, занимает довольно продолжительное время, пользователь воспринимает это как должное, но когда запрос, считающийся простым, также занимает немало времени, пользователь выражает недовольство.

Задачи алгоритма планирования в системах реального времени

- соблюдение предельных сроков — предотвращение потери данных;
- предсказуемость — предотвращение ухудшения качества в мультимедийных системах

Циклическое планирование

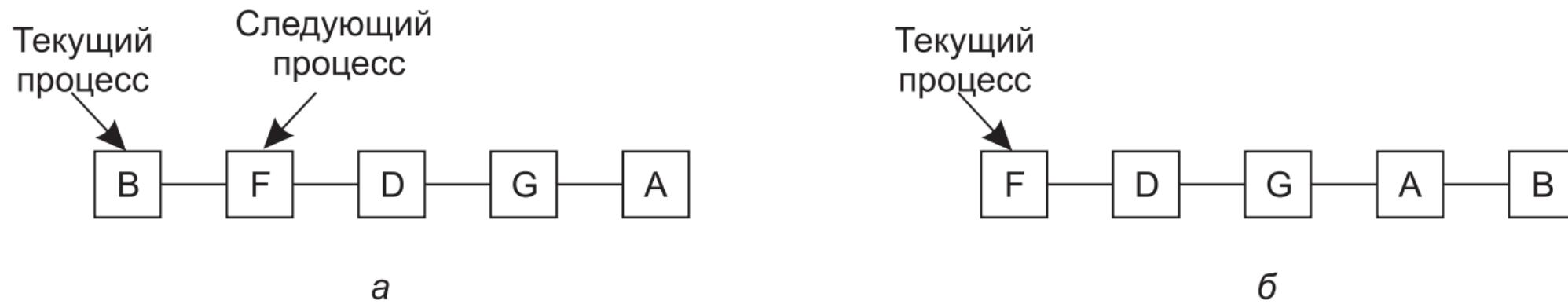


Рис. 2.22. Циклическое планирование: *а* — список процессов, находящихся в состоянии готовности; *б* — тот же список после того, как процесс *B* исчерпал свой квант времени

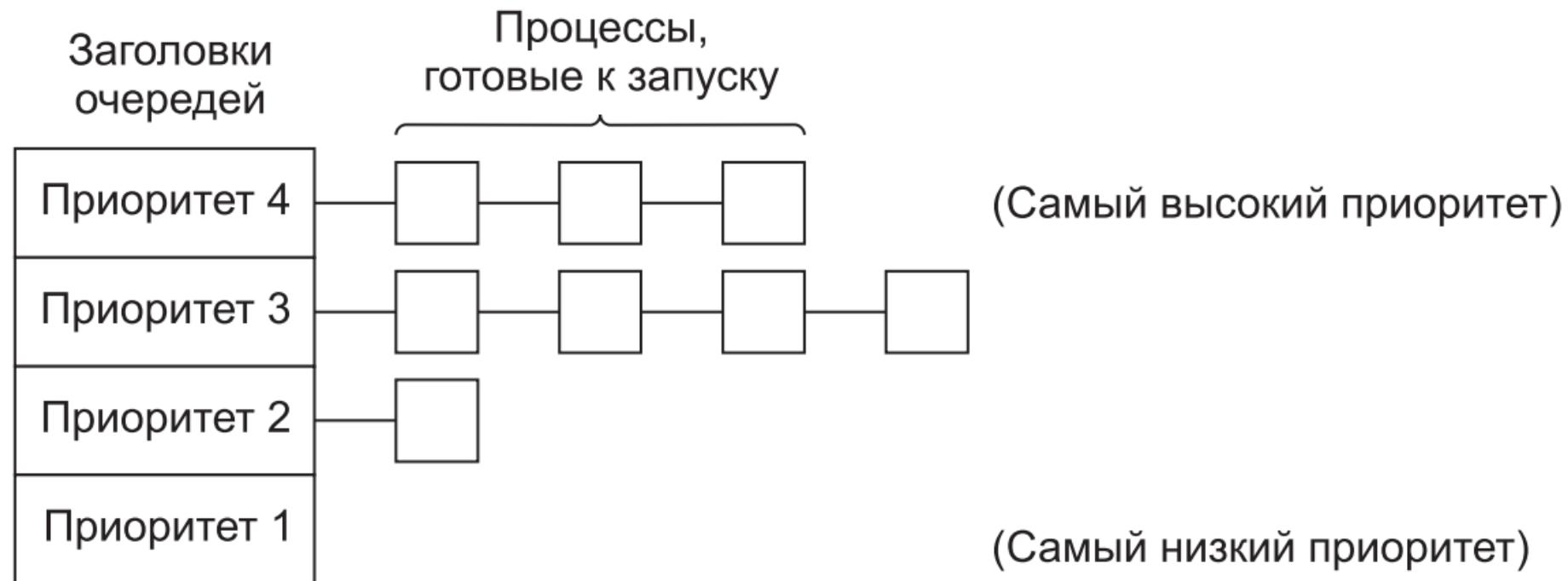
Циклическое планирование

- Каждому процессу назначается определенный интервал времени, называемый его квантом, в течение которого ему предоставляется возможность выполнения.
- Если процесс к завершению кванта времени все еще выполняется, то ресурс центрального процессора у него отбирается и передается другому процессу.
- Если процесс переходит в заблокированное состояние или завершает свою работу до истечения кванта времени, то переключение центрального процессора на другой процесс происходит именно в этот момент.
- Когда процесс исчерпает свой квант времени, он помещается в конец списка.

Циклическое планирование

- Установка слишком короткого кванта времени приводит к слишком частым переключениям процессов и снижает эффективность использования центрального процессора.
- Установка слишком длинного кванта времени может привести к слишком вялой реакции на короткие интерактивные запросы.
- Зачастую разумным компромиссом считается квант времени в 20–50 мс.

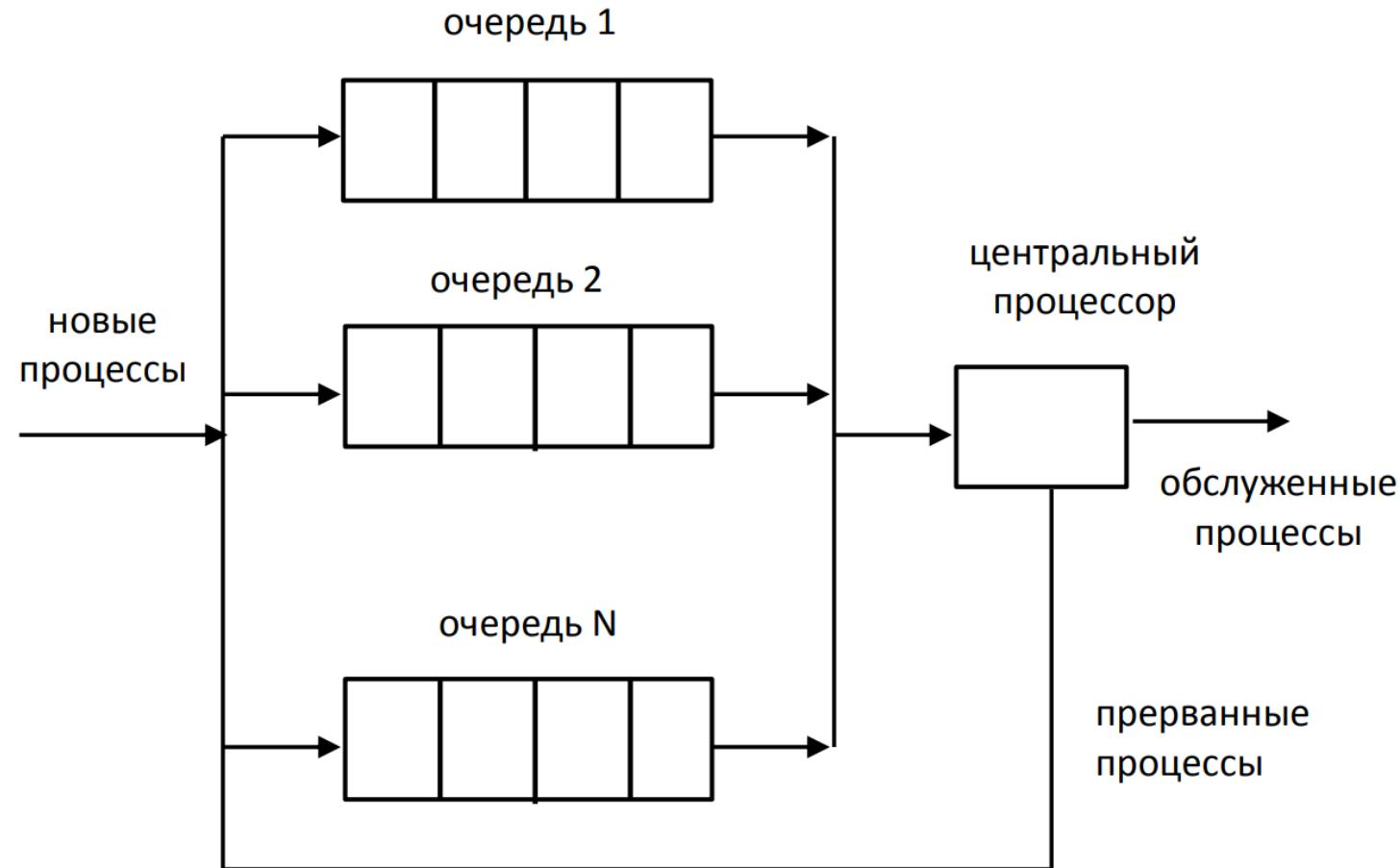
Приоритетное планирование



Приоритетное планирование

- Каждому процессу присваивается значение приоритетности и запускается тот процесс, который находится в состоянии готовности и имеет наивысший приоритет.
- Чтобы предотвратить бесконечное выполнение высокоприоритетных процессов, планировщик должен понижать уровень приоритета текущего выполняемого процесса с каждым его прерыванием.
- Если это действие приведет к тому, что его приоритет упадет ниже приоритета следующего по этому показателю процесса, произойдет переключение процессов.
- Или же каждому процессу может быть выделен максимальный квант допустимого времени выполнения. Когда квант времени будет исчерпан, шанс запуска будет предоставлен другому процессу, имеющему наивысший приоритет.

Использование нескольких очередей



Использование нескольких очередей

- Также при использовании приоритетов могут использоваться несколько очередей.
- В этом случае каждая очередь включает процессы, которые имеют одинаковый приоритет.
- Первыми обслуживаются процессы, которые имеют наивысший приоритет.

Выбор следующим самого короткого процесса

- В данном алгоритме следующим выполняется процесс, имеющий наименьшее время выполнения.
- Для оценки времени выполнения можно использовать историю предыдущих выполнений этого же процесса.

Выбор следующим самого короткого процесса

- Предположим, что оценка времени выполнения одной команды составляет T_0 . Предположим, что следующая оценка этого времени составляет T_1 .
- Мы можем обновить наш расчет, взяв взвешенную сумму этих двух чисел, то есть $aT_0 + (1 - a)T_1$.
 - Выбирая значение a , можно решить, стоит ли при оценке процесса быстро забывать его предыдущие запуски или нужно запоминать их надолго.
- При $a = \frac{1}{2}$ мы получаем следующую последовательность вычислений:
 $T_0; \frac{T_0}{2} + \frac{T_1^2}{2}; \frac{T_0}{4} + \frac{T_1}{4} + \frac{T_2}{2}; \frac{T_0}{8} + \frac{T_1}{8} + \frac{T_2}{4} + \frac{T_3}{2}.$
- После трех новых запусков значимость T_0 при новой оценке снижается до $\frac{1}{8}$.

Выбор следующим самого короткого процесса

- Предположим, что оценка времени выполнения одной команды составляет T_0 . Предположим, что следующая оценка этого времени составляет T_1 .
- Мы можем обновить наш расчет, взяв взвешенную сумму этих двух чисел, то есть $aT_0 + (1 - a)T_1$.
 - Выбирая значение a , можно решить, стоит ли при оценке процесса быстро забывать его предыдущие запуски или нужно запоминать их надолго.
- При $a = \frac{1}{2}$ мы получаем следующую последовательность вычислений:
 $T_0; \frac{T_0}{2} + \frac{T_1^2}{2}; \frac{T_0}{4} + \frac{T_1}{4} + \frac{T_2}{2}; \frac{T_0}{8} + \frac{T_1}{8} + \frac{T_2}{4} + \frac{T_3}{2}.$
- После трех новых запусков значимость T_0 при новой оценке снижается до $\frac{1}{8}$.

Гарантиированное планирование

- В данном планировании, если в процессе работы в системе зарегистрированы n пользователей, то каждый пользователь получите $1/n$ от мощности центрального процессора.
- В однопользовательской системе, имеющей n работающих процессов, при прочих равных условиях каждый из них получит $1/n$ от общего числа процессорных циклов.

Лотерейное планирование

- Основная идея состоит в раздаче процессам лотерейных билетов на доступ к различным системным ресурсам, в том числе к процессорному времени.
- Когда планировщику нужно принимать решение, в случайном порядке выбирается лотерейный билет, и ресурс отдается процессу, обладающему этим билетом.
- Применительно к планированию процессорного времени система может проводить лотерейный розыгрыш 50 раз в секунду, и каждый победитель будет получать в качестве приза 20 мс процессорного времени.

Справедливое планирование

- До сих пор мы предполагали, что каждый процесс фигурирует в планировании сам по себе, безотносительно своего владельца.
- В результате, если пользователь 1 запускает 9 процессов, а пользователь 2 запускает 1 процесс, то при циклическом планировании или при равных приоритетах пользователь 1 получит 90 % процессорного времени, а пользователь 2 — только 10 %.

Справедливое планирование

- Чтобы избежать подобной ситуации, некоторые системы перед планированием работы процесса берут в расчет, кто является его владельцем.
- В этой модели каждому пользователю распределяется некоторая доля процессорного времени и планировщик выбирает процессы, соблюдая это распределение.
- Таким образом, если каждому из двух пользователей было обещано по 50 % процессорного времени, то они его получат, независимо от количества имеющихся у них процессов.

Приоритеты в Windows

- Приоритеты потоков в Windows складываются из двух составляющих:
 - Класса приоритета процесса
 - Относительного приоритета потока
- Итоговый приоритет потока может принимать значения от 0 до 31.
- Чем ближе значение приоритета потока к 31, тем выше приоритет потока.
- Такой приоритет называется **базовым уровнем приоритета потока**.

Классы приоритетов в Windows

Класс приоритета процесса	Описание
Real-time	<p>Потоки в этом процессе обязаны немедленно реагировать на события, обеспечивая выполнение критических по времени задач.</p> <p>Такие потоки вытесняют даже компоненты операционной системы.</p>
High	<p>Потоки в этом процессе тоже должны немедленно реагировать на события, обеспечивая выполнение критических по времени задач.</p> <p>Этот класс присвоен, например, Task Manager, что дает возможность пользователю закрывать больше неконтролируемые процессы.</p>

Классы приоритетов в Windows

Класс приоритета процесса	Описание
Above normal	Класс приоритета, промежуточный между normal и high.
Normal	Потоки в этом процессе не предъявляют особых требований к выделению им процессорного времени.
Below normal	Класс приоритета, промежуточный между normal и idle.
Idle	Потоки в этом процессе выполняются, когда система не занята другой работой. Этот класс приоритета обычно используется для утилит, работающих в фоновом режиме, экранных заставок и приложений, собирающих статистическую информацию.

Относительный приоритет потоков в Windows

Относительный приоритет потока	Описание
Time-critical	Поток выполняется с приоритетом 31 в классе real-time и с приоритетом 15 в других классах
Highest	Поток выполняется с приоритетом на два уровня выше обычного для данного класса
Above normal	Поток выполняется с приоритетом на один уровень выше обычного для данного класса
Normal	Поток выполняется с обычным приоритетом процесса для данного класса
Below normal	Поток выполняется с приоритетом на один уровень ниже обычного для данного класса
Lowest	Поток выполняется с приоритетом на два уровня ниже обычного для данного класса
Idle	Поток выполняется с приоритетом 16 в классе real-time и с приоритетом 1 в других классах

Приоритет потоков в Windows

Относительный приоритет потока	Класс приоритета процесса					
	Idle	Below normal	Normal	Above normal	High	Real-time
Time-critical (критичный по времени)	15	15	15	15	15	31
Highest (высший)	6	8	10	12	15	26
Above normal (выше обычного)	5	7	9	11	14	25
Normal (обычный)	4	6	8	10	13	24
Below normal (ниже обычного)	3	5	7	9	12	23
Lowest (низший)	2	4	6	8	11	22
Idle (простаивающий)	1	1	1	1	1	16

Установка класса приоритета процесса

- Класс приоритета может устанавливаться при создании процесса с помощью **CreateProcess** в параметре **fdwCreate**.
- Либо с помощью функции *BOOL SetPriorityClass(HANDLE hProcess, DWORD fdwPriority);*
- Получить значение приоритета можно с помощью функции *DWORD GetPriorityClass(HANDLE hProcess);*

Установка класса приоритета процесса

Класс приоритета	Идентификатор
Real-time	REALTIME_PRIORITY_CLASS
High	HIGH_PRIORITY_CLASS
Above normal	ABOVE_NORMAL_PRIORITY_CLASS
Normal	NORMAL_PRIORITY_CLASS
Below normal	BELOW_NORMAL_PRIORITY_CLASS
Idle	IDLE_PRIORITY_CLASS

Установка относительного приоритета потоков

- Относительный приоритет потока устанавливается с помощью функции *BOOL SetThreadPriority(HANDLE hThread, int nPriority);*
- Получить значение относительного приоритета потока можно с помощью функции *int GetThreadPriority(HANDLE hThread);*

Установка относительного приоритета потоков

Относительный приоритет потока	Идентификатор
Time-critical	THREAD_PRIORITY_TIME_CRITICAL
Highest	THREAD_PRIORITY_HIGHEST
Above normal	THREAD_PRIORITY_ABOVE_NORMAL
Normal	THREAD_PRIORITY_NORMAL
Below normal	THREAD_PRIORITY_BELOW_NORMAL
Lowest	THREAD_PRIORITY_LOWEST
Idle	THREAD_PRIORITY_IDLE

Динамическое изменение приоритетов потоков в Windows

- Базовый приоритет потока может динамически изменяться системой, если этот приоритет находится в пределах между уровнями 0 и 15.
- Система повышает базовый приоритет потока на 2 в двух случаях:
 - при получении потоком сообщения;
 - при переходе потока в состояние готовности.
- В процессе выполнения базовый приоритет такого потока понижается на 1, с каждым отработанным интервалом времени, но никогда не опускается ниже исходного базового приоритета

Динамическое изменение приоритетов потоков в Windows

- Для динамического управлением приоритетами потоков в Windows предназначены следующие функции:
 - **SetProcessPriorityBoost** – позволяет отменить или установить режим динамического изменения базового приоритета всех потоков процесса;
 - **GetProcessPriorityBoost** – позволяет узнать, разрешен ли режим динамического изменения базовых приоритетов потоков процесса;
 - **SetThreadPriorityBoost** – позволяет отменить или установить режим динамического изменения базового приоритета только одного потока;
 - **GetThreadPriorityBoost** – позволяет узнать, разрешен ли режим динамического изменения базового приоритета потока.

Синхронизация процессов и потоков

Атомарные операции

- Действие называется **атомарным (atomic)**, или **непрерываемым**, или **непрерывным** если они удовлетворяет двум требованиям:
 - не прерывается во время своего исполнения;
 - контекст действия изменяется только самим действием.

Атомарные операции

- Атомарные действия делят на две группы:
 - элементарные атомарные действия (fine grained atomic actions);
 - составные атомарные действия (coarse grained atomic actions).

Элементарные атомарные операции

- К **элементарным** атомарным действиям относятся команды микропроцессора, которые **не могут быть прерваны** во время своего исполнения.
- Условно к атомарным можно отнести следующие команды микропроцессора:
 - операции над данными, хранящимися в регистрах микропроцессора;
 - операции чтения данных из памяти в регистры микропроцессора;
 - операции записи данных в память из регистров микропроцессора.

Составные атомарные операции

- К **составным** атомарным действиям относятся последовательности элементарных атомарных действий, которые не прерываются во время своего исполнения.

