Распределенные информационные системы

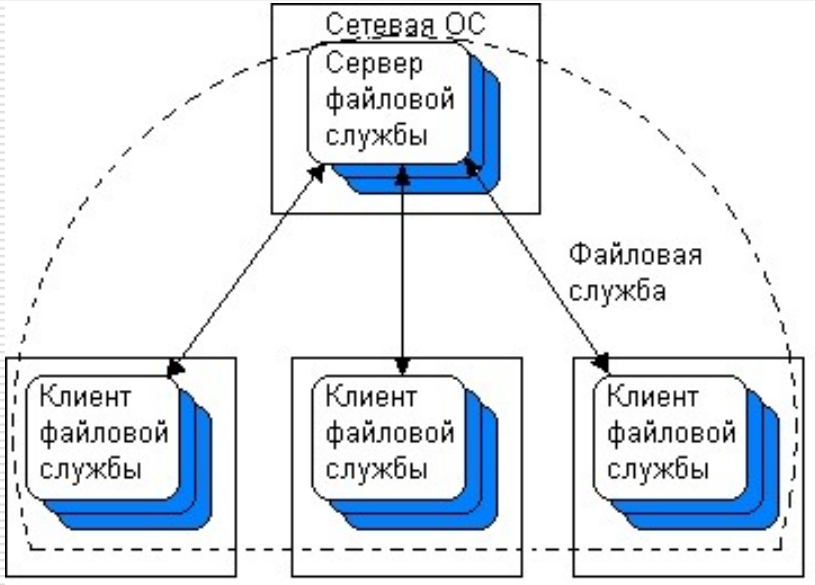
Тема 9. Процессы и потоки в РИС

Модель взаимодействия распределенной системы

- В отличие от локальных систем для которых понятие алгоритма работы программы знакомо всем программистам, в распределенных системах поведение системы определяется взаимодействием процессов работающих на удаленных машинах друг с другом и описывается **распределенными алгоритмами** работы системы.
- Поведение и состояние распределенной системы определяется теми **действиями** которые выполняются на удаленных узлах, а также **сообщениями**, которыми они обмениваются друг с другом.
- **Взаимодействие между процессами** определяет всю **функциональность** распределенной системы.
- При этом каждый процесс **имеет** свое **состояние**, обладает своим набором **данных**, включая **переменные** которое определяется узлами

Реализация взаимодействия между компонентами в распределенных системах

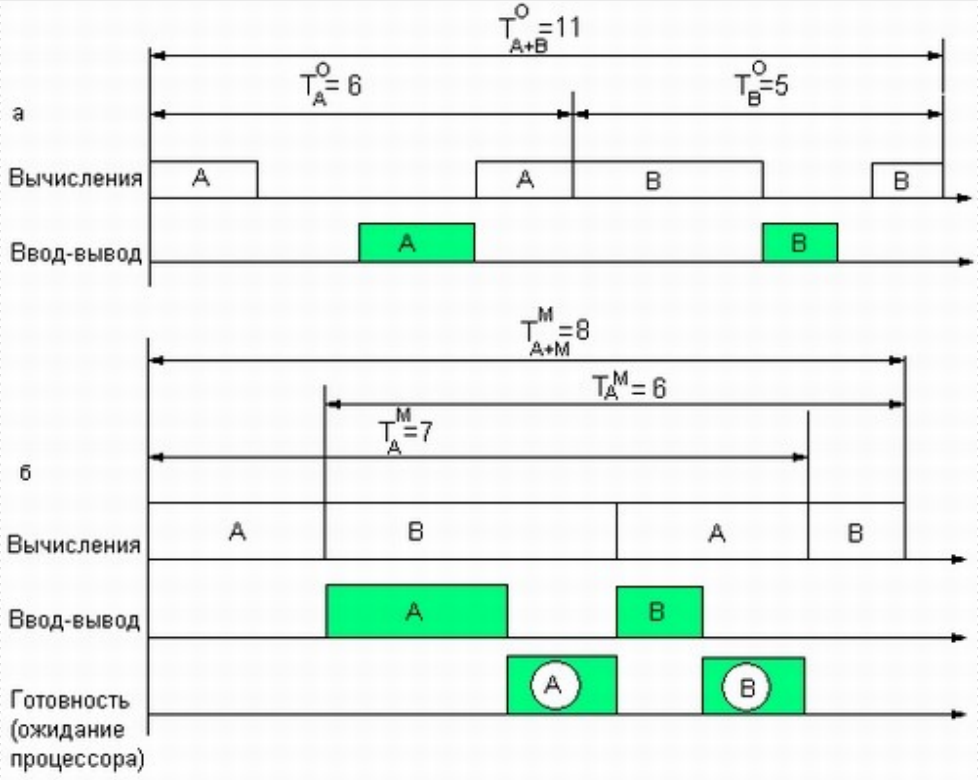
- В распределенных системах взаимодействие между компонентами реализуется **средствами операционных** систем узлов.
- Основным элементом с помощью которого реализуются функции выполняемые узлами являются **процессы**.
- **Концепция процесса** зародилась в операционных системах, где под этим понятием обычно обозначают выполняемую программу.

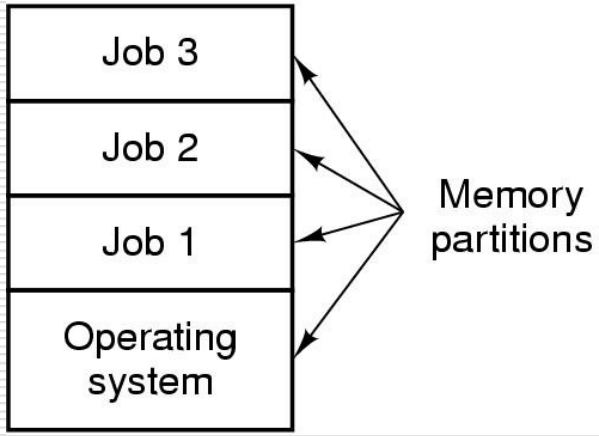


- С точки зрения ОС наиболее важными являются вопросы планирования и управления процессами.
- В РИС наиболее важными являются вопросы:
  - ✓ **скорость реакции** узла на поступающие запросы;
  - ✓ минимизации **времени ответа**;
  - ✓ обеспечение **прозрачности** доступа к ресурсам РС;
  - ✓ **масштабируемость** сервиса при возрастании потока запросов и т.п

Мультипрограммирование или многозадачный режим работы

ОС пакетной обработки





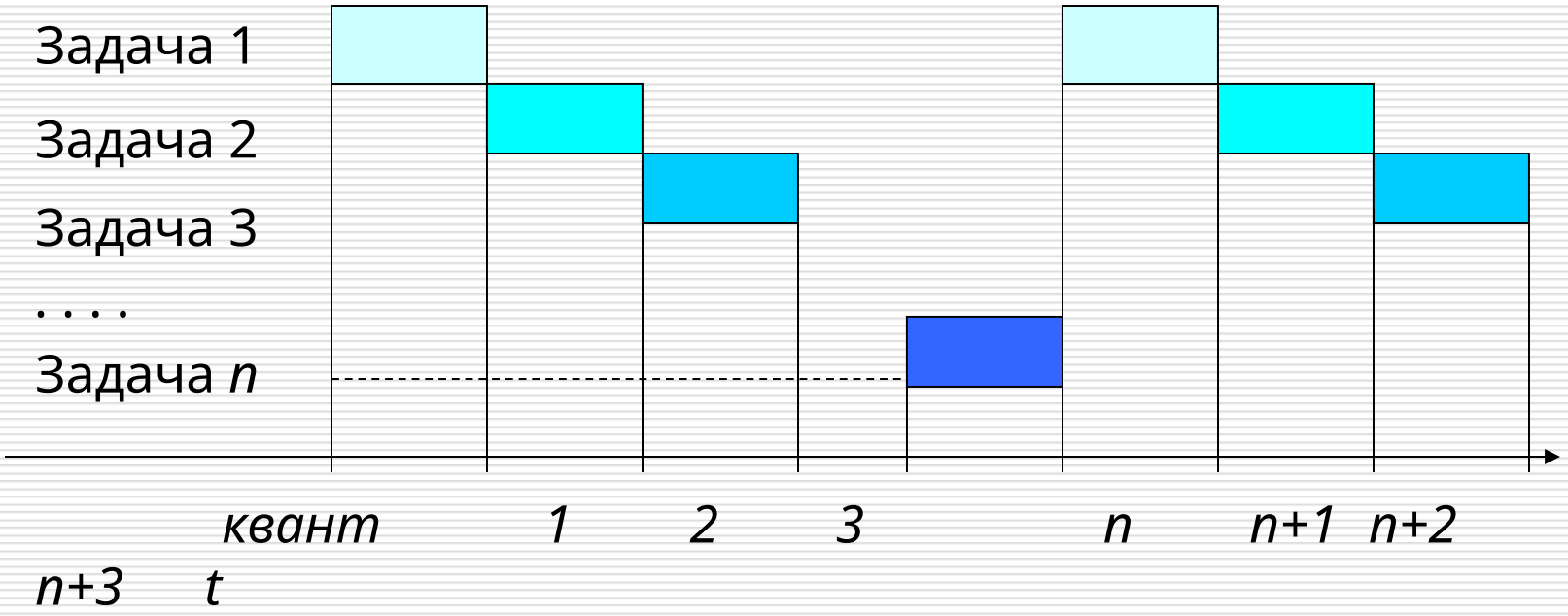
Размещение нескольких заданий в памяти.

