Процессы и потоки

Системное и прикладное программное обеспечение

Малышенко Владислав Викторович

Содержание курса

- Процессы
  - Модель процесса
  - Создание, завершение процесса
  - Иерархии, состояния, реализации процессов
- Потоки
  - Применение потоков
  - Классическая модель потоков
  - Реализации потоков
- Взаимодействие процессов
- Планирование
- Задачи взаимодействия процессов

Процесс

Процесс – абстрактное понятие, описывающие работу программы.

В современных ОС многозадачность реализована за счет предоставления пользовательской программе процессора на несколько миллисекунд. При условии чередования использования процессора между программами.

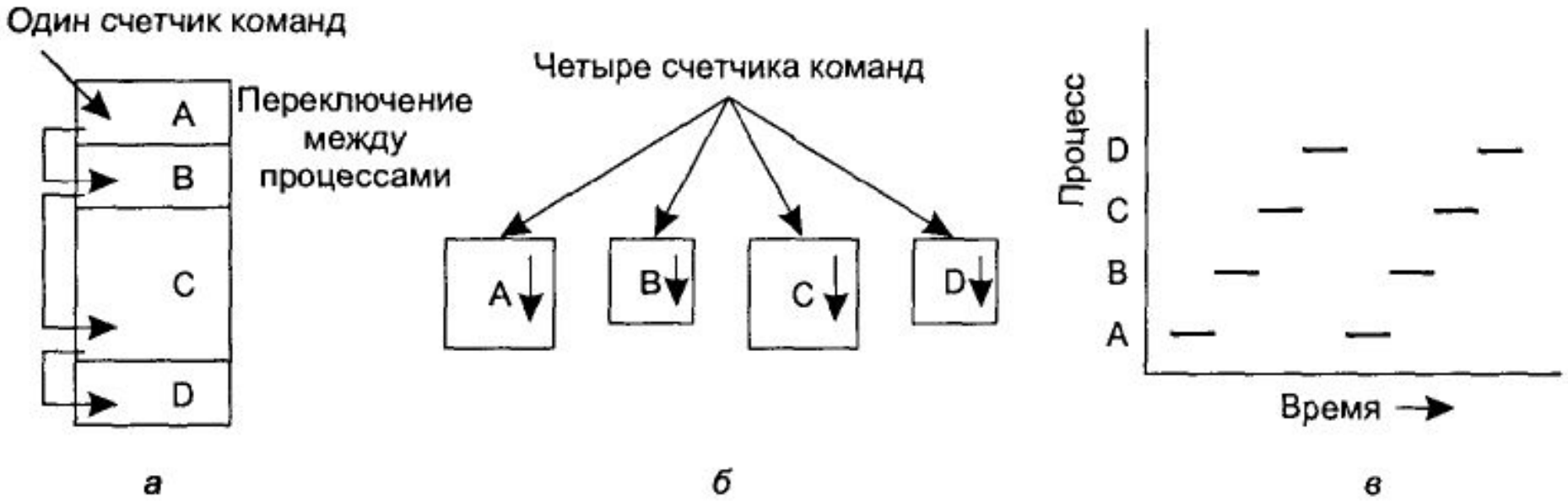
Модель процесса

Все ПО исполняемое на компьютере, а иногда и операционная система, организовано в виде **последовательных процессов**.

Процессом является выполняемая программа, включая:

- текущие значения счетчиков команд
- текущие значения регистров
- текущие значения переменных

Модель процесса (2)



а *б* *в*

Четыре программы, работающие в многозадачном режиме.

Концептуальная модель четырех независимых друг от друга последовательных процессов.

В отдельно взятый момент времени активна только одна программа.

Модель процесса (3)

- Центральный процессор переключается между процессами, следовательно, скорость вычислений процесса всегда будет разной.
- Процессы не должны программироваться с жестко заданным временем выполнения.
- Планирование процессов.
- Программа и процесс понятия схожие, но разные!
- Если программа запущенная дважды, то ею заняты два процесса.

Создание процесса

В универсальных системах определенные способы создания и прекращения процессов по мере необходимости.

Способы создания процессов:

1. Инициализация системы
2. Выполнение работающим процессом системного запроса на создание процесса
3. Запрос пользователя на создание процесса
4. Инициализация пакетного задания

Создание процесса. Инициализация операционной системы.