Частные и разделяемые переменные

- Переменная, доступ к которой имеет только один поток, называется **частной (private)** или личной переменной потока.
- Переменная, доступ к которой имеют несколько одновременно исполняемых (параллельных, конкурирующих) потоков, называется переменной **разделяемой (shared)** потоками

Атомарные и неатомарные операции

```
shared x, y;  
private a, b;
```

```
a = x;           // атомарное действие  
y = b;           // атомарное действие
```

```
x = x + 1;      // неатомарное действие, которое эквивалентно  
                  // следующей последовательности атомарных действий
```

```
private r;  
r = x;  
++r;  
x = r;
```

```
x = y;           // неатомарное действие, которое эквивалентно  
                  // следующей последовательности атомарных действий
```

```
private r;  
r = y;  
x = r;
```

Параллельные и псевдопараллельные потоки

- Одновременно исполняемые потоки называются **параллельными**, если каждый из них исполняется своим процессором (физическими ядрами).
- Одновременно исполняемые потоки называются **псевдопараллельными** или конкурирующими (concurrent), если они исполняются одним процессором (одним физическим ядром).

ГОНКИ ПОТОКОВ

- Если результат исполнения псевдопараллельных потоков зависит от последовательности атомарных действий, исполняемых этими потоками, то говорят, что эти потоки находятся в **состоянии гонки** (race condition).
- Как правило, состояние гонки является причиной ошибок работы многопоточных приложений.
- Причиной состояния гонки потоков является неправильная синхронизация этих потоков.

Синхронизация

- Неформально, под синхронизацией параллельных потоков понимают обмен между этими потоками управляющими сигналами, которые координируют их исполнение.
- Если рассматривать параллельные потоки формально, то синхронизация таких потоков это достижение некоторого фиксированного порядка (соотношения) между управляющими сигналами, которыми обмениваются эти потоки.

Синхронизация

- Порядок управляющих сигналов обеспечивает некоторые фиксированные последовательности атомарных действий, исполняемых параллельными потоками.
- Следовательно, можно сказать, что синхронизация параллельных потоков – это упорядочивание атомарных действий, исполняемых этими потоками.

Условное атомарное действие

- Под синхронизацией параллельных потоков понимается исполнение потоком атомарного действия в зависимости от некоторого условия.
- Такое атомарное действие называется **условным**.
- Другими словами, с точки зрения синхронизации потоков, каждый поток последовательно исполняет условные атомарные действия.

Условное атомарное действие

- Введем для условного атомарного действия следующее обозначение: **<await(условие) действие>**
 - где условие является логическим (булевым) выражением
- Условное атомарное действие выполняется следующим образом:
 - оператор await ждет до тех пор, пока значение условия не станет истинным;
 - как только условие стало истинным, выполняется действие

Взаимное исключение

- Взаимное исключение **<await(true) действие> := <действие>**
- В этом случае происходит безусловное выполнение атомарного действия.
- Этот случай называется **взаимным исключением**.
- Код, исполняемый внутри атомарного действия, называется **критической секцией**.

Взаимное исключение

- Иными словами, **взаимное исключение** – это некий способ, обеспечивающий правило, при котором если общие данные или файл используются одним процессом, возможность их использования всеми другими процессами исключается.

Условная синхронизация

- Условная синхронизация **<await(условие)>**
- В этом случае оператор await оповещает о наступлении некоторого события, т. е. что произошло некоторое действие.
- Этот случай называется **условная синхронизация**.

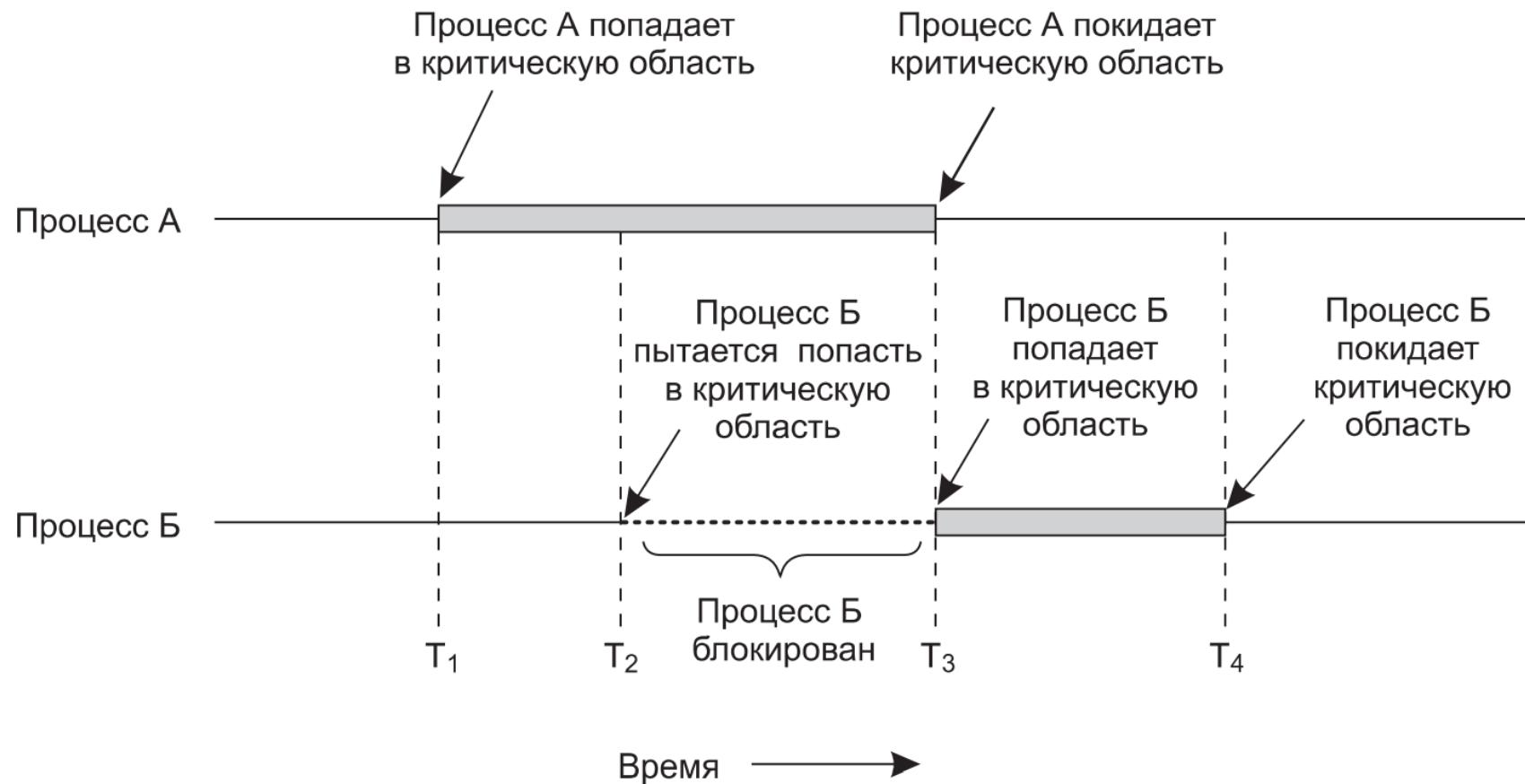
Проблема взаимного исключения

- Если бы удалось все выстроить таким образом, чтобы никакие два потока не находились одновременно в своих критических секциях, это позволило бы избежать гонки потоков.
- Хотя выполнение этого требования позволяет избежать состязательных ситуаций, его недостаточно для того, чтобы параллельные потоки правильно выстраивали совместную работу и эффективно использовали общие данные.

Проблема взаимного исключения

- Для решения данной проблемы необходимо соблюдение четырех условий:
 1. Два потока не могут одновременно находиться в своих критических областях.
 2. Не должны выстраиваться никакие предположения по поводу скорости или количества центральных процессоров.
 3. Никакие потоки, выполняемые за пределами своих критических областей, не могут блокироваться любым другим потоком.
 4. Потоки не должны находиться в вечном ожидании входа в свои критические области.

Проблема взаимного исключения



Решение проблемы взаимного исключения

- Проблема взаимного исключения можно решить двумя типами алгоритмов:
 - Алгоритмами с активным ожиданием
 - Алгоритмами с приостановкой и последующей активизации

Алгоритмы с активным ожиданием: Запрещение прерываний

- В однопроцессорных системах простейшим решением является запрещение всех прерываний каждым потоком сразу после входа в критическую область и их разрешение сразу же после выхода из критической области.
- Данный способ не работает в многопроцессорных (многоядерных) системах.
- К тому же возникает проблема, что пользователи могут забывать снимать запрет прерываний, что может вызывать необратимые последствия для операционной системы.

Алгоритмы с активным ожиданием: Блокирующие переменные

- В данном алгоритме используется одна общая (блокирующая) переменная, исходное значение которой равно нулю.
- Когда потоку требуется войти в свою критическую область, сначала он проверяет значение блокирующей переменной.
 - Если оно равно 0, процесс устанавливает его в 1 и входит в критическую область.
 - Если значение уже равно 1, процесс просто ждет, пока оно не станет равно нулю.
- В данном подходе существует гонка за общую блокирующую переменную.

Алгоритмы с активным ожиданием: Строгое чередование

```
while(TRUE) {  
    while(turn!=0)      /*цикл*/;  
    critical_region();  
    turn=1;  
    noncritical_region();  
}
```

a

```
while(TRUE) {  
    while(turn!=0)      /*цикл*/;  
    critical_region();  
    turn=0;  
    noncritical_region();  
}
```

б

Рис. 2.17. Предлагаемое решение проблемы критической области: *а* — процесс 0; *б* — процесс 1. В обоих случаях следует убедиться, что в коде присутствует точка с запятой, завершающая оператор while

Алгоритмы с активным ожиданием: Строгое чередование

- В данном алгоритме переменная **turn** указывает на номер потока\процесса, которому разрешено входить в критическую секцию.
- По завершении работы, каждый поток обновляет значение этой переменной, чтобы разрешить работу другому потоку.
- Однако данный алгоритм нарушает условие 3 решения проблемы взаимного исключения:
 - Никакие потоки, выполняемые за пределами своих критических областей, не могут блокироваться любым другим потоком.

Алгоритмы с активным ожиданием: Алгоритм Петерсона

```
bool x1, x2;
int q;      // обеспечивает ассиметричное решение задачи взаимного исключения

x1 = false;
x2 = false;

void thread1()
{
    while (true)
    {
        nonCriticalSection1();

        x1 = true;          // поток 1 хочет войти в критическую секцию
        q = 2;              // но, сначала предоставляет право входа потоку 2
        while (x2 && q == 2); // ждет, пока поток 2 находится в своей критич. секции
        criticalSection1();

        x1 = false;
    }
}
```

Алгоритмы с активным ожиданием:

Алгоритм XCHG

```
void xchg(register int r, int* x)
{
    register int temp;

    temp = r;
    r = *x;
    *x = temp;
}
```

- Данный алгоритм использует атомарность работы с регистрами процессора.
- То есть функция xchg является атомарной.

Алгоритмы с активным ожиданием: Алгоритм XCHG

```
int lock = 0;
void thread_i()
{
    while (true)
    {
        register int key_i = 1;          // ключ для входа в критическую секцию
        while (key_i == 1)              // ждем, пока вход закрыт
            xchg(key_i, &lock);
        criticalSection_i();
        xchg(key_i, &lock);           // выход из критической секции
        nonCriticalSection_i();
    }
}
```

Алгоритмы с активным ожиданием

- Постоянная проверка значения переменной, пока она не приобретет какое-нибудь значение, называется **активным ожиданием**.
- Как правило, этого ожидания следует избегать, поскольку оно тратит впустую время центрального процессора.
- Активное ожидание используется только в том случае, если есть основание полагать, что оно будет недолгим.
- Такие алгоритмы с активным ожиданием называются **спин-блокировкой**.

Алгоритмы с приостановкой и активацией

- Такие алгоритмы имеют преимущество перед алгоритмами с активным ожиданием, т.к. не расходуют в пустую время процессора.
- Простейшим примером реализаций таких алгоритмов является функция `Sleep(int milliseconds);`
- Другими примерами являются все известные примитивы синхронизации: семафор, мьютекс, событие.

Примитивы синхронизации

Не примитивы синхронизации

- Interlocked-функции – такие функции предоставляют возможность выполнять атомарные операции с переменными, использующимися несколькими потоками.
- Такие функции могут выполняться с помощью XCHG-функции (см. лекцию OS-4).
- Являются наиболее предпочтительными, когда операции являются примитивными и есть возможность такие функции использовать.

Не примитивы синхронизации в Win32 API

- LONG **InterlockedIncrement**([in, out] LONG volatile *Addend);
 - LONG64 **InterlockedIncrement64**([in, out] LONG64 volatile *Addend);
- LONG **InterlockedDecrement**([in, out] LONG volatile *Addend);
 - LONG64 **InterlockedDecrement64**([in, out] LONG64 volatile *Addend);

Не примитивы синхронизации в Win32 API

- LONG `InterlockedExchange([in, out] LONG volatile *Target, [in] LONG Value);`
 - PVOID `InterlockedExchangePointer([in, out] PVOID volatile *Target, [in] PVOID Value);`
 - SHORT `InterlockedExchange16([in, out] SHORT volatile *Destination, [in] SHORT ExChange);`
 - LONG64 `InterlockedExchange64([in, out] LONG64 volatile *Target, [in] LONG64 Value);`
- LONG `InterlockedExchangeAdd([in, out] LONG volatile *Addend, [in] LONG Value);`
 - LONG64 `InterlockedExchange64([in, out] LONG64 volatile *Target, [in] LONG64 Value);`

Не примитивы синхронизации в Win32 API

- LONG [InterlockedCompareExchange](#)([in, out] LONG volatile *Destination, [in] LONG ExChange, [in] LONG Comperand);
 - PVOID [InterlockedCompareExchangePointer](#)([in, out] PVOID volatile *Destination, [in] PVOID Exchange, [in] PVOID Comperand);
 - LONG64 [InterlockedCompareExchange64](#)([in, out] LONG64 volatile *Destination, [in] LONG64 ExChange, [in] LONG64 Comperand);

Не примитивы синхронизации в Win32 API

- LONG `InterlockedAnd([in, out] LONG volatile *Destination, [in] LONG Value);`
- LONG `InterlockedOr([in, out] LONG volatile *Destination, [in] LONG Value);`
- LONG `InterlockedXor([in, out] LONG volatile *Destination, [in] LONG Value);`

Не примитивы синхронизации в C++ 11

- `template< class T > struct atomic;`
- Объявлен в заголовочном файле `<atomic>`
- Реализует те же функции, что и в Win32, с помощью операторов

Несколько примитивов синхронизации в C++ 11

fetch_add	atomically adds the argument to the value stored in the atomic object and obtains the value held previously (public member function)
fetch_sub	atomically subtracts the argument from the value stored in the atomic object and obtains the value held previously (public member function)
fetch_and	atomically performs bitwise AND between the argument and the value of the atomic object and obtains the value held previously (public member function)
fetch_or	atomically performs bitwise OR between the argument and the value of the atomic object and obtains the value held previously (public member function)
fetch_xor	atomically performs bitwise XOR between the argument and the value of the atomic object and obtains the value held previously (public member function)
operator++ operator++(int)	increments or decrements the atomic value by one
operator-- operator--(int)	(public member function)
operator+= operator-= operator&= operator = operator^=	adds, subtracts, or performs bitwise AND, OR, XOR with the atomic value (public member function)

Не примитивы синхронизации в других языках

- System.Threading.Interlocked в C#
- java.util.concurrent.atomic.AtomicInteger в Java

ПРИМИТИВЫ СИНХРОНИЗАЦИИ

- Замок (Lock) – примеры: критическая секция, мьютекс
- Условие (Condition) – пример: событие (Event)
- Семафор (Semaphore)

Замок (Lock)

```
class Lock
{
    bool free;
    ThreadQueue tq;           // очередь потоков
public:
    Lock(): free(true) { }
    ~Lock() { . . . }
    void acquire();           // закрываем замок
    void release();           // открываем замок
};
```

Замок (Lock)

```
void Lock::acquire() // закрываем замок
{
    disableInterrupt();
    if (!free)
        tq.enqueueThread(currentThread());
    else
        free = false;
    enableInterrupt();
}
```

```
void Lock::release() // открываем замок
{
    disableInterrupt();
    if (!tq.dequeueThread())
        free = true;
    enableInterrupt();
}
```

Замок (Lock)

```
Lock lock;  
void thread_1( )  
{  
    beforeCriticalSection_1();  
    lock.acquire();  
    criticalSection_1();  
    lock.release();  
    afterCriticalSection_1();  
}  
  
void thread_2( )  
{  
    beforeCriticalSection_2();  
    lock.acquire();  
    criticalSection_2();  
    lock.release();  
    afterCriticalSection_2();  
}
```

Замок (Lock) в Win32 API – Критическая секция

- Имеет тип CRITICAL_SECTION.
- Не является объектом ядра.
- Работает только в рамках одного процесса.
- Работает быстрее, чем примитивы синхронизации – объекты ядра.

Замок (Lock) в Win32 API – Критическая секция

- *InitializeCriticalSection* – инициализация объекта;
- *EnterCriticalSection* – вход в критическую секцию;
- *TryEnterCriticalSection* – попытка входа в критическую секцию;
- *LeaveCriticalSection* – выход из критической секции;
- *DeleteCriticalSection* – завершение работы с объектом;

Замок (Lock) в Win32 API – Мьютекс

- Является объектом ядра.
- Используется при синхронизации потоков в разных процессах.
- Мьютекс находится в сигнальном состоянии, если он не принадлежит ни одному потоку.
- В противном случае мьютекс находится в несигнальном состоянии.
- Одновременно мьютекс может принадлежать только одному потоку.

Замок (Lock) в Win32 API – Мьютекс

- *CreateMutex* – создание мьютекса;
- *OpenMutex* – получение доступа к существующему мьютексу;
- *ReleaseMutex* – освобождение мьютекса (переход мьютекса в сигнальное состояние);
- *WaitForSingleObject* или *WaitForMultipleObjects* – захват мьютекса (ожидание сигнального состояния мьютекса).

Замок (Lock) в C++ стандарте

- `class mutex;`
- Объявлен в заголовочном файле `<mutex>`
- В C++ стандарте работает только в рамках одного процесса.
- Основные функции для работы:
 - `lock`
 - `try_lock`
 - `Unlock`
- Также с мьютексом можно работать с класса `std::lock_guard`.

Замок (Lock) в других языках

- Lock и System.Threading.Mutex в C#
 - lock является примером критической секции `lock (x) { // Your code... }`
 - Mutex класс позволяет синхронизировать потоки в разных процессах
- Synchronized в Java – пример критической секции

Условие (Condition)

```
class Condition
{
    bool event;
    ThreadQueue tq;      // очередь потоков
public:
    Condition (): event(false) { }
    ~Condition () { ... }
    void wait();          // ждем выполнения условия
    void signal();        // сигнализируем о выполнении условия
};
```

Условие (Condition)

```
// ждем выполнения условия  
void Condition::wait()  
{  
    disableInterrupt();  
    if (event)  
        event = false;  
    else  
        tq.enqueueThread(currentThread());  
    enableInterrupt();  
}
```

```
// сигнализируем о выполнении условия  
void signal()  
{  
    disableInterrupt();  
    if (!tq.dequeueThread())  
        event = true;  
    enableInterrupt();  
}
```

Условие (Condition)

Решение задачи условной синхронизации для двух потоков
при помощи примитива синхронизации Condition

- Для этого рассмотрим следующие потоки.

```
Condition c;           // начальное состояние несигнальное

void thread_1( )        void thread_2( )
{
    beforeCondition_1();      {
    c.wait();                  beforeCondition_2();
    afterCondition_1();      c.signal();
}                           afterCondition_2();
```

Решение при помощи примитива синхронизации Condition
проблемы взаимного исключения для двух потоков

- Для этого рассмотрим следующие потоки:

```
Condition c;           // начальное состояние несигнальное
c.signal();             // устанавливаем в сигнальное состояние

void thread_1( )        void thread_2( )
{
    beforeCriticalSection_1();      {
    c.wait();                  beforeCriticalSection_2();
    criticalSection_1();          c.wait();
    c.signal();                criticalSection_2();
    afterCriticalSection_1();    c.signal();
}                           afterCriticalSection_2();
```

Условие (Condition) в Win32 API

- Является объектом ядра.
- Бывают двух типов
 - С ручным сбросом
 - С автоматическим сбросом
- Событие с ручным сбросом можно перевести в несигнальное состояние только посредством вызова функции *ResetEvent*.
- Событие с автоматическим сбросом переходит в несигнальное состояние
 - как при помощи функции *ResetEvent*,
 - так и при помощи функций ожидания *WaitForSingleObject* или *WaitForMultipleObjects*.