Пример ОС:

DOS, OS/360 – фирма IBM

Время выполнения двух задач: в однопрограммной системе (а), в мультипрограммной системе (б)

Системы разделения времени
Квантование времени (MULTICS 1968)



Процессы выполнения

- Для выполнения программ операционная система создает несколько виртуальных процессоров, по одному для каждой программы.
- Чтобы отслеживать эти виртуальные процессоры, операционная система поддерживает ***таблицу процессов*** *(process table), содержащую записи для сохранения значений регистров* процессора, карт памяти, открытых файлов, учетных записях пользователей, привилегиях и т. п.
- *Процесс (process) часто определяют как выполняемую программу,* то есть **программу**, которая в настоящее время выполняется на одном из виртуальных процессоров операционной системы.

Понятие процесса

- **Процессом**, называют программу в момент выполнения, в некоторых ОС исполняемую программу называют **задачей**, процесс и задача являются синонимами.
- С каждым процессом связывается его **адресное пространство** — список адресов в памяти от некоторого минимума (обычно нуля) до некоторого максимума, которые процесс может прочесть и в которые он может писать. Адресное пространство содержит саму программу, данные к ней и ее стек.
- Со всяким процессом связывается некий **набор регистров**, включая счетчик команд, указатель стека и другие аппаратные регистры, плюс вся остальная информация, необходимая для запуска программы. Это **информация о процессе**.
- Вся совокупность информации о процессе, необходимая для ее продолжения процессором после прерывания называется **контекстом** процесса. Контекст процесса является частью (подмножеством) информации о процессе.
- Всякий процесс выполняемый в системе запускается от **имени пользователя** инициировавшего запуск программы на выполнение.
- **Права доступа процесса** к ресурсам системы определяются **правами доступа пользователя**, от чьего имени программа была запущена.

Идентификация процессов в системе

- Для идентификации процессов в системе используются идентификаторы процессов PIDs (Process Identifier)
- **PID** – это число присваемое процессу при запуске.
- Каждому процессу присваевается идентификатор пользователя (UID – User Idetifier) и GID (Group Identifier) запустившего данный процесс.

Таблица процессов

- Это массив (или связанный список) структур хранящий информацию о процессах исполняемых системой.
- Каждому процессу в таблице процессов соответствует один экземпляр структуры, хранящий информацию о данном процессе.
- Процесс может находиться в одном из двух состояний:
  - выполнение;
  - остановлен.

Адресное пространство и контекст процесса

- С каждым процессом связано его **адресное пространство** — список адресов ячеек памяти от нуля до некоторого максимума, откуда процесс может считывать команды и данные и куда может записывать результаты своей работы.
- **Адресное пространство** содержит:
  - выполняемую программу,
  - данные этой программы,
  - стек программы.
- Кроме этого, с каждым процессом связан набор ресурсов (**контекст процесса**), который обычно включает:
  - регистры (в том числе счетчик команд и указатель стека),
  - список открытых файлов,
  - необработанные предупреждения,
  - список связанных процессов
  - всю остальную информацию, необходимую в процессе работы программы.

Процесс как контейнер для исполняемой программы

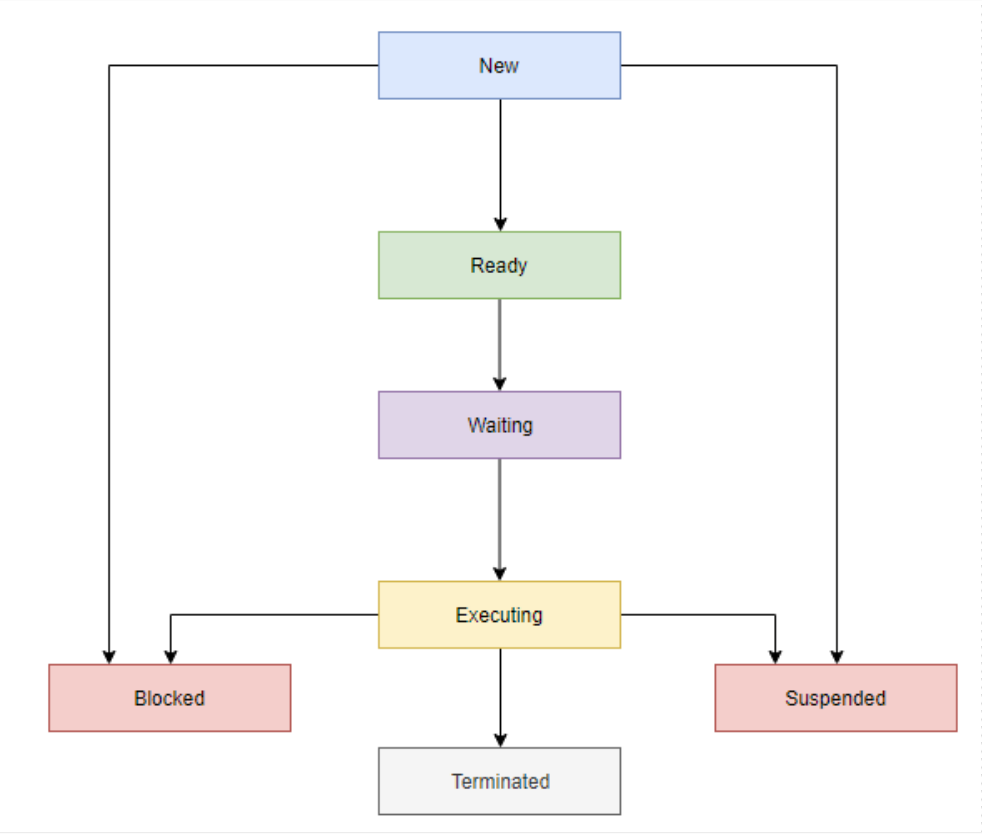
- Процесс может находиться в одном из двух состояний:
  - Останова
  - Выполнения
- Процесс (в том числе приостановленный) состоит из собственного адресного пространства, которое обычно называют **образом памяти**, **контекста процесса (**с содержимым его регистров**)** и записи в **таблице процессов**, а также другой информацией, необходимой для последующего возобновления процесса.
- Таким образом в ОС, **процесс —** это **контейнер**, в котором содержится вся информация, необходимая для работы программы в системе.

Основные характеристики процесса

- Ключевым понятием во всех операционных системах является **процесс,** который в ОС описывается с помощью ряда характеристик, основными из них являются:
- Каждый процесс существующий в системе имеет свой идентификатор **PID – Process Identifier**.
- Информация о каждом процессе, хранится в таблице операционной системы, которая называется **таблицей процессов**.
- Каждый процесс выполняется от имени того **пользователя**, который его запустил.
- Для запуска **системных процессов** используются специальные системные учетные записи пользователей (например, demon (UNIX) или NetworkManager (Windows).
- Пользователи в системе представлены с помощью пары идентификаторов: **UID** (User Identifier)/**GID** (Group/Identifier).
- **Права доступа процесса** к ресурсам компьютера определяются UID/GID пользователя от имени которого работает процесс.
- Все характеристики процесса хранятся в ОС в виде структуры, часто называемой TCB (Task Control Block, z/OS) или **PCB** (**Process Control Block,** Unix).

Жизненный цикл и состояние процесса в ОС

- **ЖЦ процесса** включает в себя следующие **стадии** :
  - создание процесса;
  - выполнение процесса;
  - уничтожение процесса.
- Процесс может находиться в одном из следующих **состояний**:
  - готов к выполнению (Ready);
  - готов в ожидании (Ready Waiting)
  - выполняется (Exectuting);
  - заблокирован (Blocked);
  - приостановлен (Sudpend);
  - завершен (Terminated).



Процессы выполнения в многоядерных системах

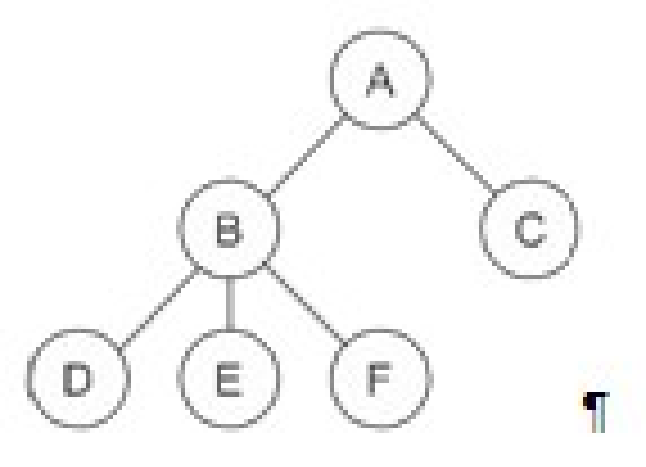
- Для выполнения программ операционная система создает несколько **виртуальных процессоров**, по одному для каждой программы.
- Чтобы отслеживать эти виртуальные процессоры, операционная система поддерживает ***таблицу процессов*** *(process table), содержащую записи для сохранения значений регистров* процессора, карт памяти, открытых файлов, учетных записях пользователей, привилегиях и т. п.
- *Процесс (process) часто определяют как выполняемую программу*, то есть **программу**, которая в настоящее время выполняется на одном из виртуальных процессоров операционной системы.