При загрузке ОС создается несколько процессов.

- Процессы, обеспечивающие взаимодействие с пользователями и выполнение заданий, являются **высокоприоритетными** процессами**.**
- Процессы, не связанные с конкретными пользователями, но выполняющими ряд специфических функций, являются **фоновыми** процессами (**демонами**).

Пример:

Получение электронной почты, web-новости, вывод на принтер.

Создание процесса (системный вызов)

Новый процесс формируется на основании системного запроса от текущего процесса.

В роли текущего процесса может выступать:

- Процесс, запущенный пользователем;
- Системный процесс;
- Процесс, инициализированный клавиатурой или мышью;
- Процесс, управляющий пакетами.

Создание процесса (системный вызов)

В UNIX существует только один системный запрос: **fork** (**ветвление**). Этот запрос создает дубликат вызываемого процесса.

В Windows же вызов всего одной функции **CreateProcess** интерфейса Win32 управляет и созданием процесса, и запуском в нем нужной программы.

Кроме CreateProcess в Win32 есть около 100 функций для управления процессами и их синхронизации.

Завершение процесса

Завершение процесса:

1. Обычный выход (преднамеренно);
2. Выход по ошибке (преднамеренно);
3. Выход по неисправимой ошибке (непреднамеренно);
4. Уничтожение другим процессом (непреднамеренно).

После окончания работы процесс генерирует системный запрос на завершение работы. В UNIX этот системный запрос – **exit**, а в Windows – **ExitProcess**.

Программы, рассчитанные на работу с экраном, также поддерживают преднамеренное завершение работы.

Иерархия процессов

В некоторых системах родительский и дочерний процессы остаются связанными между собой определенным образом.

Дочерний процесс также может, в свою очередь, создавать процессы, формируя иерархию процессов. В UNIX процесс, все его «дети» и дальнейшие потомки образуют группу процессов.

Сигнал, посылаемый пользователем с клавиатуры, доставляется всем членам группы, взаимодействующим с клавиатурой в данный момент.

Иерархия процессов (2). Пример.

В образе загрузки присутствует специальный процесс **init**. При запуске этот процесс считывает файл, в котором находится информация о количестве терминалов. Затем процесс разветвляется таким образом, чтобы каждому терминалу соответствовал один процесс.

Процессы ждут, пока какой-нибудь пользователь не войдет в систему. Если пароль правильный, процесс входа в систему запускает оболочку для обработки команд пользователя, которые, в свою очередь, могут запускать процессы. Таким образом, все процессы в системе принадлежат к единому дереву, начинающемуся с процесса init.

Иерархия процессов (3). Пример.

В Windows не существует понятия иерархии процессов, и все процессы равноправны.

Единственное, в чем проявляется что-то вроде иерархии процессов - создание процесса, в котором родительский процесс получает специальный маркер (так называемый **дескриптор**), позволяющий контролировать дочерний процесс.

Но маркер можно передать другому процессу, нарушая иерархию.

В UNIX это невозможно.

Состояние процессов

Состояние процессов

Несмотря на самостоятельность каждого процесса, наличие собственного счетчика команд и внутреннего состояния, процессам зачастую необходимо взаимодействовать с другими процессами.

Один процесс может генерировать выходную информацию, используемую другими процессами в качестве входной информации.

Пример:

Выходные данные процесса `cat` могут служить входными данными для процесса `grep`.

```
Cat chapter.txt chapter2.txt | grep tree
```

Состояние процессов (2)

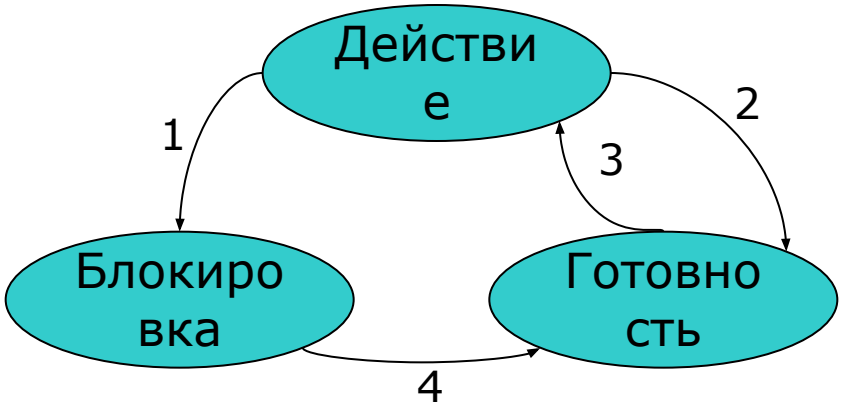
Возможны два вида блокировки процесса:

1. Процесс блокируется с точки зрения логики приложения (из-за отсутствия входных данных)
2. Процесс блокируется операционной системой (из-за отсутствия ресурсов)

Состояние процессов (диаграмма состояния)

Три возможных состояния процесса:

1. Работающий
2. Готовый к работе
3. Заблокированный



1. Процесс блокируется, ожидая входных данных
2. Планировщик выбирает другой процесс
3. Планировщик выбирает этот процесс
4. Доступны входные данные

Состояние процессов (диаграмма состояния)

Переходы 2 и 3 вызываются **планировщиком процессов**

процессы

0	1	2	…	N-3	N-2	N-1

планировщик

Реализация процессов

Для реализации модели процессов операционная система содержит **таблицу процессов.**

В таблице содержится информация – о состоянии процесса, счетчик команд, указатель стека, распределение памяти, состояние открытых файлов – необходима для переключения в состояние **готовности** или **блокировки**.

Реализация процессов (**таблица процессов**)

Управление процессом	Управление памятью	Управление файлами
Регистры	Указатель на текстовый сегмент	Корневой каталог
Счетчик команд	Указатель на сегмент данных	Рабочий каталог
Слово состояния программы	Указатель на сегмент стека	Дескрипторы файла
Указатель стека		Идентификатор пользователя
Состояние процесса		Идентификатор группы
Приоритет		
Параметры планирования		
Идентификатор процесса		
Родительский процесс		
Группа процесса		
Сигналы		
Время начала процесса		
Использованное процессорное время		
Процессорное время дочернего процесса		
Время следующего аварийного сигнала		

Реализация процессов (работа с внешними устройствами)

С каждым классом устройств ввода-вывода связана область памяти называемая **вектором прерываний**.

Вектор прерываний содержит адрес процедуры обработки прерываний.

Например: в момент прерывания диска работал пользовательский процесс 3. Содержимое счетчика команд процесса записываются в стек аппаратными средствами прерывания. Затем происходит переход по адресу, указанному в векторе прерывания диска.

Вся остальная обработка прерывания производится программным обеспечением.