Условие (Condition) в Win32 API

- *CreateEvent* – создание события;
- *OpenEvent* – получение доступа к существующему событию;
- *SetEvent* – перевод события в сигнальное состояние;
- *ResetEvent* – перевод события в несигнальное состояние;
- *PulseEvent* – освобождение нескольких потоков, ждущих сигнального состояния события с ручным сбросом;
- *WaitForSingleObject* или *WaitForMultipleObjects* – ожидание наступления события (перехода события в сигнальное состояние)

Семафор (Semaphore)

- Семафор – это неотрицательная целочисленная переменная, значение которой может изменяться только при помощи атомарных операций.
- Семафор считается **свободным**, если его значение больше нуля, в противном случае семафор считается **занятым**.
- Семафор, который может принимать только значения 0 или 1, называется **двоичным** или **бинарным** семафором.
- Семафор, значение которого может быть больше 1, обычно называют **считывающими** семафором.
- Если очередь семафора обслуживается по алгоритму FIFO, то семафор называется **сильным**, иначе – **слабым**.

Семафор (Semaphore)

```
class Semaphore
{
    int count;           // счетчик
    ThreadQueue tq;    // очередь потоков
public:
    Semaphore(int& n): count(n) {}
    ~Semaphore() { . . . }
    void wait();        // закрыть семафор
    void signal();      // открыть семафор
};
```

Семафор (Semaphore)

```
void Semaphore::wait()    // закрыть семафор
{
    disableInterrupt();
    if (count > 0)
        --count;
    else
        tq.enqueueThread(currentThread());
    enableInterrupt();
}
```

```
void Semaphore::signal()  // открыть семафор
{
    disableInterrupt();
    if (!tq.dequeueThread())
        ++count;
    enableInterrupt();
}
```

Семафор (Semaphore) в Win32 API

- Является объектом ядра.
- Семафор находится в сигнальном состоянии, если его значение больше нуля.
- В противном случае семафор находится в несигнальном состоянии.

Семафор (Semaphore) в Win32 API

- *CreateSemaphore* – создание семафора;
- *OpenSemaphore* – получение доступа к существующему семафору;
- *ReleaseSemaphore* – увеличение значения семафора на положительное число;
- *WaitForSingleObject* или *WaitForMultipleObjects* – ожидание перехода семафора в сигнальное состояние.

Семафор (Semaphore) в C++ стандарте

- *std::counting_semaphore, std::binary_semaphore*
 - counting_semaphore – это шаблонизированный класс.
 - `using binary_semaphore = std::counting_semaphore<1>;`
- Появился только в C++20 стандарте.
- Основные функции для работы:
 - release
 - acquire
 - try_acquire
 - try_acquire_for
 - try_acquire_until

Семафор (Semaphore) в других языках

- System.Threading.Semaphore в C#
- java.util.concurrent.Semaphore в Java

Классические задачи синхронизации

Классические задачи синхронизации

- Классические задачи синхронизации – это задачи синхронизации, наиболее часто встречающиеся на практике.

Классические задачи синхронизации

- Задача производителей и потребителей (producer-consumer)
- Задача читателей и писателей
- Задача о спящем парикмахере
- Задача об обедающих философах

Задача производителей и потребителей (producer-consumer)

- Существуют процессы двух типов: производители и потребители.
 - Производители производят некоторый ресурс, а потребители – потребляют его.
- Произведенные ресурсы хранятся на складе.
- Требуется обеспечить такой доступ производителей и потребителей к складу (ограниченному буферу), при котором выполняются следующие требования:
 1. Каждый ресурс может быть помещен только в одну ячейку склада.
 2. Не может быть произведено ресурсов больше, чем вместимость склада.
Считается, что ресурс произведен, если он поставлен на склад.
 3. Не может быть потреблено ресурсов больше, чем хранится на складе.
Считается, что ресурс потреблен, если он взят со склада.
 4. Один ресурс не может быть потреблен дважды.

Задача производителей и потребителей (producer-consumer)

- Если существует ограничение на размер буфера ('склада'), то данную задачу называют задачей ограниченного буфера (bounded buffer problem)
- Однако буфер может быть и бесконечным.
- Для решения данной задачи требуется три примитива синхронизации:
 - Замок для синхронизированного доступа к элементам буфера
 - Семафор для определения пустоты буфера
 - Семафор для определения полноты буфера (если имеется ограничение на размер)

Задача производителей и потребителей (producer-consumer)

```
Lock mutex;           // блокировка доступа к складу
Semaphore empty(n), full(0); // n – вместимость склада

void producer()
{
    while(true)
    {
        produceProduct(); // произвести продукт
        empty.wait();     // уменьшить кол-во свободных ячеек на складе
        mutex.acquire(); // вход в критическую секцию
        putProduct();    // поместить продукт на склад
        mutex.release(); // выход из критической секции
        full.signal();   // увеличить кол-во продуктов на складе
    }
}

void consumer()
{
    while(true)
    {
        full.wait();      // уменьшить кол-во продуктов на складе
        mutex.acquire(); // вход в критическую секцию
        getProduct();    // взять продукт со склада
        mutex.release(); // выход из критической секции
        empty.signal();  // увеличить кол-во своб. ячеек на складе
        consumeProduct(); // потребить продукт
    }
}
```

Задача читателей и писателей

- Существуют процессы двух типов: читатели и писатели.
 - Читатели могут только читать данные, а писатели могут изменять данные.
- Требуется обеспечить следующую процедуру доступа к данным:
 1. Одновременно изменять данные может только один писатель.
 2. Одновременно читать данные могут несколько читателей.
 3. Если писатель изменяет данные, то доступ для читателей к данным закрыт.
 4. Если хотя бы один читатель читает данные, то писатель не может их изменять.

Задача читателей и писателей

- В данной задаче моделируется доступ к базе данных.
 - Представим, к примеру, систему бронирования авиабилетов, в которой есть множество соревнующихся процессов, желающих обратиться к ней для чтения и записи.
 - Вполне допустимо наличие нескольких процессов, одновременно считывающих информацию из базы данных, но если один процесс обновляет базу данных, никакой другой процесс не может получить доступ к базе данных даже для чтения информации.
- Для решения данной задачи требуются два примитива синхронизации:
 - Замок для работы со счетчиком читателей
 - Семафор для запрета записи

Задача читателей и писателей

```
Integer readerCount =0; // количество читателей  
Lock mutex;  
Semaphore write(1); // количество писателей  
  
void writer()  
{  
    write.wait(); // захватить доступ к ресурсу  
    modifyData(); // изменить данные  
    write.signal(); // освободить доступ к ресурсу  
}
```

```
void reader()  
{  
    mutex.acquire(); // войти в критическую секцию  
    ++readerCount; // увеличить количество читателей  
    if (readerCount == 1) // если первый читатель,  
        write.wait(); // то запретить писать  
    mutex.release(); // выйти из критической секции  
  
    readData(); // читать данные  
  
    mutex.acquire(); // войти в критическую секцию  
    --readerCount; // уменьшить количество читателей  
    if (readerCount == 0) // если читателей нет,  
        write.signal(); // разрешить писать  
    mutex.release(); // выйти из критической секции  
}
```

Задача о спящем парикмахере

- Парикмахерская имеет комнату для стрижки клиента и комнату ожидания, в которой находятся n стульев.
- Требуется обеспечить следующий порядок работы парикмахерской:
 1. Парикмахер после обслуживания каждого клиента заходит в комнату ожидания и приглашает следующего клиента.
 2. Если клиента нет, то парикмахер идет спать.
 3. Если клиент заходит в парикмахерскую и находит парикмахера спящим, то он должен разбудить его.
 4. Если клиент заходит в парикмахерскую и видит, что все стулья в комнате ожидания заняты, то он уходит.
 5. Если клиент заходит в парикмахерскую и видит в комнате ожидания свободные стулья, то он занимает очередь.

Задача о спящем парикмахере

- В данной задаче основной проблемой является эффективное использование времени парикмахера и клиента и избегание гонки потоков:
 - Парикмахер спит, а клиент ждет парикмахера.
 - Два клиента пришли в одно время и пытаются сесть на один стул.
- Для решения данной задачи необходимы:
 - Замок для управления свободными местами
 - Семафор для управления парикмахером
 - Семафор для управления клиентами

Задача о спящем парикмахере

```
Semaphore customer(0);  
Semaphore barber(0);  
Lock accessSeat;  
int nFreeSeat = n; // количество свободных мест
```

Задача о спящем парикмахере

```
void barber()
{
    while(true) // runs in an infinite loop
    {
        customer.wait();      // ждет посетителя

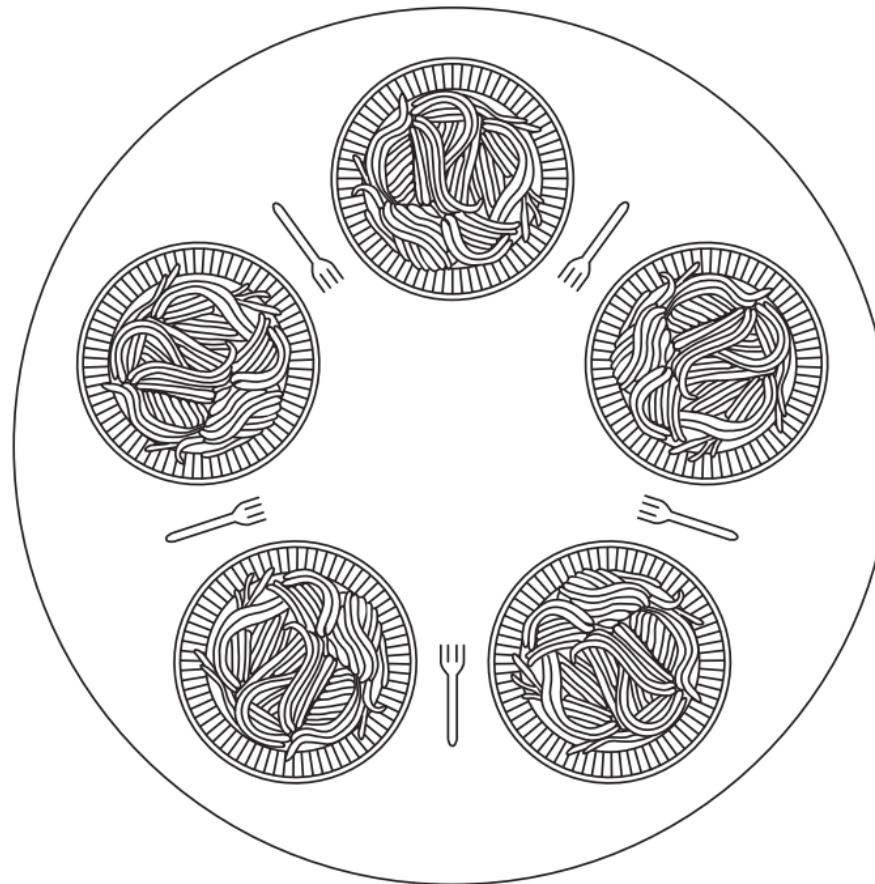
        accessSeat.acquire(); // закр. вход в зал ожидания
        ++nFreeSeat;          // приглашает клиента
        accessSeat.release(); // откр. вход в зал ожидания

        barber.signal();      // приступает к работе
        cuttingHair();         // стрижет
    }
}
```

```
void customer()
{
    accessSeat.acquire(); // заходит в зал ожидания и закрывает вход
    if (nFreeSeat > 0)   // если есть свободные места
    {
        --nFreeSeat;       // занимает место
        customer.signal(); // становится в очередь на обслуживание
        accessSeat.release(); // открывает вход в зал ожидания

        barber.wait();      // ждет в очереди обслуживания парикмахера
        haveCutingHair();  // стрижется
        exit();              // уходит
    }
    else                  // свободных мест нет
    {
        accessSeat.release(); // открывает вход в зал ожидания
        exit();                // уходит
    }
}
```

Задача об обедающих философах



Задача об обедающих философах

- Пять философов сидят за круглым столом.
- Перед каждым философом находится блюдо спагетти, которое наполняет специальный слуга.
- На столе лежат ровно пять вилок – по одной между каждыми двумя философами соседями.
- Каждый из пяти философов ведет простую жизнь:
 - некоторое время он предается размышлению,
 - а некоторое время ест спагетти.
- Проблема состоит в том, что спагетти можно есть только двумя вилками, которые лежат рядом с тарелкой.
- Требуется обеспечить такой порядок приема пищи философами, чтобы никто из них не умер с голоду.

Задача об обедающих философах

- Данная задача показывает способ избегания проблемы взаимной блокировки (deadlock)
- Изначальная цель Дейкстры при формулировании «проблемы обедающих философов» заключалась в демонстрации возможных проблем при работе с внешними устройствами компьютера, например, ленточными накопителями.
- Тем не менее, область применения данной задачи простирается гораздо дальше и сложности, рассматриваемые в задаче, чаще возникают, когда несколько процессов пытаются получить доступ к набору данных, который обновляется.

Задача об обедающих философах – ошибочное решение

```
#define N 5                                /* количество философов */

void philosopher(int i)                    /* i: номер философа (от 0 до 4) */
{
    while (TRUE) {
        think();                            /* философ размышляет */
        take_fork(i);                      /* берет левую вилку */
        take_fork((i+1) % N);              /* берет правую вилку; */
                                             /* % - оператор деления по модулю */
        eat();                              /* ест спагетти */
        put_fork(i);                       /* кладет на стол левую вилку */
        put_fork((i+1) % N);              /* кладет на стол правую вилку */
    }
}
```

Задача об обедающих философах – ошибочное решение

- Решение, представленное на предыдущем слайде ошибочно.
- Допустим, что все пять философов одновременно берут левую от себя вилку.
- Тогда никто из них не сможет взять правую вилку, что приведет к **взаимной блокировке**

Задача об обедающих философах – ошибочное решение

- Можно изменить программу так, чтобы после получения левой вилки программа проверяла доступность правой вилки.
- Если эта вилка недоступна, философ кладет на место левую вилку, ожидает какое-то время, а затем повторяет весь процесс.
- Это предложение также ошибочно, но уже по другой причине.
- При некоторой доле невезения все философы могут приступить к выполнению алгоритма одновременно и, взяв левые вилки и увидев, что правые вилки недоступны, опять одновременно положить на место левые вилки, и так до бесконечности.
- Подобная ситуация, при которой все программы бесконечно работают, но не могут добиться никакого прогресса, называется **голоданием**, или **зависанием процесса**.

Задача об обедающих философах

```
#define N      5          /* количество философов */
#define LEFT    (i+N-1) %N  /* номер левого соседа для i-го философа */
#define RIGHT   (i+1) %N   /* номер правого соседа для i-го
                           философа */

#define THINKING 0         /* философ размышляет */
#define HUNGRY   1         /* философ пытается взять вилки */
#define EATING    2         /* философ ест спагетти */
typedef int semaphore;

int state[N];
semaphore mutex = 1;
semaphore s[N];

void philosopher(int i)
{
    while (TRUE) {
        think();
        take_forks(i);
        eat();
        put_forks(i);
    }
}
```

```
void take_forks(int i)           /* i - номер философа (от 0 до N-1) */
{
    down(&mutex);
    state[i] = HUNGRY;

    test(i);
    up(&mutex);
    down(&s[i]);
}

void put_forks(i)               /* i - номер философа (от 0 до N-1) */
{
    down(&mutex);
    state[i] = THINKING;
    test(LEFT);
    test(RIGHT);

    up(&mutex);
}

void test(i)                    /* i - номер философа (от 0 до N-1) */
{
    if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
        state[i] = EATING;
        up(&s[i]);
    }
}
```

Двоичный семафор vs замок

- Двоичный семафор работает в схожем ключе, как и замок
 - Если значение = 1, то семафор свободен – иначе занят.
- Но двоичный семафор не является тем же самым, что и замок.
- Замок может освобождаться только тем потоком, который его захватил.
- Семафор же может уменьшаться\увеличиваться любым потоком.

Тупики (взаимоблокировки,
deadlocks)

Тупик

- Тупик (взаимоблокировка, deadlock) – это ситуация, в которой несколько процессов ожидают ресурсы, занятые друг другом, и ни один из них не может продолжить свое исполнение.
- Тупики могут возникать как при работе с аппаратными, так и с программными ресурсами.

Пример

Без блокировки

```
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);  
}
```

Потенциальная блокировка

```
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);  
}
```

Классификация системных ресурсов по доступу

- Классификация ресурсов по доступу:
 - совместно используемые ресурсы;
 - монопольные ресурсы.
- Совместно используемый ресурс может одновременно использоваться несколькими потоками, например, файл.
- Монопольный ресурс может быть предоставлен потоку только в монопольное использование, например, принтер или CD-привод.

Классификация ресурсов по способу распределения

- Классификация ресурсов по способу распределения:
 - перераспределяемые (preemptable) ресурсы;
 - не перераспределяемые (non-preemptable) ресурсы.
- Перераспределяемый ресурс может быть отобран у потока, владеющего этим ресурсом, и распределен другому потоку, например, страница реальной памяти.
- Не перераспределяемый ресурс не может быть отобран у потока, который владеет этим ресурсом, например, принтер.

Классификация ресурсов по времени существования

- Классификация ресурсов по времени существования:
 - повторно используемые ресурсы;
 - потребляемые ресурсы.
- Повторно используемый ресурс может использоваться потоками многократно, например, принтер.
- Потребляемый ресурс может использоваться потоком только один раз, после этого он перестает существовать, например, сообщение из очереди сообщений.

Взаимоблокировки

- **Взаимоблокировка** в группе процессов возникает в том случае, если каждый процесс из этой группы ожидает события, наступление которого зависит исключительно от другого процесса из этой же группы.

Виды взаимоблокировок

- Ресурсная взаимоблокировка – возникает в случае когда событием, наступления которого ожидает каждый процесс, является высвобождение какого-либо ресурса, которым на данный момент владеет другой участник группы.

Виды взаимоблокировок

- Коммуникационная взаимоблокировка – возникает в при обмене данными, в которых один и более процессов связываются путем обмена сообщениями.
 - Общая договоренность предполагает, что процесс А отправляет сообщение-запрос процессу В, а затем блокируется до тех пор, пока В не пошлет назад ответное сообщение.
 - Предположим, что сообщение-запрос где-то затерялось.
 - Процесс А заблокирован в ожидании ответа.
 - Процесс В заблокирован в ожидании запроса на какие-либо его действия.
 - В результате возникает взаимоблокировка
- В данном виде взаимоблокировок решением является – истечение времени ожидания. По истечении времени ожидания клиент повторяет свой запрос.

Условия возникновения ресурсных взаимоблокировок

- Для возникновения ресурсных взаимоблокировок должны выполняться **все** четыре условия:
 1. Условие взаимного исключения. Каждый ресурс либо выделен в данный момент только одному процессу, либо доступен.
 2. Условие удержания и ожидания. Процессы, удерживающие в данный момент ранее выделенные им ресурсы, могут запрашивать новые ресурсы.
 3. Условие невыгрузаемости (неперераспределяемости). Ранее выделенные ресурсы не могут быть принудительно отобраны у процесса. Они должны быть явным образом высвобождены тем процессом, который их удерживает.
 4. Условие циклического ожидания. Должна существовать кольцевая последовательность из двух и более процессов, каждый из которых ожидает высвобождения ресурса, удерживаемого следующим членом последовательности.
- Если хотя бы одно из них не соблюдается, ресурсная взаимоблокировка невозможна.

Моделирование взаимоблокировок

- Для моделирования взаимоблокировок используются графы распределения ресурсов:
 - Круглые вершины отображают процессы
 - Квадратные вершины отображают ресурсы
 - Направленное ребро, которое следует от узла ресурса (квадрата) к узлу процесса (окружности), означает, что этот ресурс был ранее запрошен, получен и на данный момент удерживается этим процессом.
 - Направленное ребро, идущее от процесса к ресурсу, означает, что процесс в данное время заблокирован в ожидании высвобождения этого ресурса.