Составляющие процесса

- В режиме **выполнения**:
  - **адресное пространство** процесса в оперативной памяти ЭВМ;
  - информация о процессе и **ресурсах** которые он использует, например об открытых файлах или установленных сетевых соединениях.
- В режиме **останова**:
  - **образ памяти** процесса (адресное пространство процесса);
  - информация о процессе хранящаяся в **таблице процессов**.

Дерево процессов в ОС

- Если процесс способен создавать несколько других процессов (называющихся **дочерними процессами**), а эти процессы в свою очередь могут создавать собственные дочерние процессы, то в системе образуется дерево процессов.
- В системе имеется процесс с PID=1, который является прародителем всех процессов.



Составляющие контента процесса

- В режиме выполнения:
  - адресное пространство процесса в оперативной памяти ЭВМ;
  - информация о процессе и ресурсах которые он использует, например об открытых файлах или установленных сетевых соединениях.
- В режиме останова:
  - образ памяти процесса (адресное пространство процесса);
  - информация о процессе хранящаяся в таблице процессов.

Жизненный цикл процесса в ОС

- Включает в себя следующие стадии:
  - Создание процесса;
  - выполнение процесса;
  - уничтожение процесса.

Системные вызовы управляющие процессами

- Процесс создается родительским процессом с помощью обращения к функции ядра «**создать процесс**».
- Обращение к функциям ОС называется системным вызовом.
- Главными системными вызовами, управляющими процессами, являются вызовы связанные с созданием и уничтожением процессов.
- Системный вызов выполняется в привилегированном режиме работы процессора.

Смена режимов работы процессора при выполнении системного вызова

Виды системных вызовов связанных с процессами

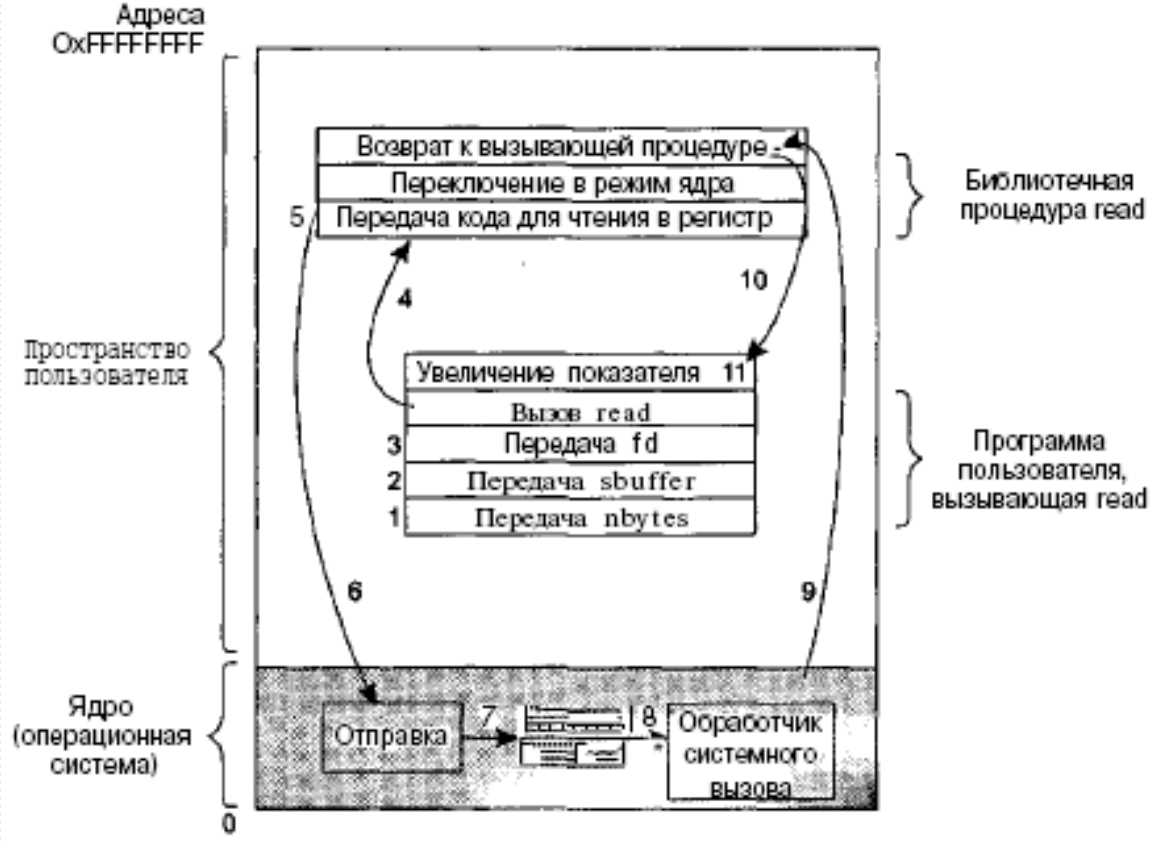
- создание процесса;
- освобождение или выделение дополнительной памяти процессу;
- ожидание завершения какого-либо процесса;
- перекрытие адресных пространств процесса;
- передача сигнала процессу;
- завершение процесса;
- и др.

Системные вызовы (Unix)

1-3 размещение параметров вызова в стек;

4 - выполнение вызова библиотечной процедуры read;

5 – размещение номера системного вызова в регистр;



6 – переключение в **режим ядра**;

7 – проверка номера системного вызова и передача управления обработчику;

8 – выполнение обработчика;

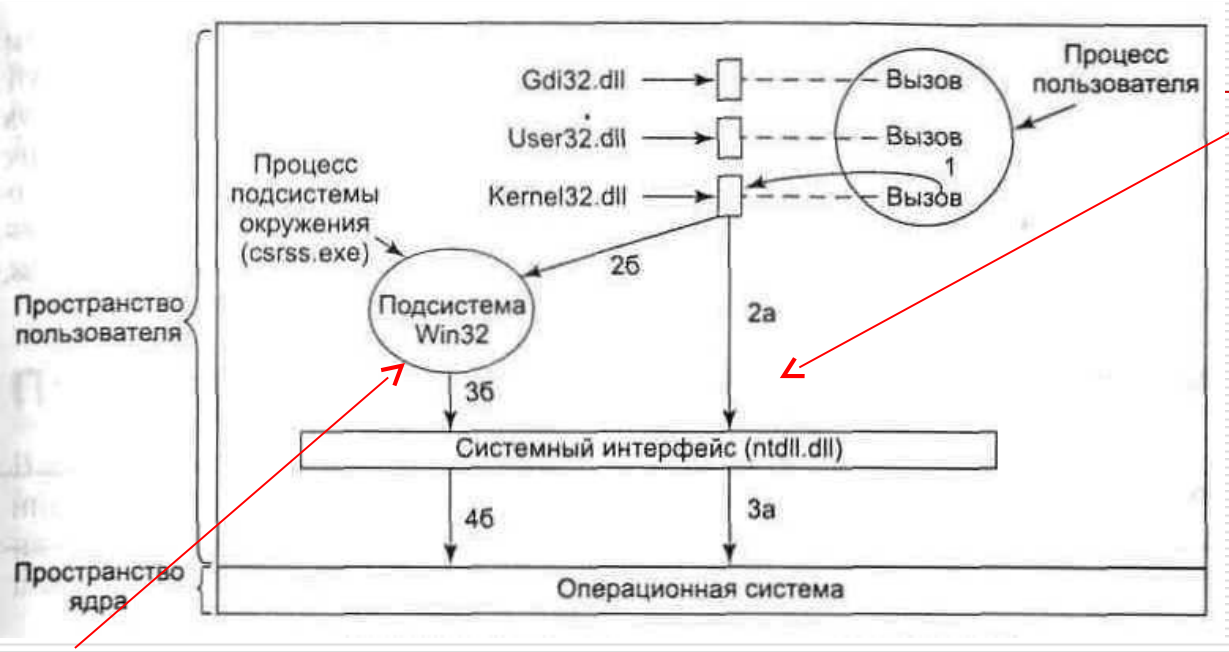
9 – возврат в **режим пользователя**;

10 – возврат из процедуры read;

11 – очистка стека (восстановление указателя вершины стека).

count = read(fd, buffer, nbytes);

Выполнение вызовов Win32 API (Windows NT/2kX)

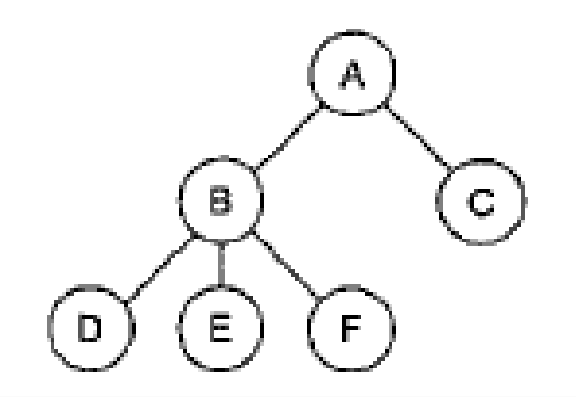


Вариант 1 (короткий путь).
2а и 3а динамические библиотеки обращаются к другой динамической библиотеке (**ntdll.dll**) которая, в свою очередь, обращается к ядру операционной системы. При этом dll может выполнить всю работу самостоятельно, совсем не обращаясь к системным вызовам;

Вариант 2 (длинный путь).