Моделирование взаимоблокировок

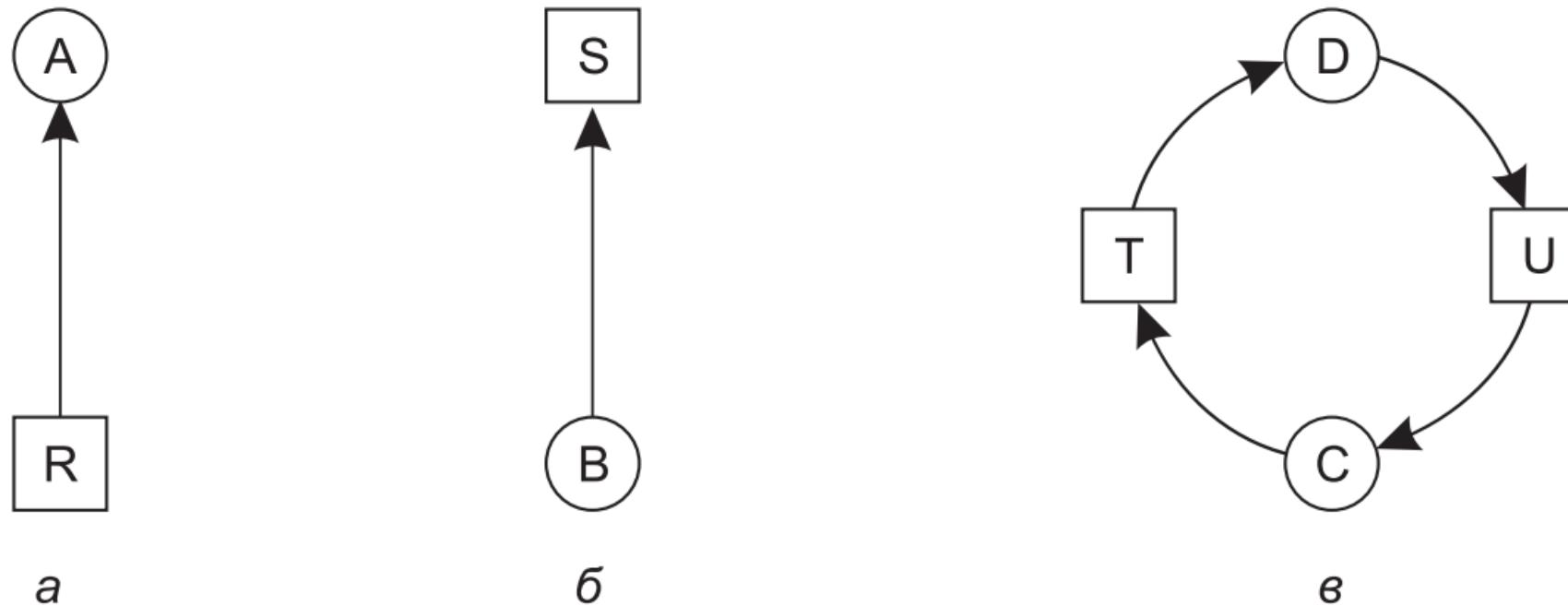


Рис. 6.1. Графы распределения ресурсов: *а* — ресурс занят; *б* — запрос ресурса;
в — взаимоблокировка

Моделирование взаимоблокировок: пример

- Представим, что есть три процесса, A , B и C , и три ресурса, R , S и T .
- Рассмотри различные модели поведения этих процессов

Моделирование взаимоблокировок: пример 1

A

Запросить R
Запросить S
Освободить R
Освободить S

B

Запросить S
Запросить Т
Освободить S
Освободить Т

C

Запросить Т
Запросить R
Освободить Т
Освободить R

Моделирование взаимоблокировок: пример 1

- Операционная система может в любое время запустить любой незаблокированный процесс, то есть она может принять решение запустить процесс А и дождаться, пока он не завершит всю свою работу, затем запустить процесс В и довести его работу до завершения и, наконец, запустить процесс С.
- Такой порядок не приводит к взаимоблокировкам (поскольку отсутствует борьба за овладение ресурсами), но в нем нет и никакой параллельной работы.
- Кроме действий по запросу и высвобождению ресурсов процессы занимаются еще и вычислениями, и вводом-выводом данных.

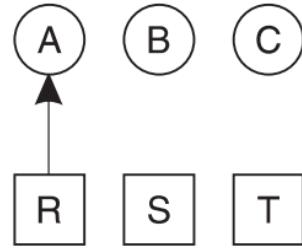
Моделирование взаимоблокировок: пример 1

- Когда процессы запускаются последовательно, отсутствует возможность использования центрального процессора одним процессом, в то время когда другой процесс ожидает завершения операции ввода-вывода.
- Поэтому строго последовательное выполнение процессов может быть неоптимальным решением.
- В то же время, если ни один из процессов не выполняет никаких операций ввода-вывода, алгоритм, при котором кратчайшее задание выполняется первым, более привлекателен, чем циклический алгоритм, поэтому при определенных условиях последовательный запуск всех процессов может быть наилучшим решением.

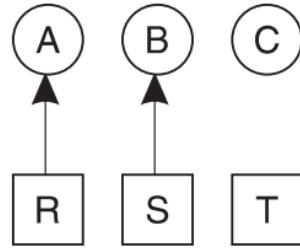
Моделирование взаимоблокировок: пример 2

- A запрашивает R
- B запрашивает S
- C запрашивает T
- A запрашивает S
- B запрашивает T
- C запрашивает R

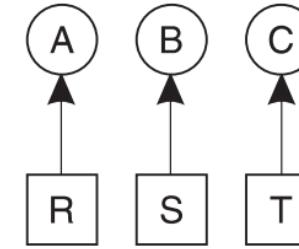
Моделирование взаимоблокировок: пример 2



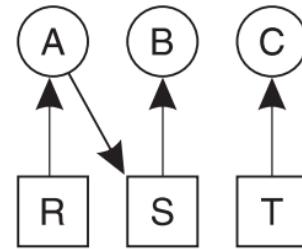
д



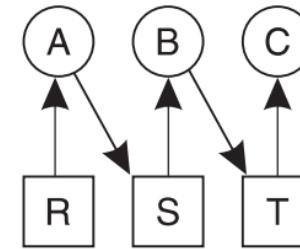
е



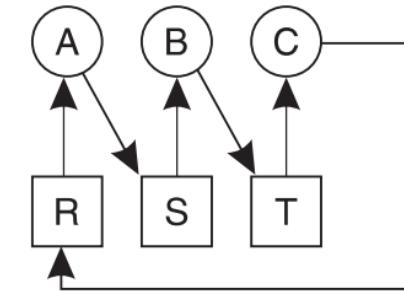
ж



з



и

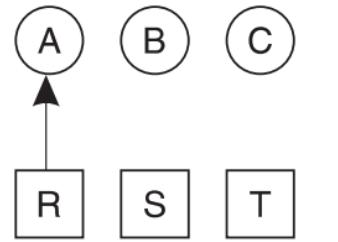


к

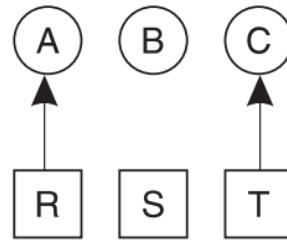
Моделирование взаимоблокировок: пример 3

- A запрашивает R
- C запрашивает T
- A запрашивает S
- C запрашивает R
- A освобождает R
- A освобождает S

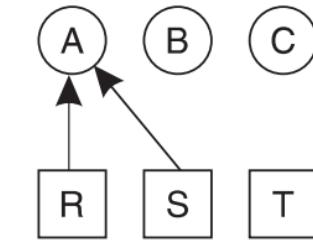
Моделирование взаимоблокировок: пример 3



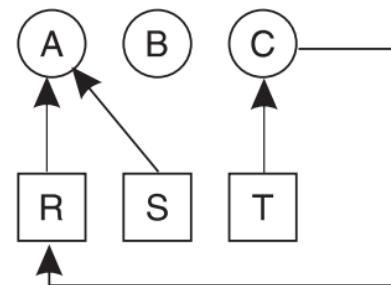
M



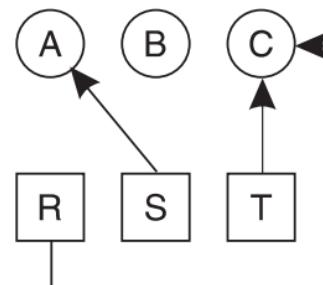
H



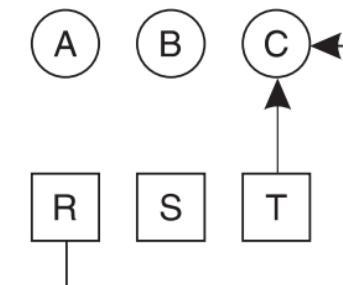
O



P



p



c

Моделирование взаимоблокировок

- Если образуется цикл, значит, возникла взаимоблокировка, а если нет, значит, нет и взаимоблокировки.
- Хотя здесь рассматривался граф ресурсов, составленный для случая использования по одному ресурсу каждого типа, ресурсные графы могут быть применены также для обработки нескольких ресурсов одного и того же типа (Holt, 1972).

Решение проблемы взаимоблокировки

1. Игнорирование проблемы. Может быть, если вы проигнорируете ее, она проигнорирует вас.
2. Обнаружение и восстановление. Дайте взаимоблокировкам проявить себя, обнаружьте их и выполните необходимые действия.
3. Динамическое уклонение от них за счет тщательного распределения ресурсов.
4. Предотвращение за счет структурного подавления одного из четырех условий, необходимых для их возникновения

Страусиный алгоритм

- Самым простым подходом к решению проблемы является «страусиный алгоритм»: спрячьте голову в песок и сделайте вид, что проблема отсутствует.
- Если взаимоблокировка возникает в среднем один раз в пять лет, а система раз в неделю сбоят из-за технических отказов и дефектов операционной системы, большинство инженеров не захотят платить за избавление от взаимоблокировок существенным снижением производительности или удобства использования.

Обнаружение взаимоблокировок и восстановление работоспособности

- Обнаружение взаимоблокировок происходит с помощью поиска цикла в графе или другими алгоритмами.
- Восстановление работоспособности может происходить одним из следующих способов:
 - Восстановление за счет приоритетного владения ресурсом
 - Восстановление путем отката
 - Восстановление путем уничтожения процессов

Восстановление за счет приоритетного владения ресурсом

- Иногда можно временно отобрать ресурс у его текущего владельца и передать его другому процессу.
- В большинстве случаев для этого может понадобиться вмешательство оператора, особенно в операционных системах пакетной обработки, запускаемых на универсальных машинах.
 - К примеру, чтобы отобрать лазерный принтер у владельца, оператор может сложить все уже отпечатанные листы в стопку.
 - Затем процесс может быть приостановлен (помечен как неработоспособный).
 - После этого принтер может быть выделен другому процессу.
 - Когда этот процесс завершит свою работу, стопка отпечатанных листов бумаги может быть помещена обратно в приемный лоток принтера и работа исходного процесса может быть возобновлена.

Восстановление путем отката

- Этот подход к восстановлению процесса применяется наиболее часто.
- Но при реализации этого подхода должны быть решены две проблемы:
 - корректное освобождение ресурсов, захваченных заблокированным потоком;
 - восстановление информации на момент, предшествующий тупику.
- Для реализации этого подхода в программе устанавливаются такие точки, в которых запоминается состояние контекста потока.
- Эти точки называются **контрольными точками потока**.
- Изменение контекста потока между двумя контрольными точками называется **транзакцией**.

Восстановление путем отката

- Транзакция может быть зафиксирована или отменена.
- Отмена транзакции называется откатом на контрольную точку.
- После отката потоки разблокируются.
- Если транзакция зафиксирована, то откат на контрольную точку невозможен.

Восстановление путем отката

- При откате выполняются следующие действия:
 1. контекст потока восстанавливается в состояние, в котором он находился в контрольной точке;
 2. восстанавливаются данные на момент прохождения контрольной точки;
 3. освобождаются заблокированные ресурсы.

Восстановление путем уничтожения процессов

- Самым грубым, но и самым простым способом прервать взаимоблокировку является уничтожение одного или нескольких процессов.
- Можно уничтожить процесс, находящийся в цикле взаимоблокировки.
- Если повезет, то другие процессы смогут продолжить свою работу.
- Если это не поможет, то все можно повторить, пока цикл не будет разорван

Уклонение от взаимоблокировок

- Одним из способов уклонения от взаимоблокировок является алгоритм банкира.
- Данный алгоритм предложил Дейкстра.
- Моделью алгоритма является работа банкира по обеспечению клиентов кредитами.

Алгоритм банкира

- Задача состоит в следующем:
 - банкир имеет ограниченное количество денег для выдачи кредитов;
 - требуется обеспечить клиентов кредитами:
 - кредит может выдаваться по частям;
 - каждый кредит не превышает ресурсов банкира;
 - сумма кредитов может превышать ресурсы банкира.

Алгоритм банкира

- Для работы алгоритма определяются следующие состояния системы:
 - состояние считается **безопасным**, если система гарантирует невозможность возникновения тупика при распределении ресурсов;
 - иначе состояние называется **небезопасным**.
- Алгоритм банкира рассматривает запросы на ресурсы и проверяет, к какому состоянию системы приведет выделение ресурса.

Алгоритм банкира

Имеет

Max

A	0	6
B	0	5
C	0	4
D	0	7

Свободно: 10

a

Имеет

Max

A	1	6
B	1	5
C	2	4
D	4	7

Свободно: 2

b

Имеет

Max

A	1	6
B	2	5
C	2	4
D	4	7

Свободно: 1

c

Рис. 6.9. Состояния распределения ресурсов: *a* — безопасное; *b* — безопасное; *c* — небезопасное

Алгоритм банкира

- В последней ситуации, если все клиенты внезапно запросят максимальные ссуды, банкир не сможет удовлетворить никого из них и мы получим взаимоблокировку.
- Небезопасное состояние неизбежно приводит к взаимоблокировке, поскольку клиенту может и не понадобиться максимальная сумма кредита, но банкир не может рассчитывать на это.

Алгоритм банкира

- Алгоритм банкира рассматривает каждый запрос по мере поступления и проверяет, приведет ли его удовлетворение к безопасному состоянию.
 - Если да, то запрос удовлетворяется, в противном случае запрос откладывается до лучших времен.
- Чтобы понять, является ли состояние безопасным, банкир проверяет, может ли он предоставить достаточно ресурсов для удовлетворения запросов какого-нибудь клиента.
 - Если да, то эти ссуды считаются возвращенными, после чего проверяется следующий ближайший к пределу займа клиент и т. д.
 - Если в конечном счете все ссуды могут быть погашены, состояние является безопасным и исходный запрос можно удовлетворить

Предотвращение взаимоблокировки

- Для того чтобы избежать возникновения взаимоблокировок, можно использовать следующие стратегии при распределении ресурсов:
 - Захват всех ресурсов;
 - Откат в случае отказа;
 - Упорядочивание запросов

Захват всех ресурсов

- Процесс должен захватывать сразу все необходимые ему для работы ресурсы и только потом начинать свою работу.
- Недостаток подхода – неэффективное использование ресурсов компьютера, т. к. они блокируются для использования другими процессами

Откат в случае отказа

- Если в процессе работы потоку требуется дополнительный ресурс, но он получает отказ на захват этого ресурса, то процесс должен освободить все принадлежащие ему ресурсы.
- Недостаток подхода – затраты на откат процесса.

Упорядочивание запросов

- Типы (классы) ресурсов линейно упорядочиваются (нумеруются).
- В процессе своей работы процесс может захватить только те ресурсы, тип которых больше типа уже используемых им ресурсов.
- Недостаток подхода – непригоден, если потребность в ресурсах определяется динамически.

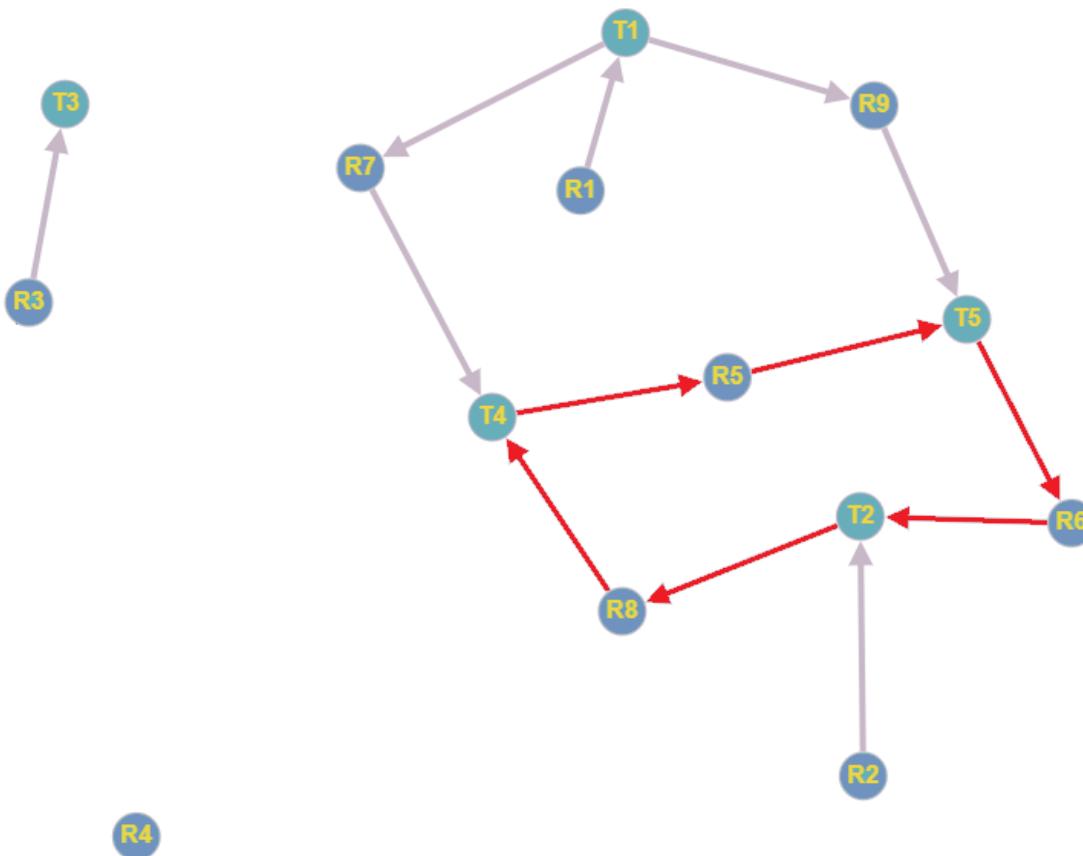
Задача №1

- Процесс включает пять потоков: $T_i, 1 \leq i \leq 5$,
- которые используют девять повторно используемых ресурсов: $R_j, 1 \leq j \leq 9$.

Задача №1

- Потоки процесса выполнили следующие запросы на распределение ресурсов:
 - поток T1:
 - захватил ресурс R1;
 - ждет ресурсы R7, R9;
 - поток T2:
 - захватил ресурсы R2, R6;
 - ждет ресурс R8;
 - поток T3:
 - захватил ресурс R3;
 - поток T4:
 - захватил ресурсы R7, R8;
 - ждет ресурс R5;
 - поток T5:
 - захватил ресурсы R5, R9;
 - ждет ресурс R6.
- Требуется определить, находится ли процесс в тупике.

Задача №1



Задача №2

- В системе работают n процессов, которые используют m ресурсов типа R .
- Пусть каждому процессу выделено P_i ресурсов и для каждого процесса известно M_i – максимальное количество ресурсов, которые он может использовать, $1 \leq i \leq n$.
- Вопрос 1. Используя алгоритм банкира, требуется определить, сколько ресурсов дополнительно можно выделить процессу j , где $n = 5, m = 20, j = 3, M_i = (5, 10, 5, 10, 3), P_i = (4, 7, 1, 5, 0)$.
- Вопрос 2. Процесс 3 запросил у системы 3 единицы ресурса R . Удовлетворит ли система этот запрос.

Задача №2

Процесс	Имеет	Максимум	Осталось
1	4	5	1
2	7	10	3
3	1	5	4
4	5	10	5
5	0	3	3

- После изначального выделения осталось $20 - 17 = 3$ свободных ресурса
- Для безопасного состояния требуется **1** ресурс для процесса №1.
- Соответственно, процессу №3 можно выделить $3 - 1 = 2$ ресурса.

Задача №2

- Так как система может выделить процессу 3 максимум 2 единицы ресурса, то процесс 3 получит отказ в выделении ресурсов.

8. Передача данных между процессами

8.1. Каналы передачи данных

© Pobegailo A.P. 2012

© Pobegailo A.P. 2012

Отправители и адресаты

- Под **обменом данными между параллельными процессами** понимается пересыпка данных от одного потока к другому потоку, предполагая, что эти потоки выполняются в контексте разных процессов.
- Поток, который посыпает данные другому потоку, называется **отправителем**.
- Поток, который получает данные от другого потока, называется **адресатом** или **получателем**.

Обмен данными между потоками одного процесса

- Если потоки выполняются в контексте **одного процесса**, то обмен данными между ними можно организовать, используя глобальные переменные и средства синхронизации потоков.

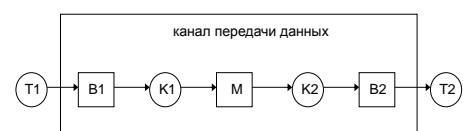
© Pobegailo A.P. 2012

© Pobegailo A.P. 2012

Обмен данными между потоками разных процессов

- Если потоки выполняются в контекстах разных процессов, то потоки не могут обращаться к общим переменным.
- В этом случае для обмена данными между процессами создается **канал передачи данных**, который является объектом ядра операционной системы и представляет собой область памяти, разделяемую несколькими процессами и используемую ими для обмена данными.

Схема канала передачи данных



T1, T2 – потоки пользователя

B1, B2 – буферы ввода-вывода

K1, K2 – потоки ядра операционной системы

M – общая память

© Pobegailo A.P. 2012

© Pobegailo A.P. 2012

Порядок работы канала передачи данных

- Пересылка данных из потока T1 в поток T2 происходит следующим образом:
 - Пользовательский поток T1 записывает данные в буфер B1, используя специальную функцию ядра операционной системы;
 - Поток K1 ядра операционной системы читает данные из буфера B1 и записывает их в общую память M;
 - Поток K2 ядра операционной системы читает данные из общей памяти M и записывает их в буфер B2;
 - Пользовательский поток T2 читает данные из буфера B2.

Реализация канала

- Обычно канал реализуется как кольцевой буфер, работающий по принципу **FIFO**.
- Для работы с каналом могут использоваться такие же функции ввода-вывода, как и для работы с файлами.

© Pobegailo A.P. 2012

© Pobegailo A.P. 2012

Способы передачи данных по каналам

- Различают два способа передачи данных по каналам:
 - потоком;
 - сообщениями.
- Если данные передаются непрерывной последовательностью байтов, то такая пересылка данных называется **передача данных потоком**.
- Если же данные пересыпаются группами байтов, то такая группа байтов называется **сообщением**, а сама пересылка данных называется **передачей данных сообщениями**.

8.2. Связи между процессами

- Прежде чем пересылать данные между процессами, нужно установить между этими процессами связь.
- Связь между процессами устанавливается как на **физическем** (аппаратном), так и **логическом** (программном) уровнях.

© Pobegailo A.P. 2012

© Pobegailo A.P. 2012

Направления передачи данных

- С точки зрения направления передачи данных различают следующие виды связей:
 - **полудуплексная связь** – данные по этой связи могут передаваться только в одном направлении;
 - **дуплексная связь** – данные по этой связи могут передаваться в обоих направлениях.

Топологии связей

- Теперь, предполагая, что рассматриваются только полудуплексные связи, определим возможные топологии связей.
- Под **топологией связи** будем понимать конфигурацию связей между процессами отправителями и получателями.

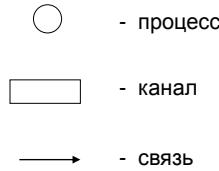
© Pobegailo A.P. 2012

© Pobegailo A.P. 2012

Виды связей

- С точки зрения топологии различают следующие виды связей:
 - $1 \rightarrow 1$ - между собой связаны только два процесса;
 - $1 \rightarrow N$ - один процесс связан с N процессами;
 - $N \rightarrow 1$ - каждый из N процессов связан с одним процессом;
 - $N \rightarrow M$ - каждый из N процессов связан с каждым из M процессов.

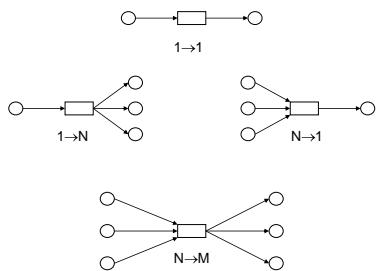
Обозначения



© Pobegailo A.P. 2012

© Pobegailo A.P. 2012

Топология связей между процессами



© Pobegailo A.P. 2012

© Pobegailo A.P. 2012

Порядок разработки систем обмена данными

- При разработке систем с обменом данными между процессами, прежде всего, должна быть выбрана топология связей и направления передачи данных по этим связям.
- После этого в программах реализуются выбранные связи между процессами, используя функции операционной системы, предназначенные для установки связи между процессами.

Порты и почтовые ящики

- Канал передачи данных, реализующий топологию $N \rightarrow 1$ обычно называется **портом**.
- Канал передачи данных, реализующий топологию $N \rightarrow M$ обычно называется **почтовым ящиком**.

Функции для установки связей между процессами

- Для установки связей между процессами обычно используются функции типа:
 - connect** – установить связь;
 - disconnect** – разорвать связь.
- Эти функции, а также функции для обмена данными между процессами обеспечивает система передачи данных, которая обычно является частью ядра операционной системы.

© Pobegailo A.P. 2012

© Pobegailo A.P. 2012

8.3. Передача сообщений

- Обмен сообщениями между процессами выполняется при помощи двух функций:
 - send** - послать сообщение;
 - receive** - получить сообщение.

Структура сообщения

Заголовок сообщения	Тип сообщения
	Идентификатор получателя
	Идентификатор отправителя
	Длина сообщения
	Управляющая информация
	Содержание сообщения
Тело сообщения	

© Pobegailo A.P. 2012

© Pobegailo A.P. 2012

Типы адресации процессов

- При передаче сообщений может использоваться прямая или косвенная адресация процессов.

Прямая адресация процессов

- При **прямой адресации** процессов в функциях *send* и *receive* явно указываются процессы отправитель и получатель.
- В этом случае функции обмена данными имеют следующий вид:
 - send(P, сообщение)* - послать сообщение процессу P;
 - receive(Q, сообщение)* - получить сообщение от процесса Q.

© Pobegailo A.P. 2012

© Pobegailo A.P. 2012

Косвенная адресация процессов

- При **косвенной адресации** в функциях *send* и *receive* указываются не адреса процессов, а имя канала связи, по которому передается сообщение.
- В этом случае функции обмена данными имеют следующий вид:
 - send(S, сообщение)* - послать сообщение по каналу связи S;
 - receive(R, сообщение)* - получить сообщение по каналу связи R.
- В последнем случае сообщение могут получать все процессы, подключенные к каналу связи R.

Симметричная и асимметричная адресация

- Адресация процессов может быть симметричной и асимметричной.
- Если при передаче сообщений используется только прямая или только косвенная адресация, то такая адресация процессов называется **симметричной**.
- В противном случае адресация процессов называется **асимметричной**.

© Pobegailo A.P. 2012

© Pobegailo A.P. 2012

Адресация в системах клиент-сервер

- Асимметричная адресация процессов используется в системах клиент-сервер.
- В этом случае клиенты знают адрес сервера и посыпают ему сообщения, используя функцию
 - `send(Server, сообщение)`
- А сервер «слушает» канал связи и принимает сообщения от всех клиентов, используя функцию
 - `receive(Connection, сообщение)`
- Часто эта функция так и называется `listen` (слушать).

Протокол

- Набор правил, по которым устанавливаются связи и передаются данные между процессами, называется **протоколом**.

© Pobegailo A.P. 2012

© Pobegailo A.P. 2012

8.4. Синхронный и асинхронный обмен данными

- При передаче данных различают синхронный и асинхронный обмен данными.

Синхронное и асинхронное отправление сообщения

- Если поток отправитель, отправив сообщение функцией `send`, блокируется до получения этого сообщения потоком адресатом, то такое *отправление сообщения* называется **синхронным**.
- В противном случае *отправление сообщения* называется **асинхронным**.

© Pobegailo A.P. 2012

© Pobegailo A.P. 2012

Синхронное и асинхронное получение сообщения

- Если поток получатель, вызвавший функцию `receive`, блокируется до тех пор, пока не получит сообщение, то такое *получение сообщения* называется **синхронным**.
- В противном случае *получение сообщения* называется **асинхронным**.

Синхронный и асинхронный обмен сообщениями

- Обмен сообщениями называется **синхронным**, если поток отправитель синхронно передает сообщения, а поток адресат синхронно принимает эти сообщения.
- В противном случае *обмен сообщениями* называется **асинхронным**.
- Пересылка данных потоком всегда происходит синхронным образом, так как в этом случае между отправителем и получателем устанавливается непосредственная связь.

© Pobegailo A.P. 2012

© Pobegailo A.P. 2012

Рандеву

- Синхронный обмен данными в случае прямой адресации процессов называется **рандеву** (rendezvous), что переводится с французского языка как «встреча».
- Такой механизм обмена сообщениями используется в языке программирования Ада.

© Pobegailo A.P. 2012

8.5. Буферизация

- Буфером** называется *вместимость связи между процессами*, то есть количество сообщений, которые могут одновременно пересыпаться по этой связи.

© Pobegailo A.P. 2012

Типы буферизации

- Существенно различаются три типа буферизации:
 - нулевая вместимость связи* (нет буфера), в этом случае возможен только синхронный обмен данными между процессами;
 - ограниченная вместимость связи* (ограниченный буфер), в этом случае, если буфер полон, то отправитель сообщения должен ждать очистки буфера хотя бы от одного сообщения;
 - неограниченная вместимость связи* (неограниченный буфер), в этом случае отправитель никогда не ждет при отправке сообщения.

© Pobegailo A.P. 2012

- Как видно из этих определений типы буферизации тесно связаны с синхронизацией передачи данных и поэтому также должны учитываться при разработке систем, которые используют обмен данными между процессами.

© Pobegailo A.P. 2012

8.6. Анонимные каналы в Windows

- Анонимным каналом** называется объект ядра операционной системы, который обеспечивает передачу данных между процессами, выполняющимися на одном компьютере.
- Процесс, который создает анонимный канал, называется **сервером анонимного канала**.
- Процессы, которые связываются с анонимным каналом, называются **клиентами анонимного канала**.

© Pobegailo A.P. 2012

Свойства анонимных каналов

- не имеют имени;
- полудуплексные;
- передача данных потоком;
- синхронный обмен данными;
- возможность моделирования любой топологии связей.

© Pobegailo A.P. 2012

Порядок работы с анонимным каналом

- создание анонимного канала сервером;
- соединение клиентов с каналом;
- обмен данными по каналу;
- закрытие канала.

Соединение клиентов с анонимным каналом

- Так как анонимный канал не имеет имени, то доступ к такому каналу имеют только родительский процесс-сервер и дочерние процессы-клиенты этого канала.
- Чтобы процесс-клиент наследовал дескриптор анонимного канала, этот дескриптор должен быть наследуемым.
- Явная передача наследуемого дескриптора процессу-клиенту анонимного канала может выполняться одним из следующих способов:
 - через командную строку;
 - через поля *hStdInput*, *hStdOutput* и *hStdError* структуры *STARTUPINFO*;
 - посредством сообщения *WM_COPYDATA*;
 - через файл.

© Pobegailo A.P. 2012

© Pobegailo A.P. 2012

Функции для работы с анонимным каналом

- *CreatePipe* – создание анонимного канала;
- *WriteFile* – запись данных в анонимный канал;
- *ReadFile* – чтение данных из анонимного канала;

8.7. Именованные каналы в Windows

- **Именованным каналом** называется объект ядра операционной системы, который обеспечивает передачу данных между процессами, выполняющимися на компьютерах в одной локальной сети.
- Процесс, который создает именованный канал, называется **сервером именованного канала**.
- Процессы, которые связываются с именованным каналом, называются **клиентами именованного канала**.

© Pobegailo A.P. 2012

© Pobegailo A.P. 2012

Свойства именованных каналов

- имеют имя, которое используется клиентами для связи с именованным каналом;
- могут быть как полудуплексные, так и дуплексные;
- передача данных может осуществляться как потоком, так и сообщениями;
- обмен данными может быть как синхронным, так и асинхронным;
- возможность моделирования любой топологии связей.

Порядок работы с именованными каналами

- создание именованного канала сервером;
- соединение сервера с экземпляром именованного канала;
- соединение клиента с экземпляром именованного канала;
- обмен данными по именованному каналу;
- отсоединение сервера от экземпляра именованного канала;
- закрытие именованного канала клиентом и сервером.

© Pobegailo A.P. 2012

© Pobegailo A.P. 2012

Функции для соединения с именованным каналом

- *CreateNamedPipe* – создание именованного канала;
- *ConnectNamedPipe* – соединение сервера с клиентом именованного канала;
- *DisconnectNamedPipe* – отсоединение сервера от именованного канала;
- *WaitNamedPipe* – ожидание клиентом свободного экземпляра именованного канала;
- *CreateFile* – соединение клиента с именованным каналом;

Функции для передачи данных по именованному каналу

- *WriteFile* – запись данных в именованный канал;
- *ReadFile* – чтение данных из именованного канала;
- *PeekNamedPipe* – копирование данных из именованного канала;
- *TransactNamedPipe* – обмен сообщениями по именованному каналу;

© Pobegailo A.P. 2012

© Pobegailo A.P. 2012

Функции для работы с состоянием и свойствами именованного канала

- *GetNamedPipeHandleState* – определение состояния именованного канала;
- *SetNamedPipeHandleState* – изменение состояния именованного канала;
- *GetNamedPipeInfo* – получить информацию об атрибутах именованного канала;

© Pobegailo A.P. 2012

8.8. Почтовые ящики в Windows

- **Почтовым ящиком** называется объект ядра операционной системы, который обеспечивает передачу сообщений от процессов-клиентов к процессам-серверам, выполняющимся на компьютерах в пределах локальной сети.
- Процесс, который создает почтовый ящик, называется **сервером почтового ящика**.
- Процессы, которые связываются с почтовым ящиком, называются **клиентами почтового ящика**.

© Pobegailo A.P. 2012

Свойства почтовых ящиков

- имеют имя, которое используется клиентами для связи с почтовыми ящиками;
- направление передачи данных от клиента к серверу;
- передача данных осуществляется сообщениями;
- обмен данными может быть как синхронным, так и асинхронным.

© Pobegailo A.P. 2012

Передача сообщений почтовыми ящиками

- Хотя передача данных осуществляется только от клиента к серверу, один почтовый ящик может иметь несколько серверов.
- Это происходит в том случае, если несколько серверов создают почтовые ящики с одинаковыми именами.
- Тогда все сообщения, которые посыпает клиент в такой почтовый ящик, будут получать все серверы этого почтового ящика.
- Таким образом, можно сказать, что почтовые ящики обеспечивают одностороннюю связь типа "многие ко многим".
- При этом доставка сообщения от клиента к серверам почтового ящика не подтверждается системой.

© Pobegailo A.P. 2012

Порядок работы с почтовым ящиком

- создание почтового ящика сервером;
- соединение клиента с почтовым ящиком;
- обмен данными через почтовый ящик;
- закрытие почтового ящика клиентом и сервером.

Функции для работы с почтовыми ящиками

- *CreateMailslot* – создание почтового ящика;
- *CreateFile* – соединение клиента с почтовым ящиком;
- *WriteFile* – запись данных в почтовый ящик;
- *ReadFile* – чтение данных из почтового ящика;
- *GetMailslotInfo* – получение информации о свойствах почтового ящика;
- *SetMailslotInfo* – изменение сервером времени ожидания от клиента.

10. Виртуальная память

10.1. Концепция виртуальной памяти

- Интегральные схемы, предназначенные для хранения программ и данных, называются **физическими памятью**.
- Обычно, под физической памятью понимается память, к которой процессор может обращаться, используя адресную шину и шину данных, а внутренняя память самого процессора представляется регистрами.
- Каждый байт физической памяти имеет свой номер или индекс, который называется **физическими адресом**.
- При обращении к физической памяти процессор должен выставить на адресную шину физический адрес памяти, к которой он хочет получить доступ.

© Pobegailo A.P. 2012

© Pobegailo A.P. 2012

Логическая память процесса

- Под **логической памятью процесса** понимается массив байтов, к которым может обратиться процесс.
- Индекс каждого элемента этого массива называется **логическим адресом**.
- Так как логическая память процесса представляется линейным массивом байт, то логический адрес процесса обычно называют **линейным адресом**.

© Pobegailo A.P. 2012

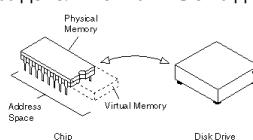
Проблема отображения логической памяти в физическую

- Так как в действительности процесс может работать только с данными в физической памяти, то во время работы процесса необходимо отображать логическую память процесса в физическую память компьютера.
- Обычно, прямое отображение невозможно, по той простой причине, что объем логической памяти процесса превышает объем физической памяти компьютера.

© Pobegailo A.P. 2012

Виртуальная память

- Для решения этой задачи физическую память компьютера дополняют памятью на дисках.



- Полученную расширенную память называют **виртуальной памятью**, а адрес элемента этой памяти называют **виртуальным адресом**.

© Pobegailo A.P. 2012

Виртуальная память

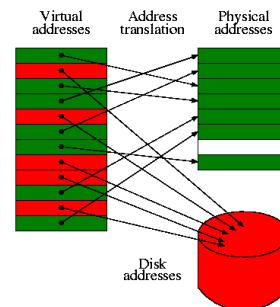
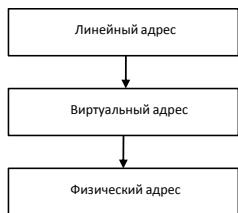


Схема преобразование адресов при работе процесса



© Pobegailo A.P. 2012

Программная и аппаратная поддержка схемы преобразования адресов

- Преобразование линейного адреса процесса в виртуальный адрес выполняется **операционной системой** посредством настройки регистров микропроцессора.
- Обычно, линейный адрес процесса отличается от виртуального адреса только интерпретацией бит в этом адресе.
- Преобразование виртуального адреса в физический выполняется **аппаратным обеспечением**, а именно, микропроцессором.

© Pobegailo A.P. 2012

10.2. Организация виртуальной памяти

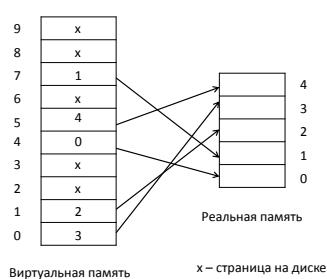
- Для реализации преобразования виртуального адреса в физический адрес поступают следующим образом:
 - виртуальную память разбивают на блоки одинаковой длины, обычно равной 4 Кб, которые называются **страницами**;
 - при обращении процесса по виртуальному адресу процессор проверяет, находится ли виртуальная страница с этим адресом в реальной памяти;
 - если нет, то происходит загрузка этой виртуальной страницы в реальную память компьютера и настройка адресного пространства процесса на работу с этой страницей.

© Pobegailo A.P. 2012

- Такая организация виртуальной памяти называется **страничной**.
- Файлы, в которых хранятся страницы виртуальной памяти, называются **файлами страниц** или **файлами подкачки** (swap files).

© Pobegailo A.P. 2012

Отображение виртуальных страниц на реальные страницы



© Pobegailo A.P. 2012

Форматы адресов

- а) формат реального адреса:
- | | |
|---|---|
| r | d |
|---|---|
- r – номер реальной страницы
- d – смещение в реальной странице
- б) формат виртуального адреса:
- | | |
|---|---|
| v | d |
|---|---|
- v – номер виртуальной страницы
- d – смещение в виртуальной странице

© Pobegailo A.P. 2012

- Например, если виртуальный адрес имеет длину 32 бита, а длина страницы равна 4 Кб, то младшие 12 бит рассматриваются как смещение внутри страницы, старшие 20 бит – как номер виртуальной страницы.
- Это разбиение обусловлено тем, что $2^{12} = 4096$ байт, т. е. 4 Кб.
- Единственное различие между виртуальным и реальным адресами состоит в том, что виртуальный адрес может иметь большую длину поля, отведенного на номер виртуальной страницы.