- 2б, 3б и 4б. Для других вызовов Win32 API выбирает маршрут: процесс пользователя обращается к подсистеме **Win32 (csrss.exe)**, которая выполняет некоторую работу, и затем обращается к системному вызову (**ntdll.dll**);
- **В первой версии Windows NT** практически все вызовы Win32 API выбирали маршрут 2б, 3б, 4б, так как большая часть операционной системы (например, графика) была помещена в пространство пользователя.
- Начиная с версии NT 4.0, для увеличения производительности большая часть кода была перенесена в ядро (в драйвер **Win32/GDI**);
- В Windows 2000 и всех последующих версиях WinNT только небольшое количество вызовов Win32 API, например вызовы для создания процесса или потока, идут по длинному пути. Остальные вызовы выполняются напрямую, минуя подсистему окружения Win32.

Дерево процессов



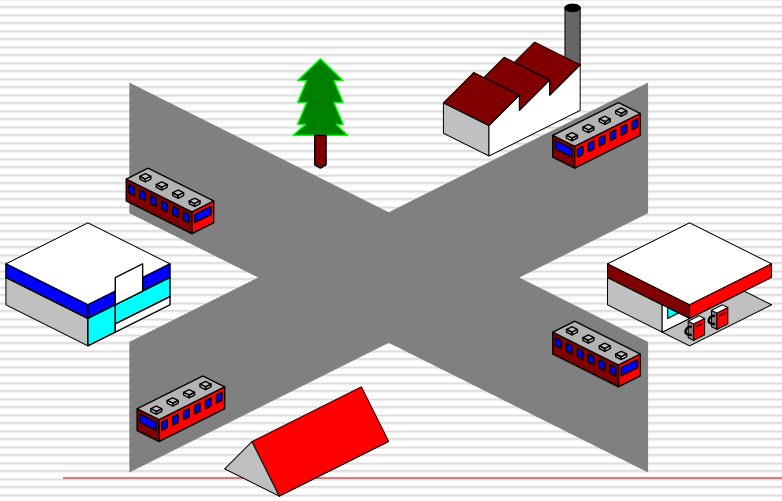
- Если процесс может создавать несколько других процессов (называющихся **дочерними процессами),** а эти процессы, в свою очередь, тоже могут создать дочерние процессы, таким образом формируется дерево процессов,

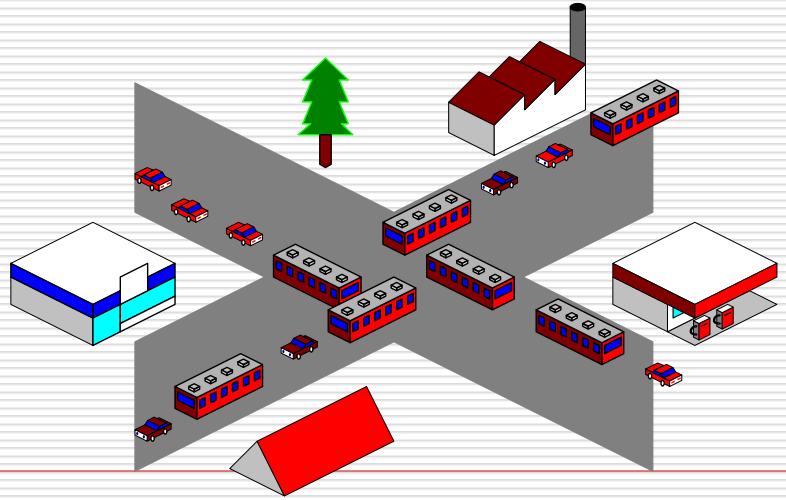
Связанные процессы и межпроцессное взаимодействие

- Связанные процессы — это те, которые объединены для выполнения некоторой задачи, и им нужно часто передавать данные от одного к другому и синхронизировать свою деятельность.
- Такая связь называется межпроцессным взаимодействием.
- **Межпроцессное взаимодействие** (Inter-process communication (IPC)) — это набор методов для обмена данными между потоками процессов. Процессы могут быть запущены как на одном и том же компьютере, так и на разных, соединенных сетью.

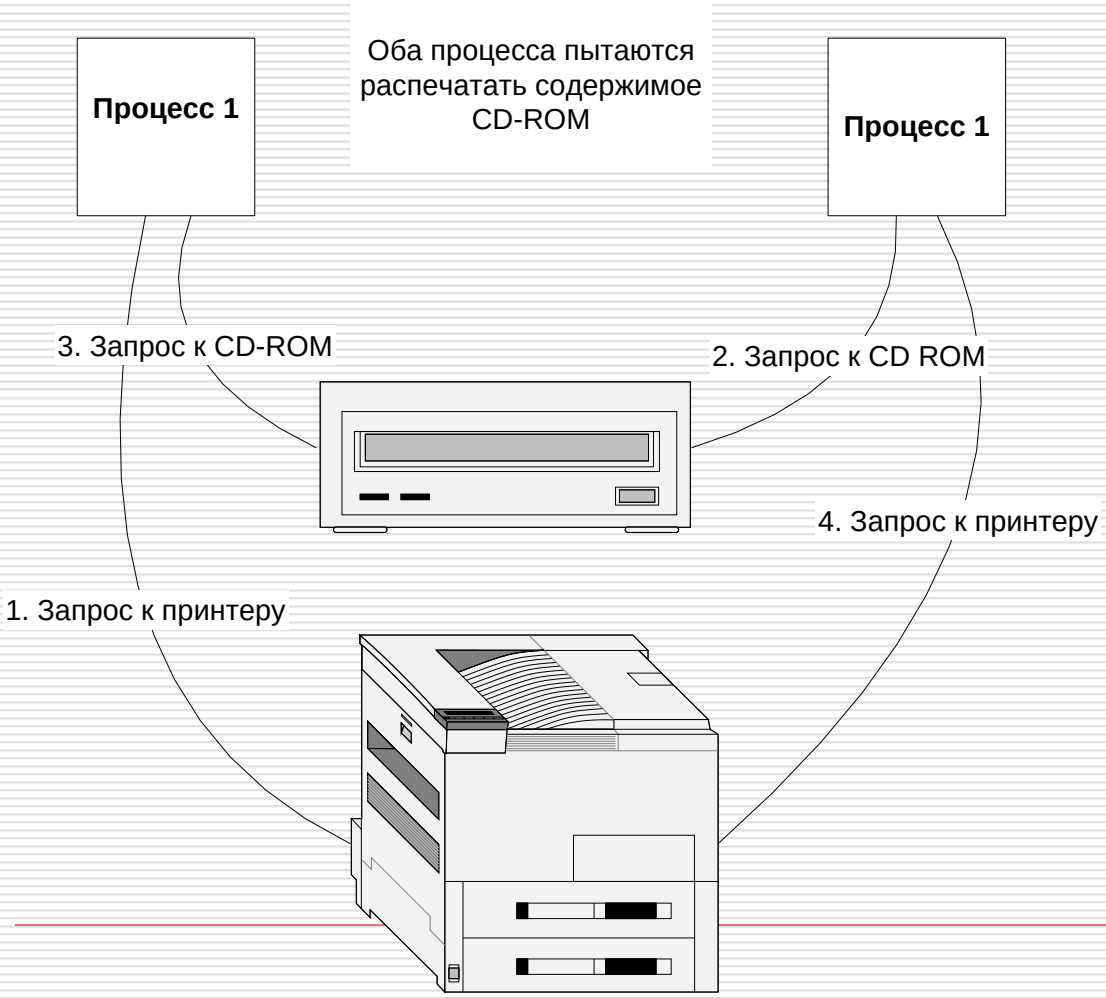
Взаимоблокировка процессов

- Когда взаимодействуют два или более процессов, они могут попадать в патовые ситуации, из которых невозможно выйти без посторонней помощи.
- Такая ситуация называется тупиком, тупиковой ситуацией или взаимоблокировкой.





Пример тупика



Методы межпроцессного взаимодействия (1)

- Существуют следующие методы межпроцессного взаимодействия:
  - **Каналы** (pipe) средство связи стандартного вывода одного процесса со стандартным вводом другого. Каналы старейший из инструментов IPC, существующий приблизительно со времени появления самых ранних версий оперативной системы UNIX.
  - **Сигналы**. Сигналы являются программными прерываниями, которые посылаются процессу, когда случается некоторое событие. Сигналы могут возникать синхронно с ошибкой в приложении, например SIGFPE (ошибка вычислений с плавающей запятой) и SIGSEGV (ошибка адресации), но большинство сигналов является асинхронными.
  - **Очереди сообщений.** Очереди сообщений представляют собой связный список в адресном пространстве ядра. Очереди сообщений как средство межпроцессной связи позволяют процессам взаимодействовать, обмениваясь данными. Данные передаются между процессами дискретными порциями, называемымисообщениями. Процессы, использующие этот тип межпроцессной связи, могут выполнять две операции:послать или принять сообщение.

Методы межпроцессного взаимодействия (2)

- **Семафоры и мьютексы.** Семафор (semaphore) - это целая переменная, значение которой можно опрашивать и менять только при помощи неделимых (атомарных) операций. Двоичный семафор может принимать только значения 0 или 1. Вычислительный семафор может принимать целые неотрицательные значения. **Мьютекс** - отличается от семафора тем, что только владеющий им поток может его освободить.
- **Разделяемая память.** Разделяемая память может быть наилучшим образом описана как отображение участка (сегмента) памяти, которая будет разделена между более чем одним процессом. Это гораздо более быстрая форма IPC, потому что здесь нет никакого посредничества (т.е. каналов, очередей сообщений и т.п.).
- **Сокеты.** Сокеты обеспечивают двухстороннюю связь типа точка-точка *между двумя процессами.* Они являются основными компонентами межсистемной и межпроцессной связи. Каждый сокет представляет собой конечную точку связи, с которой может быть совмещено некоторое имя. Он имеет определенный тип, и один процесс или несколько, связанных с ним процессов.

Сигналы передаваемые процессам

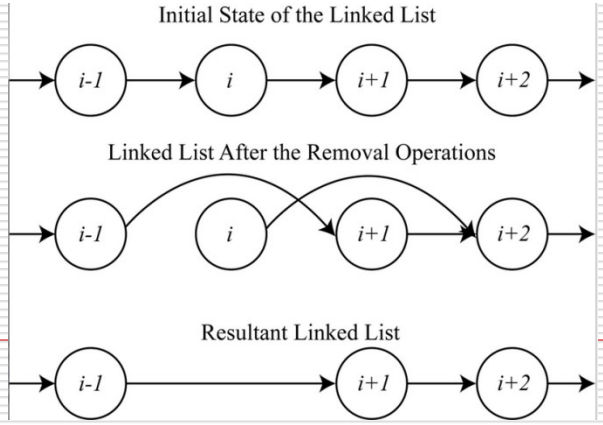
- Сигналы являются программными аналогами аппаратных **прерываний** и могут быть сгенерированы по различным причинам, а не только из-за истечения какого-либо интервала времени.
- Многие **аппаратные прерывания** (например, вызванные выполнением недопустимой команды или использованием неправильного адреса) также **преобразуются в сигналы процессу**, в котором произошла ошибка.
- Сигнал вызывает:
  - временную **остановку работы** процесса независимо от того, что процесс делает в данный момент;
  - **сохраняет** его **регистры** в стеке
  - запускает специальную **процедуру обработки сигнала** (например, передающую повторно предположительно потерянное сообщение).
- **После завершения обработки** сигнала работающий процесс запускается **заново в том состоянии**, в котором он находился до сигнала.

Семафор

- **Семафо́р** (англ. *semaphore*) — примитив синхронизации[1] работы процессов и потоков, в **основе которого лежит счётчик**, над которым можно производить две атомарные операции: увеличение и уменьшение значения на единицу, при этом операция уменьшения для **нулевого значения** счётчика является **блокирующейся**[2]. Служит для построения более сложных механизмов синхронизации[1] и используется **для синхронизации параллельно работающих задач,** для **защиты передачи данных** через разделяемую память, для **защиты** критических секций, а также **для управления доступом к аппаратному обеспечению**.
- **Легковесный семафор** — механизм, позволяющий в ряде случаев **уменьшить количество системных вызовов** за счёт использования активного цикла ожидания в течение некоторого времени перед выполнением блокировки[3].
- Семафоры могут быть **двоичными** и **вычислительными**.
  - **Вычислительные семафоры** могут принимать **целочисленные неотрицательные** значения и используются для работы с ресурсами, количество которых ограничено, либо участвуют в синхронизации параллельно исполняемых задач.
  - **Двоичные семафоры** являются частным случаем вычислительного семафора и могут **принимать** только два **значения: 0 и 1**

Мьютекс

- **Мью́текс** (англ. *mutex*, от *mutual exclusion* — «взаимное исключение») — аналог одноместного семафора, служащий в программировании для синхронизации одновременно выполняющихся потоков.
- Мьютекс отличается от семафора тем, что **только владеющий** им поток может **его освободить**, т.е. перевести в отмеченное состояние. Мьютексы — это один **из вариантов семафорных** механизмов для организации взаимного исключения. Они реализованы во многих ОС, их **основное назначение** — организация **взаимного исключения для потоков** из одного и того же или из разных процессов.



Цель использования мьютексов — **защита данных** от повреждения в результате **асинхронных изменений** (состояние гонки), однако могут порождаться другие проблемы — например взаимная блокировка (клинч).

Основные типы сокетов

- **Поточный** - обеспечивает двухсторонний, последовательный, надежный, и недублированный поток данных без определенных границ. Тип сокета - SOCK_STREAM, в домене Интернета он использует протокол **TCP**.
- **Датаграммный**- поддерживает двухсторонний поток сообщений. Приложение, использующее такие сокеты, может получать сообщения в порядке, отличном от последовательности, в которой эти сообщения посылались. Тип сокета - SOCK_DGRAM, в домене Интернета он использует протокол **UDP**.
- **Сокет последовательных пакетов** - обеспечивает двухсторонний, последовательный, надежный обмен датаграммами фиксированной максимальной длины. Тип сокета - SOCK_SEQPACKET. Для этого типа сокета не существует специального протокола.
- **Простой сокет** - обеспечивает доступ к основным протоколам связи.

Недостатки процессов

- Хотя процессы являются строительными блоками распределенных систем, практика показывает, что дробления на **процессы**, предоставляемого операционными системами, на базе которых строятся распределенные системы, **недостаточно**.
- Вместо этого оказывается, что наличие более тонкого дробления в форме нескольких ***потоков выполнения (threads) на процесс значительно упрощает построение*** распределенных приложений и позволяет добиться лучшей производительности.

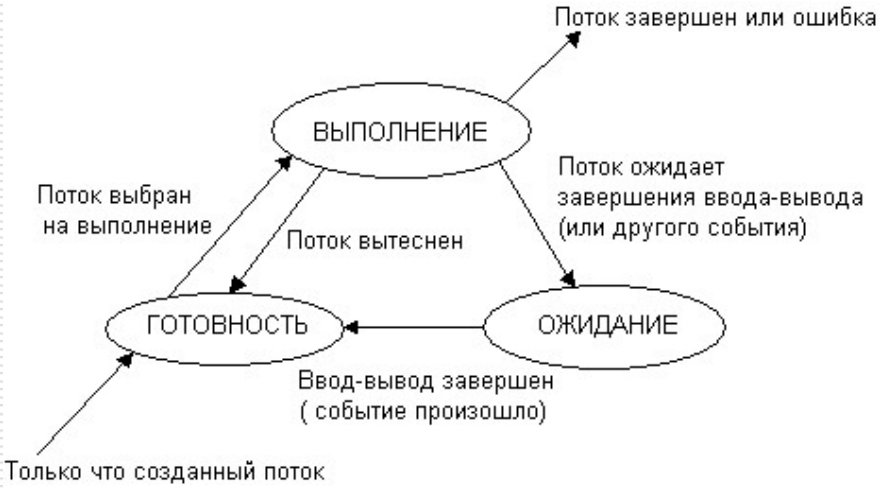
Понятие потока выполнения

- **Основная причина** использования потоков заключается в том, что 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.

Преимущества потоков выполнения

1. **Возможность использования** параллельными процессами **единого адресного пространства и всех имеющихся данных**. Эта возможность играет весьма важную роль для тех приложений, которым не подходит использование нескольких процессов (с их раздельными адресными пространствами).
2. 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.
3. 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** во времени, ускоряя работу приложения.
4. И наконец, потоки весьма полезны для систем, имеющих **несколько** центральных процессоров, где есть реальная возможность параллельных вычислений.