© Pobegailo A.P. 2012

Преобразование виртуальных адресов в реальные адреса

- Для преобразования виртуальных адресов в реальные адреса в системной области физической памяти для каждого процесса хранится таблица страниц, строки которой имеют следующую структуру.

© Pobegailo A.P. 2012

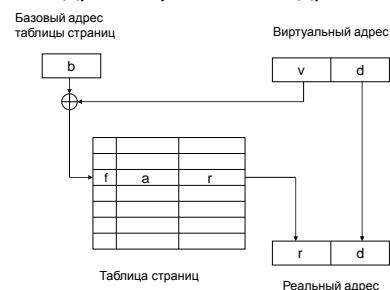
Формат строки таблицы страниц процесса



- f – флаг, отмечающий нахождение виртуальной страницы в реальной памяти;
a – адрес виртуальной страницы во внешней памяти;
r – адрес реальной страницы в физической памяти.

© Pobegailo A.P. 2012

Схема преобразования виртуального адреса в реальный адрес



© Pobegailo A.P. 2012

Алгоритм отображения виртуальной памяти в реальную память (выполняется аппаратно – микропроцессором)

- Найти в таблице страниц строку, соответствующую номеру виртуальной страницы. Индекс этой строки равен $b + v$.
- Если $f = 1$, то виртуальная страница находится в реальной памяти.
- Если $f = 0$, то виртуальная страница находится на диске. В этом случае выполнить следующие действия:
 - Загрузить виртуальную страницу в реальную память.
 - Установить в столбце г адрес виртуальной страницы в реальной памяти.
 - Установить в столбце f значение 1.
- Вычислить реальный адрес, который формируется из адреса реальной страницы, который задан значением г, и смещения в виртуальной странице, которое задано значением д.

© Pobegailo A.P. 2012

10.3. Алгоритмы замещения страниц

- При подкачке виртуальной страницы в физическую память может оказаться, что все страницы физической памяти уже заняты другими виртуальными страницами.
- В этом случае одна из физических страниц выталкивается из физической памяти на диск, а на её место с диска загружается требуемая виртуальная страница.

© Pobegailo A.P. 2012

Проблема замещения страницы виртуальной памяти

- В этом случае возникает следующая проблема: какую виртуальную страницу вытолкнуть из физической памяти на диск.
- Для определения такой страницы чаще всего используются следующие алгоритмы.

1. Алгоритм FIFO (first in – first out)

- На диск выталкивается первая из загруженных в реальную память виртуальных страниц.

© Pobegailo A.P. 2012

© Pobegailo A.P. 2012

2. Алгоритм LRU (least recently used)

- На диск выталкивается виртуальная страница, которая дольше всего не использовалась.

3. Алгоритм NRU (not recently used)

- На диск выталкивается страница, которая не использовалась в заданный интервал времени.
- Если таких страниц несколько, то выбирается случайная.

© Pobegailo A.P. 2012

© Pobegailo A.P. 2012

4. Алгоритм LFU (least frequently used)

- На диск выталкивается виртуальная страница, которая меньше всего использовалась.

Пробуксовка страниц

- Все эти алгоритмы имеют свои преимущества и недостатки и для каждого из них можно подобрать такой случай использования виртуальных страниц, при котором система начнет пробуксовывать.
- То есть при каждом новом обращении к памяти будет необходима подкачка новой виртуальной страницы в реальную память.

© Pobegailo A.P. 2012

© Pobegailo A.P. 2012

Проблема оптимизации замещения страниц

- Оптимального алгоритма подкачки виртуальных страниц не существует, так поведение приложений непредсказуемо.
- Если загрузкой виртуальных страниц управляет сама программа, все шаги выполнения которой заранее известны, то, очевидно, что возможен оптимальный порядок загрузки и выгрузки виртуальных страниц, который минимизирует обращения к диску.

© Pobegailo A.P. 2012

Затирание виртуальных страниц

- Так как запись виртуальной страницы на диск довольно медленная операция, то, обычно, элемент таблицы страниц содержит также флаг, отмечающий была ли произведена **запись** на виртуальную страницу, находящуюся в реальной памяти.
- Если записи не было, то эта виртуальная страница просто **затирается** при подкачке с диска на её место другой виртуальной страницы.

© Pobegailo A.P. 2012

10.4. Рабочее множество процесса

Свойство локальности при работе процесса

- Эмпирически было определено, что при работе многих программ наблюдается свойство локальности.
- То есть выполняемый в какой-то интервал времени код программы и используемая программой память расположены локально, а не разбросаны по всей программе.
- Обычно, программисты пишут свои программы, также следя этому правилу.

© Pobegailo A.P. 2012

© Pobegailo A.P. 2012

Рабочее множество страниц процесса

- Поэтому для эффективной работы программы необходимо, чтобы какое-то множество, часто используемых на данном интервале времени виртуальных страниц, находилось в реальной памяти.
- Это множество виртуальных страниц называется **рабочим множеством** процесса.

© Pobegailo A.P. 2012

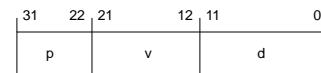
- Естественно, что рабочее множество страниц процесса изменяется со временем, какие-то страницы начинают использоваться реже, а какие-то – чаще.
- Но для каждого процесса можно определить размеры рабочего множества страниц таким образом, что на каждом интервале времени все часто используемые страницы будут находиться в реальной памяти.

© Pobegailo A.P. 2012

10.5. Организация виртуальной памяти в Windows

- В операционных системах Windows используется страничная организация виртуальной памяти.
- При этом линейный адрес процесса совпадает с его виртуальным адресом, который имеет следующий формат:

Формат виртуального адреса в Windows



p – смещение в каталоге страниц

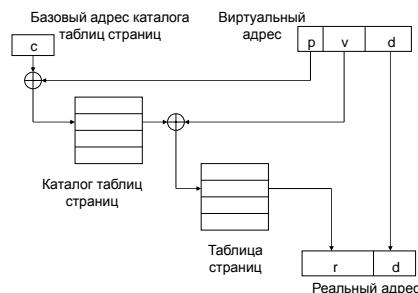
v – номер виртуальной страницы

d – смещение в виртуальной странице

© Pobegailo A.P. 2012

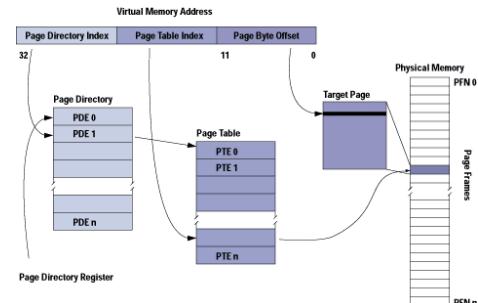
© Pobegailo A.P. 2012

Схема преобразования виртуального адреса в реальный



© Pobegailo A.P. 2012

Схема преобразования виртуального адреса в реальный



Каталог таблиц страниц

- В Windows адрес таблицы страниц процесса хранится в специальной таблице, которая называется **каталогом таблиц страниц**.
- Каждая строка каталога таблиц страниц содержит адрес уникальной таблицы страниц.
- В свою очередь поле p виртуального адреса имеет уникальное значение для каждого процесса.
- Поэтому каждый процесс использует единственную таблицу страниц.
- Отсюда следует, что адресные пространства разных процессов изолированы друг от друга.

Формат строки таблицы страниц

31	27	26	7	6	3	2	0
p		a	f	s			

s – состояние страницы

f – номер файла подкачки страницы

a – физический адрес страницы

p – атрибуты доступа к странице

© Pobegailo A.P. 2012

© Pobegailo A.P. 2012

Состояния виртуальной страницы

- Поле **s** описывает состояние виртуальной страницы, которое определяется комбинацией трех бит.
- Различные комбинации этих бит определяют следующие состояния виртуальной страницы:
 - страницы нет в реальной памяти (invalid page);
 - страница находится в реальной памяти (valid page);
 - страница находится в реальной памяти и была модифицирована (valid dirty page);
 - страница загружается в реальную память (invalid page in transition);
 - страница сохраняется на диск (invalid dirty page in transition).

Файлы подкачки

- Поле **f** задает номер файла подкачки на диске.
- Учитывая длину этого поля, можно определить до 16 файлов подкачки виртуальных страниц.

© Pobegailo A.P. 2012

© Pobegailo A.P. 2012

Физический адрес страницы

- Поле **a** содержит физический адрес страницы в реальной памяти, при условии, что страница загружена с диска.
- Учитывая длину этого поля, можно определить, что виртуальная память процесса может содержать 220 виртуальных страниц.
- Отсюда следует, что так как длина виртуальной страницы равна 4Кб, то вся виртуальная память процесса составляет $220 * 4 \text{ Кб} = 4 \text{ Гб}$ виртуальной памяти.

© Pobegailo A.P. 2012

Атрибуты доступа к странице

- Поле **p** содержит атрибуты доступа к виртуальной странице, которые могут принимать следующие значения:
 - PAGE_NOACCESS – доступ к странице запрещен;
 - PAGE_READONLY – доступно только чтение страницы;
 - PAGE_READWRITE – доступны чтение и запись страницы.

© Pobegailo A.P. 2012

10.6. Менеджер виртуальной памяти в Windows

- В Windows управлением виртуальной памяти занимается специальный процесс, который называется менеджером виртуальной памяти (Virtual Memory Manager – VMM).
- Менеджер виртуальной памяти поддерживает свое внутреннее описание состояния каждой страницы в реальной памяти.

© Pobegailo A.P. 2012

Состояния страниц реальной памяти

- В соответствии с этим описанием каждая страница реальной памяти может находиться в одном из следующих состояний:
 - страница в рабочем состоянии и используется процессом (valid);
 - страница записывается на диск (modified);
 - страница удаляется из рабочего множества страниц процесса (standby);
 - страница освобождена процессом, но не заполнена нулями (free);
 - страница заполнена нулями и может использоваться любым процессом (zeroed);
 - страница в нерабочем состоянии (bad).

© Pobegailo A.P. 2012

Управление рабочим множеством страниц процесса и алгоритм подкачки страниц

- Для каждого процесса операционная система Windows определяется рабочее множество страниц этого процесса.
- Во время работы процесса менеджер виртуальной памяти периодически проверяет частоту использования страниц из рабочего множества процесса.
- Если некоторая виртуальная страница используется редко, то она удаляется из рабочего множества процесса.
- При замещении страниц Windows использует алгоритм LRU, но только с тем отличием, что он применяется не для всех виртуальных страниц, находящихся в реальной памяти, а отдельно для рабочего множества страниц каждого процесса.

© Pobegailo A.P. 2012

10.7. Состояние виртуальной памяти процесса

© Pobegailo A.P. 2012

Объем виртуальной памяти процесса

- В операционных Windows виртуальный адрес процесса отличается от линейного адреса этого же процесса только интерпретацией бит линейного адреса.
- Поэтому можно сказать, что каждому процессу в Windows также доступно два гигабайта виртуальной памяти.

© Pobegailo A.P. 2012

- Это не значит, что процесс может использовать всю эту память одновременно.
- Количество виртуальной памяти, доступной процессу, зависит от емкости физической памяти и дисков.
- Чтобы ограничить процесс в использовании виртуальной памяти, некоторые страницы в таблице страниц могут быть помечены просто как недоступные.

© Pobegailo A.P. 2012

Состояния виртуальных страниц

- С точки зрения процесса страницы его виртуальной памяти могут находиться в одном из трех состояний:
 - **свободны** для использования (free);
 - **зарезервированы**, но не используются процессом (reserved);
 - **распределены** процессу для использования (committed).

© Pobegailo A.P. 2012

Распределение страниц виртуальной памяти процессом

- Первоначально при запуске процесса все страницы виртуальной памяти считаются свободными (free), естественно кроме тех, в которые загружена сама программа.
- Чтобы распределить для использования свободные или зарезервированные страницы виртуальной памяти, процесс должен вызвать функцию *VirtualAlloc*.
- Только после успешного завершения этой функции процесс может использовать распределенную ему виртуальную память.

© Pobegailo A.P. 2012

Резервирование страниц виртуальной памяти

- Третье состояния характеризует виртуальные страницы как зарезервированные (reserved).
- Это значит, что эти виртуальные страницы зарезервированы процессом для дальнейшего использования и не будут выделяться системой для использования процессу без точного указания процессом их адреса.
- При резервировании виртуальных страниц реальная память под эти страницы не выделяется.

© Pobegailo A.P. 2012

10.8. Работа с виртуальной памятью в Windows

© Pobegailo A.P. 2012

Функции для работы с виртуальной памятью

- VirtualAlloc* – резервирование или распределение области виртуальной памяти;
- VirtualFree* – освобождение области виртуальной памяти после завершения работы;
- VirtualLock* – блокировать виртуальные страницы в реальной памяти;
- VirtualUnlock* – отмена блокировки виртуальных страниц в реальной памяти;
- VirtualProtect* – изменение атрибутов доступа к области виртуальной памяти;
- VirtualQuery* – определение состояния области виртуальной памяти процесса;

© Pobegailo A.P. 2012

Функции для работы с рабочим множеством страниц процесса

- GetProcessWorkingSetSize* – определение количества страниц, которые входят в рабочее множество процесса;
- SetProcessWorkingSetSize* – изменение минимального и максимального размера рабочего множества страниц процесса;

© Pobegailo A.P. 2012

Вспомогательные функции

- CopyMemory* – копирование блока виртуальной памяти; результат выполнения функции непредсказуем, если исходный и результирующий блоки памяти перекрываются;
- MoveMemory* – перемещение блока виртуальной памяти, блоки могут перекрываться;
- FillMemory* – заполнение блока виртуальной памяти определенным значением;
- ZeroMemory* – заполнение блока виртуальной памяти нулями.

© Pobegailo A.P. 2012

Управление файлами

Файл

- Файл является механизмом абстрагирования.
- Он предоставляет способ сохранения информации на диске и последующего ее считывания, который должен оградить пользователя от подробностей о способе и месте хранения информации и деталей фактической работы дисковых устройств.

Имена файлов

- Конкретные правила составления имен файлов варьируются от системы к системе, но все ныне существующие операционные системы в качестве допустимых имен файлов позволяют использовать от одной до восьми букв.
- Многие файловые системы поддерживают имена длиной до 255 символов.
- Некоторые файловые системы различают буквы верхнего и нижнего регистров, а некоторые не делают таких различий.
 - Система UNIX подпадает под первую категорию, а старая MS-DOS — под вторую.

Немного о Windows

- Windows 95 и Windows 98 использовали файловую систему MS-DOS под названием **FAT-16**, и поэтому они унаследовали множество ее свойств, касающихся, например, построения имен файлов.
- В Windows 98 было представлено расширение **FAT-16**, которое привело к системе **FAT-32**, но обе эти системы очень похожи друг на друга.
- В добавок к этому Windows NT, Windows 2000, Windows XP, Windows Vista, Windows 7 и Windows 8 по-прежнему поддерживают обе файловые системы FAT, которые к настоящему времени фактически уже устарели.

Немного о Windows

- Но новые операционные системы имеют собственную намного более совершенную файловую систему **NTFS**, которая обладает несколько иными свойствами (к примеру, допускает имена файлов в кодировке Unicode).
- На самом деле для Windows 8 имеется вторая файловая система, известная как **ReFS** (или Resilient File System — восстанавливаемая файловая система), но она предназначена для серверной версии.
- Кроме того, существует также новая FAT-подобная файловая система, известная как **exFAT**. Это созданное компанией Microsoft расширение к FAT-32, оптимизированное для флеш-накопителей и больших файловых систем.

Расширение имен файлов

- Многие операционные системы поддерживают имена файлов, состоящие из двух частей, разделенных точкой, как, например, prog.c.
- Та часть имени, которая следует за точкой, называется расширением имени файла и, как правило, несет в себе некоторую информацию о файле.
- К примеру, в MS-DOS имена файлов состоят из 1–8 символов и имеют (необязательно) расширение, состоящее из 1–3 символов.
- В UNIX количество расширений выбирает сам пользователь, так что имя файла может иметь два и более расширений, например homepage.html.zip, где .html указывает на наличие веб-страницы в коде HTML, а .zip — на то, что этот файл (homepage.html) был сжат архиватором.

Расширение имен файлов

Расширение	Значение
.bak	Резервная копия файла
.c	Исходный текст программы на языке С
.gif	Изображение формата GIF
.hlp	Файл справки
.html	Документ в формате HTML
.jpg	Статическое растровое изображение в формате JPEG
.mp3	Музыка в аудиоформате MPEG layer 3
.mpg	Фильм в формате MPEG
.o	Объектный файл (полученный на выходе компилятора, но еще не прошедший компоновку)
.pdf	Документ формата PDF
.ps	Документ формата PostScript
.tex	Входной файл для программы форматирования TEX
.txt	Обычный текстовый файл
.zip	Архив, сжатый программой zip

Расширение имен файлов

- В некоторых системах (например, во всех разновидностях UNIX) расширения имен файлов используются в соответствии с соглашениями и не навязываются операционной системой.
 - Файл file.txt может быть текстовым файлом, но это скорее напоминание его владельцу, чем передача некой значимой информации компьютеру.
- В то же время компилятор языка С может выдвигать требование, чтобы компилируемые им файлы имели расширение .c, и отказываться выполнять компиляцию, если они не имеют такого расширения.

Расширение имен файлов

- Система Windows, напротив, знает о расширениях имен файлов и присваивает каждому расширению вполне определенное значение.
- Пользователи (или процессы) могут регистрировать расширения в операционной системе, указывая программу, которая станет их «владельцем». При двойном щелчке мыши на имени файла запускается программа, назначенная этому расширению, с именем файла в качестве параметра.
 - Например, двойной щелчок мыши на имени file.docx запускает Microsoft Word, который открывает файл file.docx в качестве исходного файла для редактирования

Структура файла

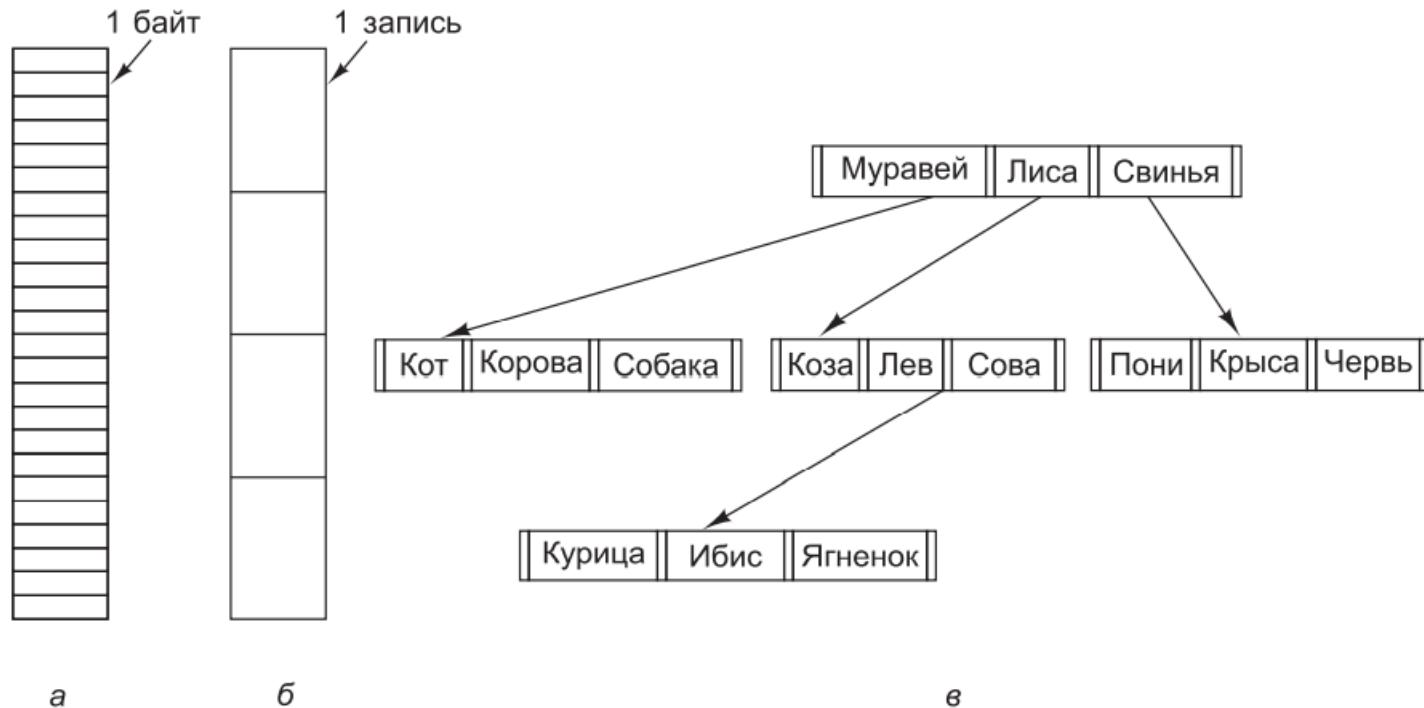


Рис. 4.1. Три типа файлов: *а* — последовательность байтов; *б* — последовательность записей;
в — дерево

Структура файла – последовательность байтов

- Когда операционная система считает, что файлы – это не более чем последовательность байтов, она предоставляет максимум гибкости.
- Программы пользователя могут помещать в свои файлы все, что им заблагорассудится, и называть их, как им удобно. Операционная система ничем при этом не помогает, но и ничем не мешает.
- Эта файловая модель используется всеми версиями UNIX (включая Linux и OS X) и Windows.