Процессы и потоки



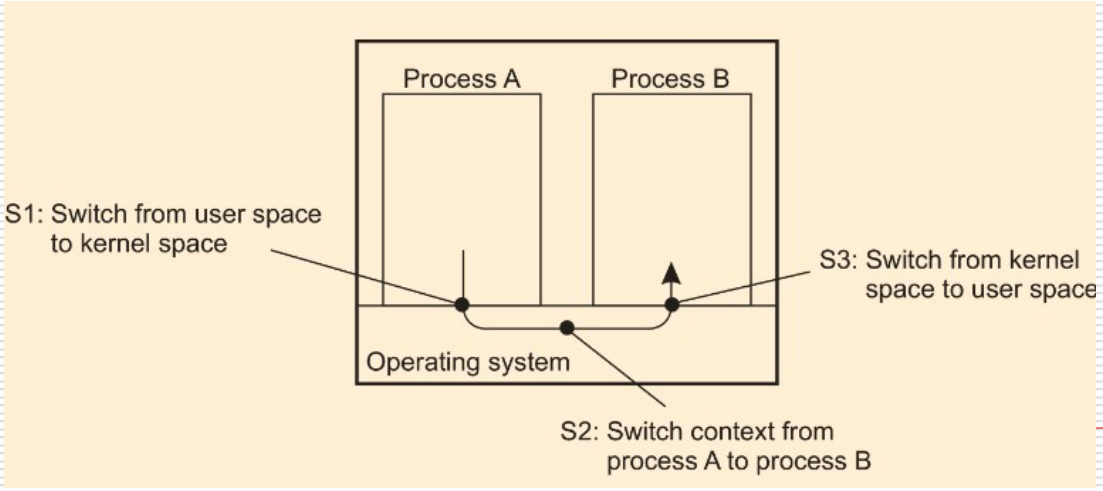
- **Создание потоков требует от ОС меньших накладных расходов, чем процессов.**
- **Все потоки одного процесса всегда принадлежат одному приложению, поэтому ОС изолирует потоки в гораздо меньшей степени,**
- **Поскольку каждый поток может иметь доступ к любому виртуальному адресу процесса, один поток может использовать стек другого потока.**

- **Между потоками одного процесса** нет **полной защиты**, потому что, во-первых, это невозможно, а во-вторых, не нужно.
- Чтобы организовать взаимодействие и обмен данными, потокам вовсе не требуется обращаться к ОС, им **достаточно использовать общую память** — один поток записывает данные, а другой читает их.
- С другой стороны, **потоки разных процессов** по-прежнему хорошо **защищены друг от друга**.

Потоки в локальных системах

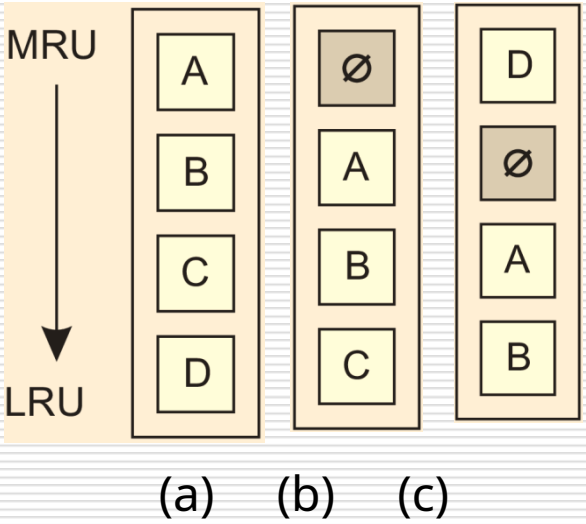
Межпроцессное взаимодействие

- **Многопроцессная реализация** часто используется при построении больших приложений. Подобные приложения часто разрабатываются в виде наборов совместно работающих программ, каждая из которых выполняется **отдельным процессом**.
- Этот подход типичен для среды UNIX. Кооперация между программами реализуется в виде **межпроцессного взаимодействия** (через механизмы IPC – Inter Process Communication).
- Поскольку IPC требует **вмешательства в ядро**, процесс обычно вынужден **сначала** переключиться из пользовательского режима **в режим ядра** (точка *S1 на* рисунке). Это требует изменения карты памяти в блоке MMU, а также сброса буфера TLB.
- В **ядре происходит переключение контекста процесса** (точка S2),



после чего второй процесс может быть активизирован очередным переключением **из режима ядра в пользовательский режим** (точка *S3).* *Последнее переключение* вновь потребует изменения карты памяти в блоке MMU и буфера сброса TLB.

Влияние переключения контекстов процессов на состояние кэш-памяти



- Рассмотрим состояние кэша процессора:
- a) состояние кэша перед прерыванием, блок D является кандидатом на **удаление** из кэша при возникновении прерывания (b).
- В дальнейшем, этот блок может опять понадобиться, что приведет его **возврату** в кэш (c).
- В результате будет **разрушена оптимальная структура кэша**, что повлечет **замедление** работы приложения.

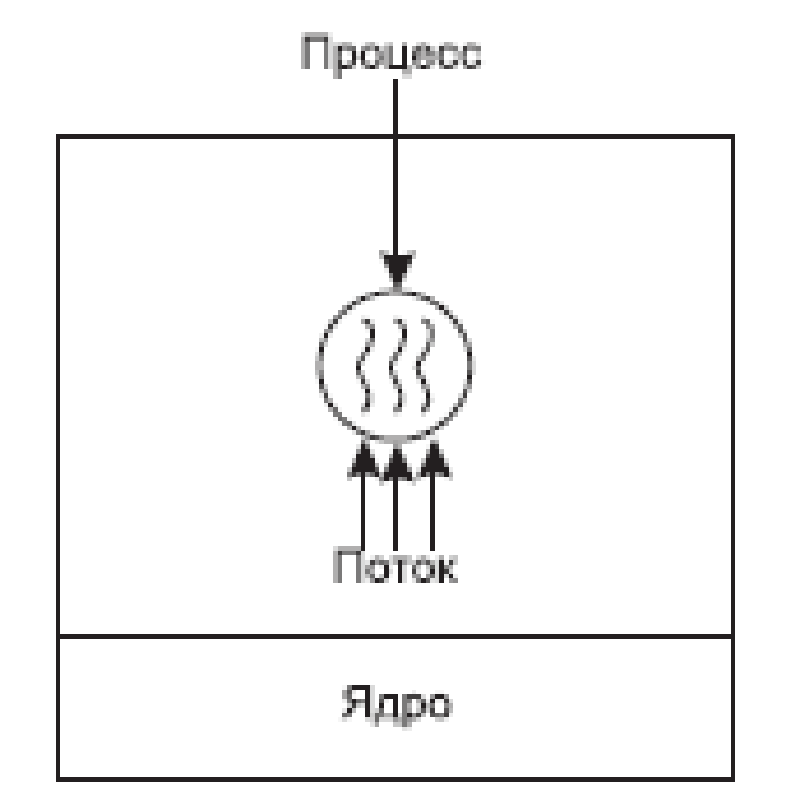
Потоки выполнения в нераспределенных системах

- Имеется **особая причина** использовать потоки выполнения:
  - многие приложения просто **легче разрабатывать**, структурировав их в виде набора взаимосвязанных потоков выполнения.
  - Например, в случае текстового редактора в отдельные потоки можно выделить обработку ввода пользователя, проверку орфографии и грамматики, оформление внешнего вида документа, создание индекса и т. п.

Реализация пакетов потоков выполнения

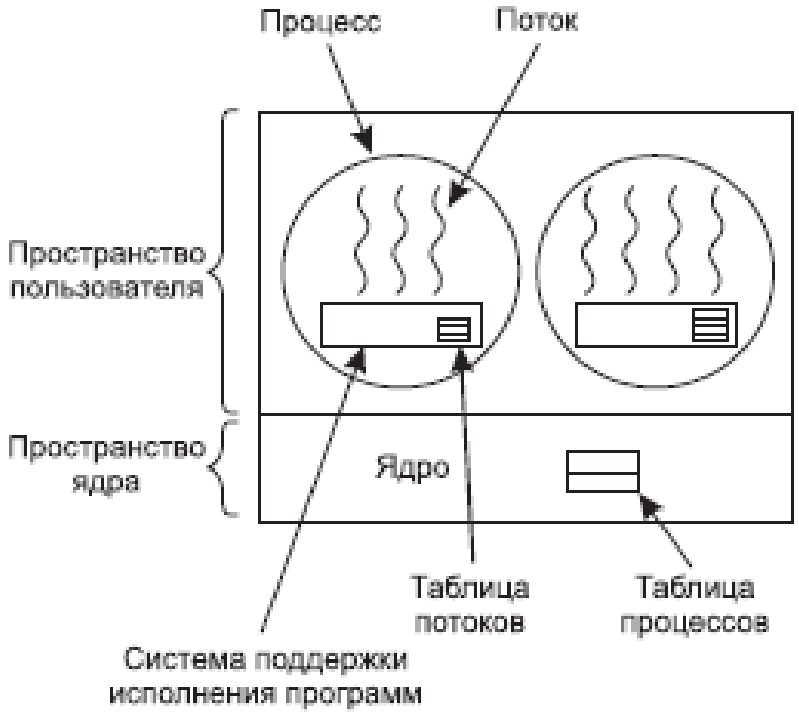
- Потоки выполнения обычно существуют в виде **пакетов потоков**.
- Подобные **пакеты содержат механизмы** для создания и уничтожения потоков, а также для работы с переменными синхронизации, такими как мьютексы и условные переменные.
- Существует два основных подхода к реализации пакетов для потоков выполнения.
  - Первый из них состоит **в создании библиотеки для работы с потоками** выполнения, выполняющейся исключительно в режиме пользователя.
  - Второй подход предполагает, что **за потоки выполнения отвечает** (и управляет ими) **ядро**.

Модель отображения потоков на процессы многие-к-одному



- В модели Many-to-one threading - несколько потоков отображаются на один планируемый процесс.
- В результате при использовании системного вызова блокирующего данный процесс **блокируются все потоки** связанные с данным процессом.

Преимущества потоков выполнения на пользовательском уровне (1)



- Библиотека для работы с потоками выполнения на пользовательском уровне имеет множество преимуществ.
- **Во-первых, дешевле** обходится создание и уничтожение потоков выполнения.
  - Поскольку все управление потоками реализуется в адресном пространстве пользователя, стоимость создания потока выполнения определяется в первую очередь **затратами на память**, выделяемую для создания стека под поток.
  - Аналогично и **уничтожение потока** выполнения в основном состоит в освобождении памяти, задействованной под стек, после того как надобность в потоке отпадает. Обе операции достаточно дешевы.

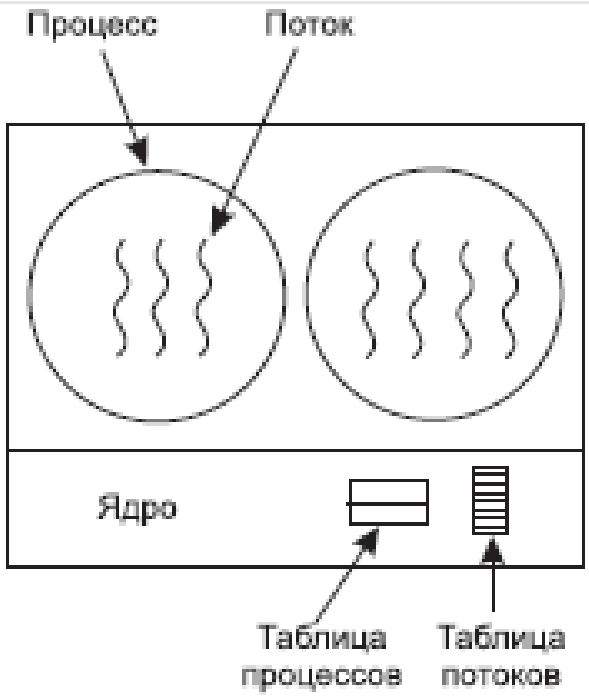
Преимущества потоков выполнения на пользовательском уровне (2)

- **Второе преимущество** потоков выполнения на пользовательском уровне состоит в том, что **переключение контекста** требует всего **нескольких** инструкций.
  - В основном **в сохранении** и последующем восстановлении сохраненных значений при переключении с потока на поток **нуждаются исключительно** в операциях со значениями **регистров процессора**.
  - Нет необходимости **изменять карты памяти**, сбрасывать **буфер TLB**, контролировать **загрузку** процессора и т. д.
  - **Переключение контекста** потоков выполнения производится при необходимости в **синхронизации двух потоков**, например, при обращении к секции совместно используемых данных.

Недостатки потоков пользовательского уровня

- Главный недостаток потоков пользовательского уровня связан с развертыванием **модели** потоковой передачи «**многие к одному**»:
  - несколько потоков отображаются на одну планируемую сущность.
  - Как следствие, **системный вызов блокировки** немедленно **заблокирует весь процесс**, которому принадлежит поток, а значит, **и все другие потоки** в этом процессе.
- Потоки особенно полезны для структурирования больших приложений на части, которые могут логически выполняться одновременно.
- В этом случае блокировка ввода-вывода не должна препятствовать выполнению других частей в это время. Для таких приложений **потоки пользовательского** уровня **бесполезны**.
- Эти проблемы можно в основном обойти, реализовав **потоки в ядре** операционной системы, что приведет к так называемой **однозначной модели потоковой передачи**, в которой каждый поток является планируемым объектом.

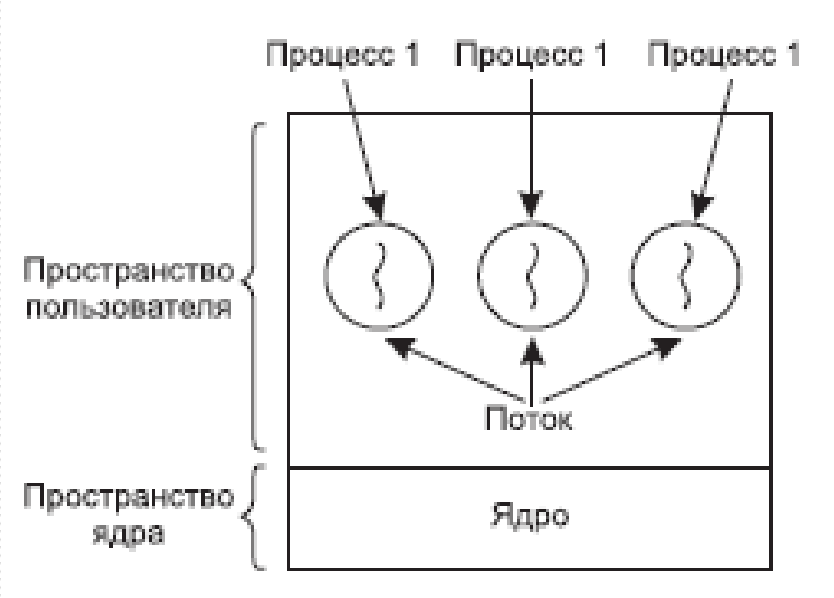
Реализация потоков в ядре



- Для потоков реализуемых на уровне ядра вся информация о потоках аналогична той, что используется для пользовательских потоков.
- При работе в режиме ядра создание и уничтожение потоков требует **более существенных затрат.**
  - Хотя потоки на уровне ядра могут решить многие проблемы, но их **главный недостаток** в весьма существенных **затратах на реализацию системных вызовов**, поэтому если потоки создаются/удаляются достаточно часто, то это влечет за собой существенные издержки.
  - Другой проблемой являются **сигналы**. Сигналы предназначены для процессов, а не для потоков.

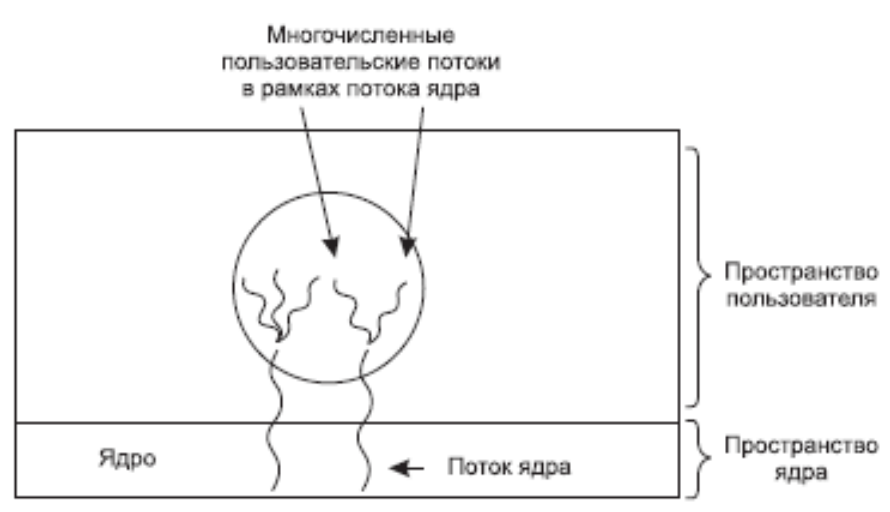
- Однако в свете того факта, что **производительность переключения контекста** обычно определяется **неэффективным использованием кешей** памяти, а не различием между потоковой моделью «многие к одному» или «один к одному», многие операционные системы сейчас предлагают последнюю модель хотя бы из-за ее простоты.

Модель один-к-одному



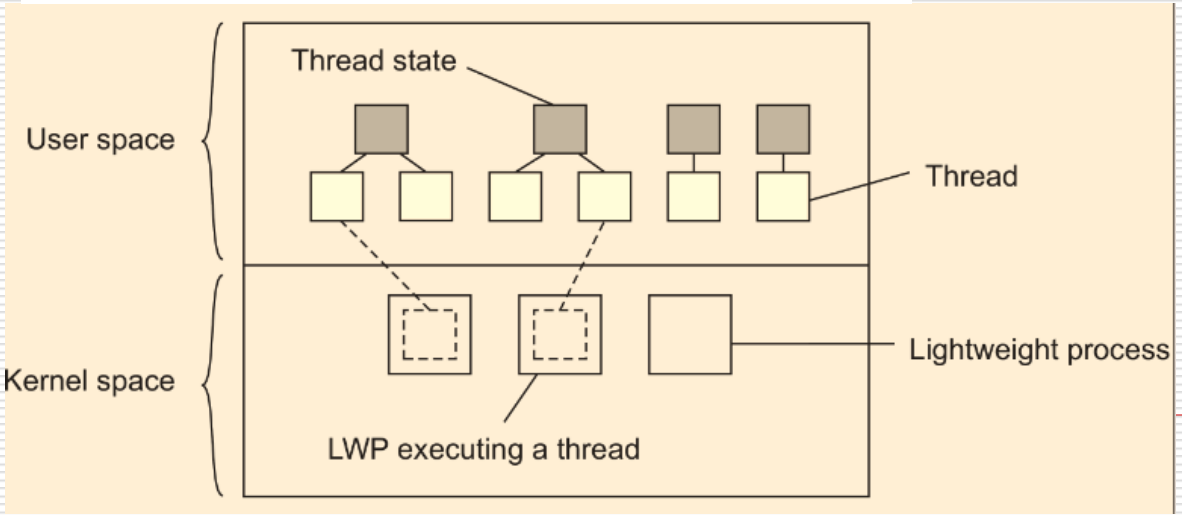
- В модели One-to-one threading – каждый процесс имеет только один поток, который диспетчируется независимо от других.
- Достоинством этой модели является возможность выполнения операций над потоками без необходимости обращения к ядру.