Структура файла – последовательность записей

- Основная идея файла как последовательности записей состоит в том, что операция чтения возвращает одну из записей, а операция записи перезаписывает или дополняет одну из записей.
- Ранее многие операционные системы в основе своей файловой системы использовали файлы, состоящие из 80-символьных записей, — в сущности, образы **перфокарт**.
- Эти операционные системы поддерживали также файлы, состоящие из 132-символьных записей, предназначавшиеся для строковых принтеров (которые в то время представляли собой большие печатающие устройства, имеющие 132 столбца).
- Программы на входе читали блоки по 80 символов, а на выходе записывали блоки по 132 символа, даже если заключительные 52 символа были пробелами.
- Ни одна современная универсальная система больше не использует эту модель в качестве своей первичной файловой системы.

Структура файла – дерево

- При древовидной организации, файл состоит из дерева записей, необязательно одинаковой длины, каждая из которых в конкретной позиции содержит ключевое поле.
- Дерево сортируется по ключевому полю, позволяя выполнять ускоренный поиск по конкретному ключу.
- Здесь основной операцией является не получение «следующей» записи, хотя возможно проведение и этой операции, а получение записи с указанным ключом.
- Этот тип файла отличается от бессистемных битовых потоков, используемых в UNIX и Windows, и используется в некоторых больших универсальных компьютерах, применяемых при обработке коммерческих данных.

Типы файлов

- Многие операционные системы поддерживают несколько типов файлов.
- К примеру, в системах UNIX и Windows имеются обычные файлы и каталоги.
- В системе UNIX имеются также символьные и блочные специальные файлы.
- **Обычными** считаются файлы, содержащие информацию пользователя.
- **Каталоги** – это системные файлы, предназначенные для поддержки структуры файловой системы.
- **Символьные специальные файлы** имеют отношение к вводу-выводу и используются для моделирования последовательных устройств ввода-вывода, к которым относятся терминалы, принтеры и сети.
- **Блочные специальные файлы** используются для моделирования дисков.

Типы файлов

- Как правило, к обычным файлам относятся либо файлы ASCII, либо двоичные файлы.
- ASCII-файлы состоят из текстовых строк.
- В некоторых системах каждая строка завершается символом возврата каретки. В других системах используется символ перевода строки. Некоторые системы (например, Windows) используют оба символа.
- Строки не обязательно должны иметь одинаковую длину.

Типы файлов

- Все остальные файлы относятся к двоичным — это означает, что они не являются ASCII-файлами.
- Их распечатка будет непонятным и бесполезным набором символов.
- Обычно у них есть некая внутренняя структура, известная использующей их программе.

Двоичный файл

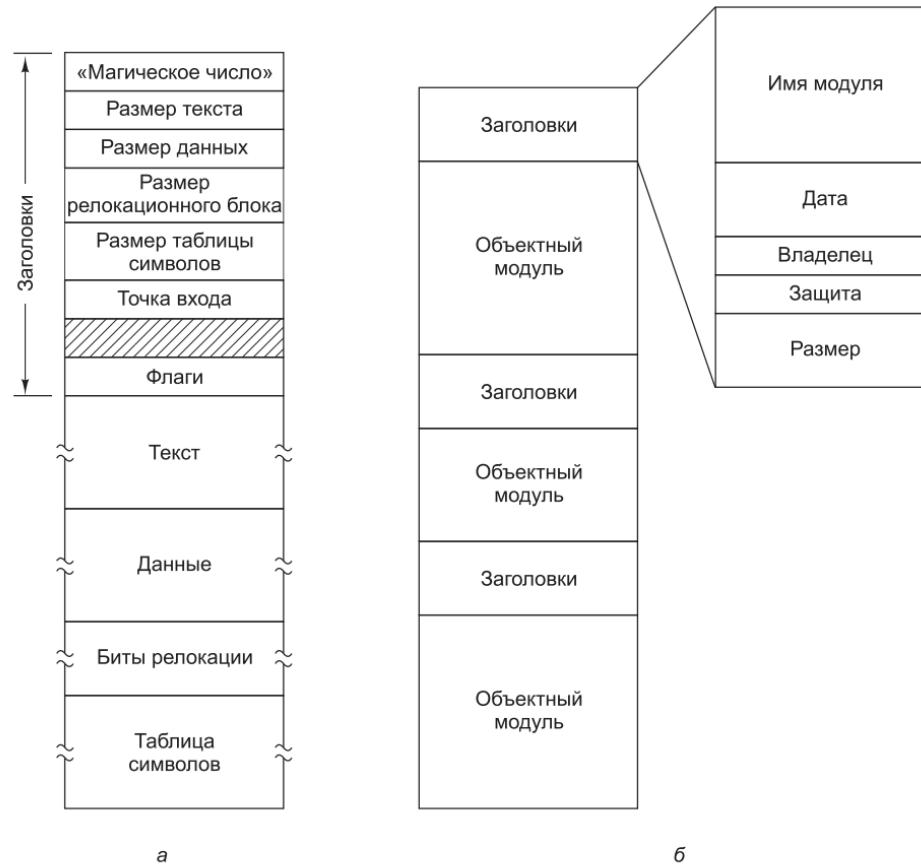


Рис. 4.2. Примеры структур двоичных файлов: а — исполняемый файл; б — архив

Доступ к файлам

- **Последовательный** – с данным доступом процесс мог читать все байты или записи файла только по порядку, с самого начала, но не мог перепрыгнуть и считать их вне порядка их следования.
 - Но последовательные файлы можно было перемотать назад, чтобы считать их по мере необходимости.
- **Произвольный** – файлы, в которых байты или записи могли быть считаны в любом порядке

Атрибуты файлов

Атрибут	Значение
Защита	Кто и каким образом может получить доступ к файлу
Пароль	Пароль для получения доступа к файлу
Создатель	Идентификатор создателя файла
Владелец	Текущий владелец
Флаг «только для чтения»	0 — для чтения и записи; 1 — только для чтения
Флаг «скрытый»	0 — обычный; 1 — не предназначенный для отображения в перечне файлов
Флаг «системный»	0 — обычный; 1 — системный
Флаг «архивный»	0 — прошедший резервное копирование; 1 — нуждающийся в резервном копировании
Флаг «ASCII/двоичный»	0 — ASCII; 1 — двоичный
Флаг произвольного доступа	0 — только последовательный доступ; 1 — произвольный доступ
Флаг «временный»	0 — обычный; 1 — удаляемый по окончании работы процесса
Флаги блокировки	0 — незаблокированный; ненулевое значение — заблокированный
Длина записи	Количество байтов в записи
Позиция ключа	Смещение ключа внутри каждой записи
Длина ключа	Количество байтов в поле ключа
Время создания	Дата и время создания файла
Время последнего доступа	Дата и время последнего доступа к файлу
Время внесения последних изменений	Дата и время внесения в файл последних изменений
Текущий размер	Количество байтов в файле
Максимальный размер	Количество байтов, до которого файл может увеличиваться

Операции с файлами

- **Create (Создать).** Создает файл без данных. Цель вызова состоит в объявлении о появлении нового файла и установке ряда атрибутов.
- **Delete (Удалить).** Когда файл больше не нужен, его нужно удалить, чтобы освободить дисковое пространство. Именно для этого и предназначен этот системный вызов.
- **Open (Открыть).** Перед использованием файла процесс должен его открыть. Цель системного вызова open — дать возможность системе извлечь и поместить в оперативную память атрибуты и перечень адресов на диске, чтобы ускорить доступ к ним при последующих вызовах.
- **Close (Закрыть).** После завершения всех обращений к файлу потребность в его атрибутах и адресах на диске уже отпадает, поэтому файл должен быть закрыт, чтобы освободить место во внутренней таблице.
 - Многие системы устанавливают максимальное количество открытых процессами файлов, определяя смысл существования этого вызова.
 - Информация на диск пишется блоками, и закрытие файла вынуждает к записи последнего блока файла, даже если этот блок и не заполнен.

Операции с файлами

- **Read (Произвести чтение).** Считывание данных из файла. Как правило, байты поступают с текущей позиции. Вызывающий процесс должен указать объем необходимых данных и предоставить буфер для их размещения.
- **Write (Произвести запись).** Запись данных в файл, как правило, с текущей позиции. Если эта позиция находится в конце файла, то его размер увеличивается. Если текущая позиция находится где-то в середине файла, то новые данные пишутся поверх существующих, которые утрачиваются навсегда.
- **Append (Добавить).** Этот вызов является усеченной формой системного вызова write. Он может лишь добавить данные в конец файла.
 - Как правило, у систем, предоставляющих минимальный набор системных вызовов, вызов append отсутствует, но многие системы предоставляют множество способов получения того же результата, и иногда в этих системах присутствует вызов append.

Операции с файлами

- **Get attributes (Получить атрибуты).** Процессу для работы зачастую необходимо считать атрибуты файла.
 - К примеру, имеющаяся в UNIX программа make обычно используется для управления проектами разработки программного обеспечения, состоящими из множества сходных файлов.
 - При вызове программа make проверяет время внесения последних изменений всех исходных и объектных файлов и для обновления проекта обходится компиляцией лишь минимально необходимого количества файлов.
 - Для этого ей необходимо просмотреть атрибуты файлов, а именно время внесения последних изменений.
- **Set attributes (Установить атрибуты).** Значения некоторых атрибутов могут устанавливаться пользователем и изменяться после того, как файл был создан. Такую возможность дает именно этот системный вызов. Характерным примером может служить информация о режиме защиты. Под эту же категорию подпадает большинство флагов.

Операции с файлами

- **Seek (Найти).** При работе с файлами произвольного доступа нужен способ указания места, с которого берутся данные.
 - Одним из общепринятых подходов является применение системного вызова `seek`, который перемещает указатель файла к определенной позиции в файле.
 - После завершения этого вызова данные могут считываться или записываться с этой позиции.
- **Rename (Переименовать).** Нередко пользователю требуется изменить имя существующего файла. Этот системный вызов помогает решить эту задачу.
 - Необходимость в нем возникает не всегда, поскольку файл может быть просто скопирован в новый файл с новым именем, а старый файл затем может быть удален.

Каталоги

- **Каталоги** – это системные файлы, предназначенные для поддержки структуры файловой системы.
- Существуют системы с
 - одноуровневыми каталогами
 - иерархическими системами каталогов

Системы с одноуровневыми каталогами

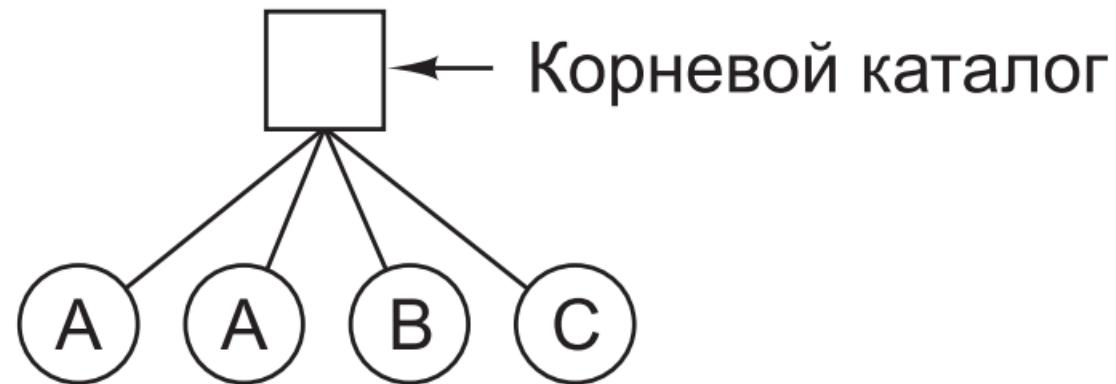


Рис. 4.3. Система с одноуровневым каталогом,
содержащим четыре файла

Иерархические системы каталогов

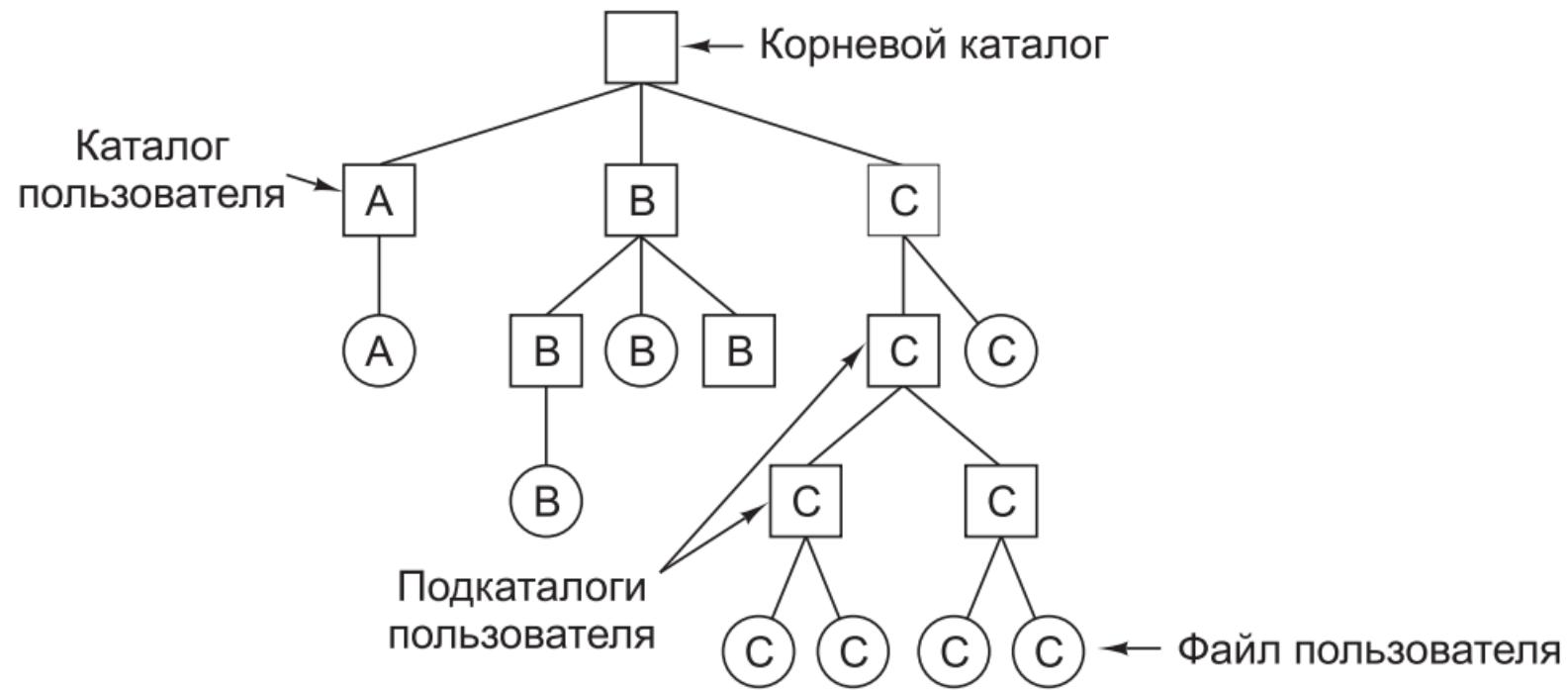


Рис. 4.4. Иерархическая система каталогов

Операции с каталогами

- **Create (Создать каталог).** Каталог создается пустым, за исключением точки и двойной точки, которые система помещает в него автоматически (или в некоторых случаях при помощи программы `mkdir`).
- **Delete (Удалить каталог).** Удалить можно только пустой каталог. Каталог, содержащий только точку и двойную точку, рассматривается как пустой, поскольку они не могут быть удалены.
- **Opendir (Открыть каталог).** Каталоги могут быть прочитаны.
 - К примеру, для вывода имен всех файлов, содержащихся в каталоге, программа `ls` открывает каталог для чтения имен всех содержащихся в нем файлов.
 - Перед тем как каталог может быть прочитан, он должен быть открыт по аналогии с открытием и чтением файла.
- **Closedir (Закрыть каталог).** Когда каталог прочитан, он должен быть закрыт, чтобы освободить пространство во внутренних таблицах системы.

Операции с каталогами

- **Readdir (Прочитать каталог).** Этот вызов возвращает следующую запись из открытого каталога.
 - Раньше каталоги можно было читать с помощью обычного системного вызова `read`, но недостаток такого подхода заключался в том, что программист вынужден был работать с внутренней структурой каталогов, о которой он должен был знать заранее.
 - В отличие от этого, `readdir` всегда возвращает одну запись в стандартном формате независимо от того, какая из возможных структур каталогов используется.
- **Rename (Переименовать каталог).** Во многих отношениях каталоги подобны файлам и могут быть переименованы точно так же, как и файлы.

Операции с каталогами

- **Link (Привязать).**
 - Привязка представляет собой технологию, позволяющую файлу появляться более чем в одном каталоге.
 - В этом системном вызове указываются существующий файл и новое имя файла в некотором существующем каталоге и создается привязка существующего файла к указанному каталогу с указанным новым именем.
 - Таким образом, один и тот же файл может появиться в нескольких каталогах, возможно, под разными именами.
 - Подобная привязка, увеличивающая показания файлового счетчика i-узла (предназначенного для отслеживания количества записей каталогов, в которых фигурирует файл), иногда называется жесткой связью, или жесткой ссылкой (hard link).
- **Unlink (Отвязать).** Удалить запись каталога.
 - Если отвязываемый файл присутствует только в одном каталоге (что чаще всего и бывает), то этот вызов удалит его из файловой системы.
 - Если он фигурирует в нескольких каталогах, то он будет удален из каталога, который указан в имени файла. Все остальные записи останутся.
 - Фактически системным вызовом для удаления файлов в UNIX (как ранее уже было рассмотрено) является unlink.

Буферизация ввода-вывода

- **Буфером ввода-вывода** называется область оперативной памяти, предназначенная для временного хранения записей файла.
- Буфера ввода-вывода предназначены для решения двух задач:
 - устранение несоответствия между размером логической записи файла, определяемым в приложении, и размером данных, который записывается на диск;
 - снижение влияния внешних устройств на скорость работы процессора, которая значительно превышает скорость работы внешних устройств.

Ввод-вывод данных файловой системой

- Для решения этих задач при выводе данных файловая система сначала полностью заполняет буфер логическими записями, а затем дает команду внешнему устройству на запись данных на диск.
- При вводе данных система управления файлами сначала заполняет буфер данными, прочитанными с диска, а затем управляет чтением логических записей из буфера в программу пользователя.

Организация буферов ввода-вывода

- Для ускорения ввода-вывода данных обычно используется несколько буферов ввода-вывода, которые организованы в кольцевую очередь.
- Во время работы пользовательского процесса с одним буфером, файловая система параллельно осуществляет ввод или вывод данных в другие буфера.

Работа с файлами в Windows

- **CreateFile** – создание нового или открытие уже существующего файла;
- **WriteFile** – запись данных в файл, эта функция может использоваться как для синхронной, так и для асинхронной записи данных;
- **ReadFile** – чтение данных из файла, эта функция может использоваться как для синхронного, так и асинхронного чтения данных;

Работа с файлами в Windows

- **SetFilePointer** – установка указателя позиции файла;
- **FlushFileBuffers** – освободить буфер от записей;
- **CloseHandle** – закрытие доступа к файлу;
- **DeleteFile** – удаление файла с диска.
- **CopyFile** – копирование файла;
- **MoveFile** – перемещение файла.
- **ReplaceFile** – замещение файла.