Гибридная модель реализация потоков



В этом случае используются потоки **на уровне ядра** и **один** или несколько потоков на **пользовательском уровне**.

Эта модель обладает максимальной гибкостью



Использование потоков против применения группы конкурирующих процессов

- Применение потоков является способом **одновременного** и **параллельного** исполнения в рамках **одного приложения**.
- На практике, часто можно встреть реализации когда приложения строятся как **коллекция параллельно работающих процессов**, объединяемых с помощью средств межпроцессного взаимодействия предлагаемых операционными системами.
  - Примером применения такого подхода является реализация веб-сервера Apache, который по-умолчанию стартует как пятерка процессов, предназначенных для обслуживания поступающих запросов.
  - **Каждый процесс является *однопоточной* реализацией** сервера, способной взаимодействовать с другими экземплярами с помощью стандартных средств ОС.
- Использование процессов вместо потоков имеет **одно важное преимущество** – обеспечивается **разделение пространств** данных процессов.

  Разделение процессов и их пространств данных обеспечивается аппаратными средствами процессоров.
- Это преимущество нельзя недооценивать, так как при использовании потоков вся забота о параллельном доступе к разделяемым данным **ложиться на разработчика** многопоточного приложения.

Потоки в распределенных системах

Почему потоки применяют в РИС

- **Важнейшим свойством потоков** выполнения является возможность выполнения системных блокирующих вызовов, **без блокировки всего процесса** в рамках которого работает поток.
- Это свойство является **привлекательным** для **распределенных систем**, что делает их более приспособленными к быстрым коммуникациям и обеспечивает одновременное поддержание нескольких логических соединений между компонентами РС.

Многопоточные клиенты

- Для обеспечения высокой степени прозрачности клиенты должны обладать поддержкой работы в многопоточном режиме.
- Например Web-браузер, для сокрытия достаточно больших величин задержек должен обеспечивать возможность поддержания **нескольких одновременных соединений** с веб-вервером для одновременной закачки содержимого различных частей веб страниц.
- Кроме того современные браузеры могут одновременного выполнять **доставку нескольких веб страниц** с **различных серверов**, что также требует **многопоточности.**

Эффективность использования многопоточности на современных архитектурах

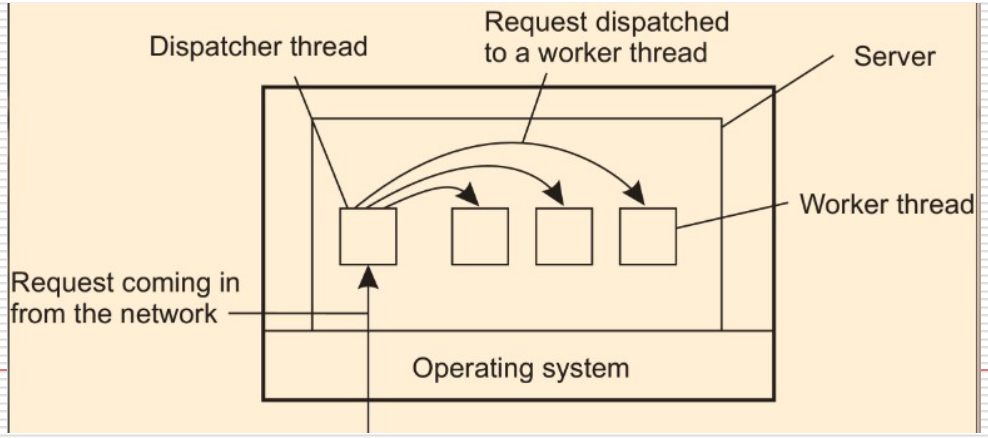
- В исследовании, посвященном тому, насколько многопоточность задействует многоядерный процессор, Блейк и др. [2010] рассматривал выполнение различных приложений на современных архитектурах.
- Чтобы правильно измерить степень использования многоядерного процессора, Блейк и др. использовал метрику, известную как **параллелизм на уровне потоков (TLP)**.
  - Пусть c_i обозначает долю времени, в течение которой ровно i потоков выполняется одновременно. Тогда параллелизм на уровне потоков определяется как:

$$TLP = \frac{\sum_{i=1}^{N} i \cdot c_i}{1 - c_0}$$

  - где N - макс во потоков, которые (могут) выполняться одновременно.
- **Было выяснено**, что типичный веб-браузер имел значение **TLP** от **1,5 до 2,5,** что означает, что для эффективного использования параллелизма клиентская машина должна иметь **два или три ядра** или, аналогично, **2-3 процессора**.
- Эти результаты интересны, если учесть, что **современные веб-браузеры создают сотни потоков**, и что **десятки потоков активны одновременно** (обратите внимание, что активный поток не обязательно работает; он может быть заблокирован в ожидании завершения запроса ввода-вывода).
- Таким образом, мы видим, что **многопоточность используется для организации приложения,** но ***эта многопоточность не приводит*** к значительному **повышению производительности за счет использования оборудования**.

Многопоточные серверы

- Рассмотрим файловый сервер. Обычно файловый сервер ожидает обращений от клиентов с запросами на выполнение файловых операций.
- **Часто такой сервер** реализуется в виде **нескольких параллельно исполняемых потоков**, один из которых выполняет роль диспетчера запросов (поток диспетчер), а остальные реализуют рабочие потоки.
- Данная реализация обеспечивает необходимую **прозрачность** и производительность этого файлового сервера.



Однопоточный сервер

- Основной цикл работы файлового сервера включает в себя:
  - получение запроса,
  - изучение запроса
  - выполнение запроса вплоть до его завершения:
    - В ожидании диска сервер простаивает и не обрабатывает никакие другие запросы.
    - Следовательно, запросы от других клиентов не могут быть обработаны.
  - ожидание нового запроса
- Кроме того, если файловый сервер работает на выделенной машине, как это обычно бывает, ЦП просто бездействует, пока файловый сервер ожидает диск.
- В результате в единицу времени можно обрабатывать намного меньше запросов. Таким образом потоки позволяют получать значительную производительность, но каждый поток программируется последовательно, обычным образом.

Сервер как большой однопоточный конечный автомат

1. Когда **поступает запрос**, единственный **поток проверяет** его.
2. Если это может быть выполнено из кеша в памяти, хорошо, но если нет, поток **должен получить доступ к диску**.
3. Однако вместо выполнения операции с блокирующим диском поток планирует **асинхронную** (т. е. неблокирующую) **операцию с диском**, по окончании которой он позже будет прерван операционной системой.
4. Чтобы это сработало, поток будет **записывать статус запроса** (а именно, что у него есть отложенная операция с диском) и **продолжает проверять**, были ли какие-либо другие **входящие запросы**, требующие его внимания.
5. **После завершения отложенной операции с диском** операционная **система уведомит поток**, который затем в должное время найдет статус связанного запроса и **продолжит его обработку**.
6. В конце концов, **ответ будет отправлен** исходному клиенту, **снова** с использованием **неблокирующего вызова** для отправки сообщения по сети.

- В этом дизайне **утрачена модель** «**последовательного процесса**», которая была в случаях многопоточного и однопоточного дизайна.
- Каждый раз, когда потоку необходимо выполнить операцию блокировки, ему необходимо точно записать, где он находился при обработке запроса, возможно, также сохраняя дополнительное состояние.
- Как только это будет сделано, он может начать операцию и продолжить другую работу. Другая работа означает обработку вновь поступивших запросов или постобработку запросов, для которых ранее запущенная операция завершилась. Конечно, если нет работы, поток действительно может заблокироваться.
- Процесс **работает как конечный автомат**, который получает событие и затем реагирует на него, в зависимости от того, что в нем находится.

Три способа построения сервера

Модель	Характеристика
Многопоточная	Параллелизм, блокировка системных вызовов
Однопоточный процесс	Отсутствие параллелизма, блокировка системных вызовов
Конечный автомат	Параллелизм, блокировка системных вызовов отсутствует

- **Многопоточная модель**. Блокирующие системные вызовы **упрощают программирование**, поскольку они выглядят как обычные вызовы процедур. Кроме того, **использование нескольких потоков** обеспечивает **параллелизм** и, следовательно, **повышение производительности**.
- **Однопоточный сервер** сохраняет **легкость и простоту блокировки системных вызовов**, но может **серьезно снизить производительность** с точки зрения количества запросов, которые могут быть обработаны за единицу времени.
- Подход **с конечным автоматом** обеспечивает **высокую производительность** за счет **параллелизма**, но использует **неблокирующие** вызовы, которые обычно **сложно программировать** и, следовательно, поддерживать.

Спасибо за внимание !