ФУНКЦИИ для работы с атрибутами файла.

- **GetFileAttributes** – определить атрибуты файла;
- **SetFileAttributes** – изменить атрибуты файла;
- **GetFileInformationByHandle** – получить информацию о файле;
- **GetFileType** – определить тип файла;
- **GetBinaryType** – определить, является ли файл исполняемым;
- **GetFileSize** – определить размер файла;
- **SetEndOfFile** – изменить размер файла.

Функции для блокировки доступа к файлу.

- **LockFile** – блокирование доступа к файлу для монопольного использования;
- **UnlockFile** – разблокирование доступа к файлу.

Отображение файлов в память

- В операционных системах Windows реализован механизм, который позволяет отображать на адресное пространство процесса не только содержимое файлов подкачки, но и содержимое обычных файлов.
- В этом случае файл или его часть рассматривается как набор виртуальных страниц процесса, которые имеют последовательные логические адреса.
- Файл, который отображен на адресное пространство процесса, называется **отображенным в память файлом**.
- Отображение файла в адресном пространстве процесса называется также **представлением** или **видом файла** (file view).
- После отображения файла в адресное пространство процесса, доступ к файлу может осуществляться через указатель как к обычным данным в адресном пространстве процесса.

Когерентность данных

- Несколько процессов могут одновременно отображать один и тот же файл в свое адресное пространство.
- В этом случае операционная система обеспечивает согласованность содержимого файла для всех процессов, если доступ к этим данным осуществляется как к области виртуальной памяти процесса, т. е. для доступа к файлу не используется функция WriteFile.
- Такая согласованность данных, хранящихся в файле, отображенном в память несколькими процессами, называется **когерентностью данных**.

Назначение механизма отображения файлов в память

- Загрузка программы на выполнения в адресном пространстве процесса;
- Динамическое подключение библиотек функций во время выполнения программы;
- Обмен данными между процессами, принимая во внимание то, что система обеспечивает когерентность данных в файле, отображаемом в память.

Порядок работы с файлом, отображаемым в память

1. Открыть файл, который будет отображаться в память.
2. Создать объект ядра, который выполняет отображение файла.
3. Отобразить файл или его часть на адресное пространство процесса.
4. Выполнить необходимую работу с видом файла.
5. Отменить отображение файла.
6. Закрыть объект ядра для отображения файла.
7. Закрыть файл, который отображался в память.

ФУНКЦИИ ДЛЯ ОТОБРАЖЕНИЯ ФАЙЛА В ПАМЯТЬ

- **CreateFileMapping** – создать объект ядра для отображения файла в память;
- **MapViewOfFile** – отобразить файл в память;
- **UnmapViewOfFile** – отменить отображение файла в память;
- **FlushViewOfFile** – сброс содержимого вида на диск.

Динамически подключаемые библиотеки

- **Динамически подключаемая библиотека или DLL (Dynamic Link Library)**, представляет собой программный модуль, который может быть загружен в виртуальную память процесса как статически во время создания исполняемого модуля процесса, так и динамически во время исполнения процесса операционной системой.
- Программный модуль оформленный в виде DLL хранится на диске в виде файла, который имеет тип DLL, и может содержать как функции, так и данные.
- Для загрузки DLL в память используется механизм отображения файлов в память.
- Динамически подключаемые библиотеки предназначены главным образом для разработки функционально замкнутых библиотек функций, которые могут использоваться разными приложениями.

Создание DLL

- Создаются DLL подобно обычным исполняемым модулям.
- Как и каждая программа на C++, динамически подключаемая библиотека должна иметь главную функцию, которая отмечает точку входа в программу при её исполнении операционной системой.
- В отличие от исполняемых модулей, в которых эта функция называется `main`, в DLL главная функция называется `DllMain`.
- Функция `DllMain` вызывается операционной системой при загрузке DLL в адресное пространство процесса и при создании этим процессом нового потока.
- Главное назначение функции `DllMain`:
 - инициализация DLL при её загрузке;
 - захват и освобождение ресурсов при создании и завершении нового потока в процессе.

Функции, предназначенные для работы с DLL

- **LoadLibrary** – загрузка DLL;
- **FreeLibrary** – отключение DLL;
- **GetProcAddress** – доступ к импортируемым из DLL функциям и переменным.

9. Управление устройствами компьютера

9.1. Логическая структура компьютера

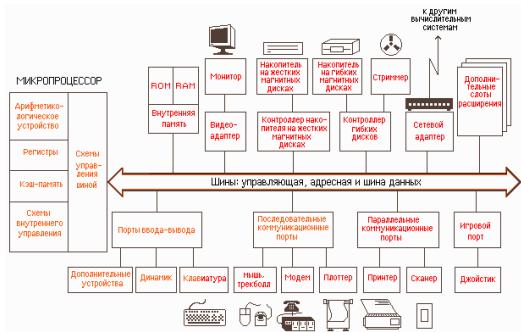


Логическая структура компьютера с архитектурой общая шина

© Pobegailo A.P. 2012

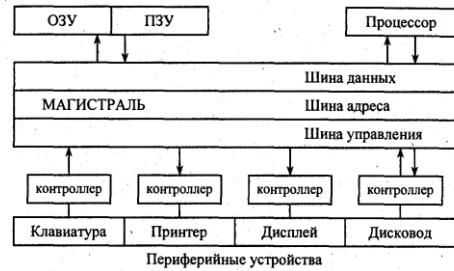
© Pobegailo A.P. 2012

Логическая структура компьютера (на экзамене не нужна)



© Pobegailo A.P. 2012

Логическая структура компьютера (на экзамене не нужна)



© Pobegailo A.P. 2012

Обозначения

- ЦП – центральный процессор (CPU – central processing unit);
- ОП – оперативная память (RAM – random access memory);
- Y1, Y2, ..., Un – устройства (devices).

- Центральным процессором** называется интегральная схема (ИС), которая выполняет две основные функции:
 - исполняет команды программы;
 - управляет работой устройств компьютера.
- Оперативной памятью** называется набор ИС, которые предназначены для хранения команд и данных программы.

© Pobegailo A.P. 2012

© Pobegailo A.P. 2012

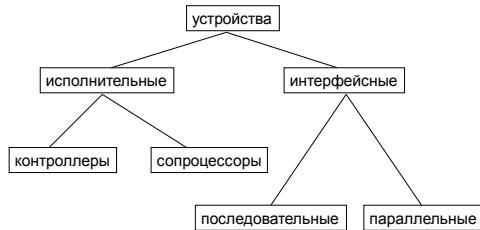
- Устройством называется ИС или набор ИС, которые выполняют вспомогательные функции, исполняя специальные команды.
- Системной шиной называется набор ИС и линий передачи сигналов, которые предназначены для обмена сигналами между ЦП, ОП и устройствами.

- Системная шина обеспечивает 3 вида обмена данными:
 - ЦП \Leftrightarrow ОП;
 - ЦП \Leftrightarrow Y;
 - ОП \Leftrightarrow Y.
- Системная шина состоит из трех шин:
 - шины управления, по которой передаются управляющие сигналы;
 - шины данных, по которой передаются данные;
 - адресной шины, по которой передаются адреса.

© Pobegailo A.P. 2012

© Pobegailo A.P. 2012

9.2. Типы устройств



Классификация устройств

© Pobegailo A.P. 2012

Исполнительные устройства

- Исполнительные устройства – это ИС, которые функционально дополняют ЦП.
- Различают два типа исполнительных устройств:
 - контроллеры;
 - процессоры.

© Pobegailo A.P. 2012

Контроллеры

- Контроллеры – это ИС, которые предназначены для управления другими устройствами, освобождая от этих функций ЦП.
- Пример1. Контроллер прерываний управляет диспетчеризацию сигналов прерываний от устройств.
- Пример 2. Контроллер прямого доступа к памяти управляет прямым доступом устройств к ОП.

© Pobegailo A.P. 2012

Сопроцессоры

- Сопроцессоры – это процессоры специального назначения.
- Пример. Процессор цифровой обработки сигналов (DSP – digital signal processor).

© Pobegailo A.P. 2012

Интерфейсные устройства

- Интерфейсные устройства – это ИС, которые управляют обменом данными между компьютером и внешними устройствами ввода/вывода данных.
- Часто интерфейсные устройства называются *интерфейсами* или устройствами ввода/вывода.

Последовательные и параллельные интерфейсы

- *Последовательные интерфейсы* передают данные последовательно по битам. Например, интерфейс RS 232.
- *Параллельные интерфейсы* передают параллельно целое слово данных (8, 16, 32 или 64 бита).

© Pobegailo A.P. 2012

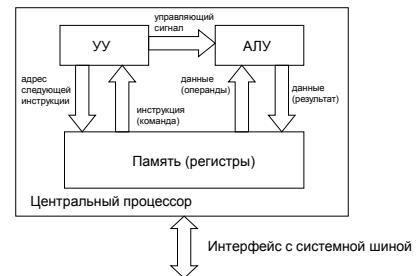
© Pobegailo A.P. 2012

Драйверы

- В операционных системах для доступа к устройствам используются специальные программы, которые называются *драйверами* устройств.

© Pobegailo A.P. 2012

9.3. Логическая архитектура центрального процессора



© Pobegailo A.P. 2012

Обозначения

- УУ – устройство управление;
- Назначение УУ:
 - исполняет инструкции перехода;
- АЛУ – арифметико-логическое устройство;
- Назначение АЛУ:
 - исполняет арифметические команды;
 - исполняет логические команды.

- Память – регистры микропроцессора, в которых хранятся:
 - команды;
 - данные;
 - состояние процессора.
- Введем следующие обозначения для регистров микропроцессора:
 - PC – счетчик команд (program counter);
 - IR – регистр команд (instruction register).
- Через *MemPtr* обозначим массив байтов основной памяти.

© Pobegailo A.P. 2012

© Pobegailo A.P. 2012

Цикл работы процессора

```

while (true)
{
    IR = MemPtr[PC];      // извлекаем команду
    ++PC;                 // увеличиваем счетчик
    if (InstructionCode(IR) == Jump)
        PC = JumpAddress(IR);
    else
        Execute(IR);      // вызов АЛУ
}

```

© Pobegailo A.P. 2012

Слово состояния процессора

- PSW - processor status word (слово состояния процессора)
- Слово состояния процессора содержит информацию о текущем состоянии процессора:
 - результат последней операции в процессоре:
 - равен 0,
 - отрицателен,
 - в результате операции произошел перенос из старшего разряда,
 - произошло арифметическое переполнение
 - флаг прерываний,
 - и другие флаги в зависимости от типа микропроцессора.

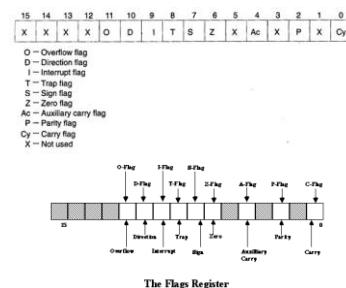
© Pobegailo A.P. 2012

Регистр флагов

- Слово состояния процессора хранится в регистре процессора, который называется:
- Processor status register – регистр состояния процессора или
- Flag register – регистр флагов (микропроцессоры Intel)

© Pobegailo A.P. 2012

Регистр флагов микропроцессора Intel 8086
(на экзамене не нужен)



© Pobegailo A.P. 2012

Архитектура фон Неймана

- Такая архитектура и принципы работы процессора называются архитектурой фон Неймана.
- Принципы архитектуры фон Неймана:
 - данные и инструкции хранятся в основной памяти (программа хранится в основной памяти);
 - для доступа к содержимому основной памяти используется адресация (данные и инструкции в основной памяти неразличимы);
 - инструкции выполняются **последовательно**, порядок исполнения инструкций изменяется явно (последовательность исполнения инструкций изменяют команды перехода).

© Pobegailo A.P. 2012

Структура микропроцессора Intel 8080
(на экзамене не нужна)

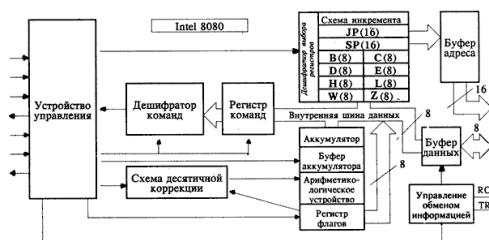
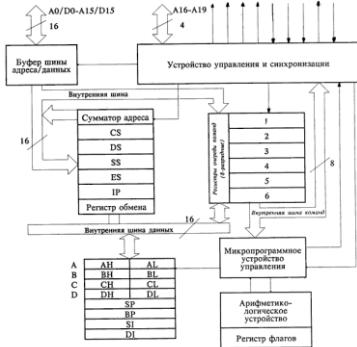


Рис.1. Внутренняя структура микропроцессора 8080.

© Pobegailo A.P. 2012

Структура микропроцессора Intel 8086 (на экзамен не нужна)



© Pobegailo A.P. 2012

© Pobegailo A.P. 2012

9.4. Прерывания

Контекст процессора. Перестановка контекста процессора

- Состояние регистров процессора при исполнении программы называется **контекстом процессора**.
- Контекст процессора определен только в точках между выполнением двух последовательных команд.
- В этих точках можно изменить контекст процессора, т. е. запомнить текущий контекст процессора в памяти, а затем загрузить из памяти в процессор новый контекст.
- Такая операция называется **перестановкой контекста процессора**.

© Pobegailo A.P. 2012

Точка прерывания. Прерывание программы.

- Точка, в которой происходит перестановка контекста процессора, называется **точкой прерывания программы**, т. к. в этом случае последовательность выполнения команд программы прерывается и управление передается другой программе.
- Сам процесс перестановки контекста процессора называется **прерыванием программы**.

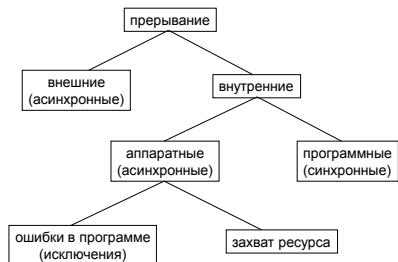
© Pobegailo A.P. 2012

Прерывание

- Прерывание программы происходит только в том случае, если в процессоре установлен специальный флаг, который сигнализирует о прерывании программы.
- Процессор проверяет состояние этого флага после выполнения каждой команды.
- Сигнал, устанавливающий флаг прерывания, называется **прерыванием**.

© Pobegailo A.P. 2012

Классификация прерываний по отношению к исполняемой программе



© Pobegailo A.P. 2012

Внешние прерывания

- Внешние прерывания генерируются внешними устройствами и являются асинхронными по отношению к исполняемой программе, т. к. могут прервать программу в любой точке прерывания.

Внутренние прерывания

- Внутренние прерывания инициализируются самой исполняемой программой:
 - явно в случае программных прерываний;
 - неявно в случае аппаратных прерываний.

© Pobegailo A.P. 2012

© Pobegailo A.P. 2012

Внутренние аппаратные прерывания

- Внутренние аппаратные прерывания являются асинхронными и могут происходить по следующим причинам:
 - ошибки в работе программы, например, деление на нуль или неправильная адресация. Такие прерывания также называются *исключениями*;
 - захват аппаратного ресурса, например, подкачка виртуальной страницы в виртуальную память.

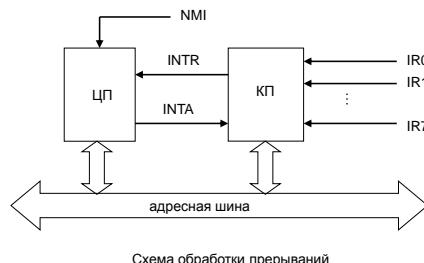
Внутренние программные прерывания

- Внутренние программные прерывания являются синхронными и явно инициируются программой посредством исполнения команды (функции), которая обращается к ядру ОС.

© Pobegailo A.P. 2012

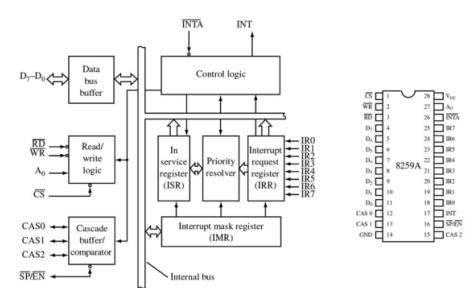
© Pobegailo A.P. 2012

9.5. Обработка прерываний



© Pobegailo A.P. 2012

Структура контроллера прерываний Intel 8259A (на экзамен не нужна)



© Pobegailo A.P. 2012

- Обозначения устройств:
 - ЦП – центральный процессор;
 - КП – контроллер прерываний.
- Обозначение сигналов:
 - IR0, ..., IR7 – сигналы прерывания от устройств (interrupt request);
 - INTR – сигнал запроса на прерывание от КП (interrupt request);
 - INTA – сигнал подтверждения прерывания от ЦП (interrupt acknowledge);
 - NMI – сигнал немаскируемого прерывания.

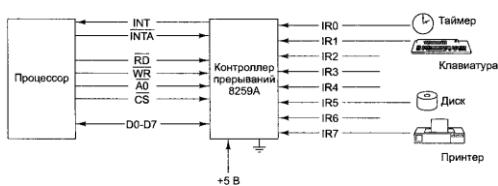
© Pobegailo A.P. 2012

Маскирование прерываний

- ЦП может блокировать обработку сигнала по линии INTR.
- КП может блокировать обработку сигнала по линиям IR0, ..., IR7.
- В этом случае говорят, что соответствующие прерывания **маскируются**.
- Прерывания по линии NMI не маскируются. Эта линия служит для передачи сигнала прерывания в случае аварии оборудования, например, падение напряжения в сети.

© Pobegailo A.P. 2012

Подключение внешних устройств к контроллеру прерываний (на экзамене не нужно)



© Pobegailo A.P. 2012

Диспетчеризация прерываний

- КП выполняет диспетчеризацию сигналов прерывания от устройств, которые поступают на входы IR0, ..., IR7.
- Для этого КП программируется таким образом, что каждому из входов IR0, ..., IR7 ставится в соответствие свой приоритет.
- Если на входы КП поступает несколько сигналов прерывания, то обрабатывается сигнал с наибольшим приоритетом, а остальные маскируются.

© Pobegailo A.P. 2012

Дисциплины обслуживания прерываний

- Такая дисциплина обслуживания прерываний называется **одноуровневой с приоритетами**.
- Контроллеры прерываний могут подключаться каскадом, т. е. выход INTR одного КП подключается к одному из входов IR0, ..., IR7 другого КП.
- В этом случае дисциплина обслуживания прерываний называется **многоуровневой с приоритетами**.

© Pobegailo A.P. 2012

Алгоритм обработки сигналов прерывания от устройств

- Сигналы прерывания поступают на входы IR0, ..., IR7 контроллера прерываний.
- КП обрабатывает сигналы и выдает запрос ЦП на вход INTR.
- Если вход INTR не маскирован, то ЦП передает КП подтверждение о получении сигнала прерывания по линии INTA.
- По второму сигналу от ЦП по линии INTA контроллер прерываний устанавливает на адресную шину адрес программы обработки прерывания.
- ЦП передает управление программе обработки прерывания.

© Pobegailo A.P. 2012

Передача управления программе обработки прерывания

- Передача управления программе обработки прерывания происходит путем перестановки контекста процессора, которая выполняется аппаратно.
- Если процессор сохраняет только свое текущее слово состояния (PSW – processor status word), то программа обработки прерывания должна иметь следующую структуру.

Структура программы обработки прерывания

```
вход:      // сохранение PSW
сохранение содержимого регистров;
обработка прерывания;
восстановление содержимого регистров;
выход;    // восстановление PSW
```

© Pobegailo A.P. 2012

© Pobegailo A.P. 2012

*9.6. Работа с устройствами в Windows

- В Windows, как и в других ОС, для обработки прерываний от устройств и для доступа к устройствам используются специальные программы, которые называются **драйверами устройств**.
- Взаимодействие с драйверами происходит по специальным интерфейсам, которые называются **интерфейсами устройств**.

- Все интерфейсы устройств делятся на классы.
- Для поиска устройств, которые поддерживают требуемый интерфейс, предназначена функция
SetupDiGetClassDevs
- Для работы с драйверами устройств предназначена функция
DeviceIoControl

© Pobegailo A.P. 2012

© Pobegailo A.P. 2012