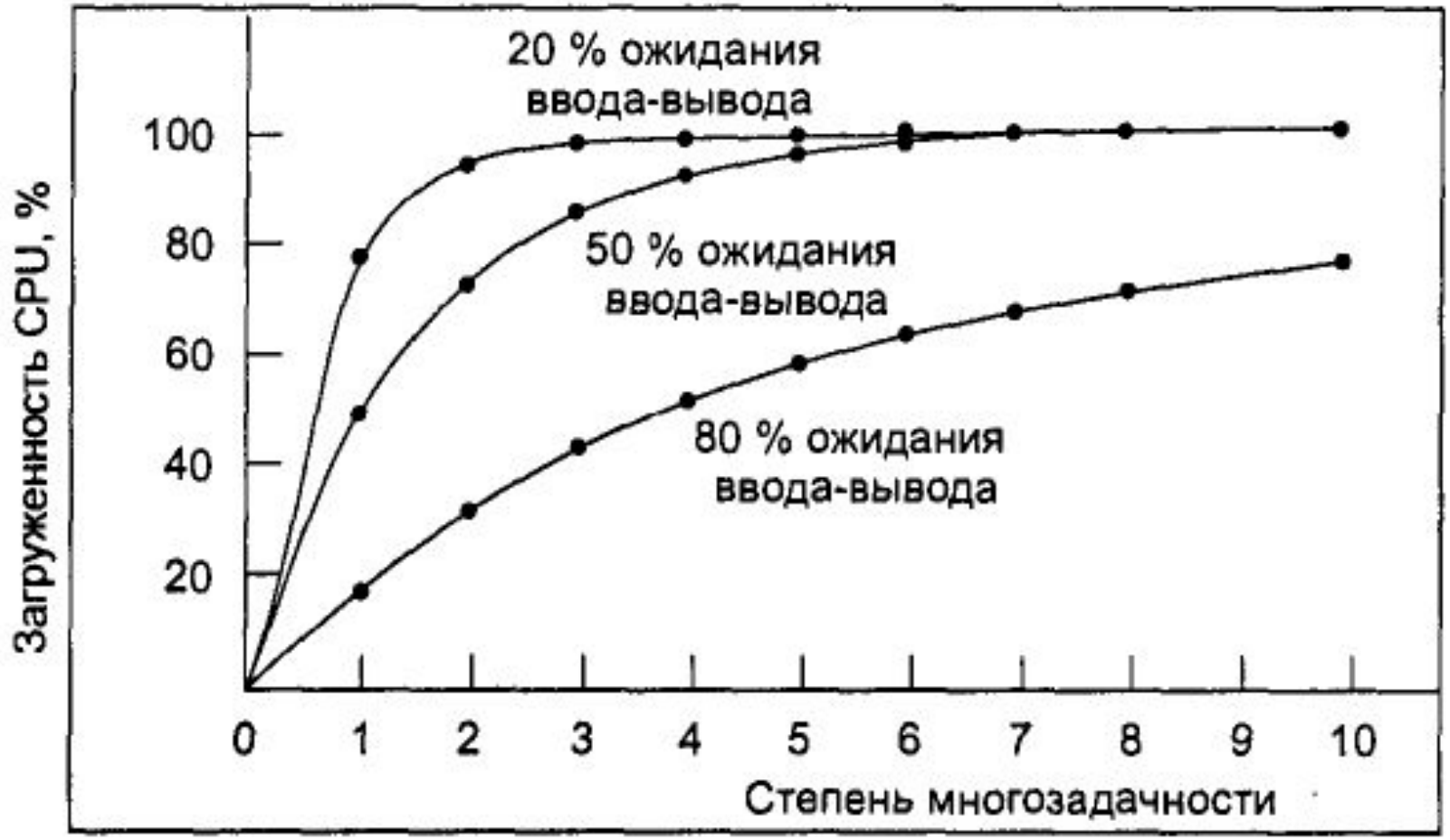
Реализация процессов (**схема обработки прерываний**)

1. Аппаратное обеспечение сохраняет в стеке счетчик команд и т. п.
2. Аппаратное обеспечение загружает новый счетчик команд из вектора прерываний
3. Процедура на ассемблере сохраняет регистры
4. Процедура на ассемблере устанавливает новый стек
5. Запускается программа обработки прерываний на C
6. Планировщик выбирает следующий процесс
7. Программа на C передает управление процедуре на ассемблере
8. Процедура на ассемблере запускает новый процесс

Моделирование многозадачности

- При использовании многозадачности повышается эффективность загрузки центрального процессора. Грубо говоря, если средний процесс выполняет вычисления только 20 % от того времени, которое он находится в памяти, то при присутствии в памяти одновременно пяти процессов центральный процессор должен быть занят все время.
- Более совершенная модель рассматривает эксплуатацию центрального процессора с точки зрения теории вероятности. Предположим, что процесс проводит часть p своего времени в ожидании завершения операции ввода-вывода. Если в памяти находится одновременно n процессов, вероятность того, что все n процессов ждут ввод-вывод, равна p^n. Тогда степень загрузки центрального процессора будет выражаться формулой:
  - *Степень загрузки центрального процессора* $= 1 - p^n$.

Моделирование многозадачности



Моделирование многозадачности (пример)

- Предположим, что компьютер имеет 2 Гб памяти, 1 Гб отдано операционной системе, а каждая программа пользователя занимает по 256 Мбайт.
- При таких заданных размерах одновременно можно загрузить в память четыре пользовательские программы. При 80 % времени на ожидание ввода-вывода в среднем мы получим загруженность процессора равной $1-0{,}8^4$, или около 60 %.
- Добавление еще 1 Гб памяти позволит системе повысить степень многозадачности от четырех до восьми и таким образом повысить степень загрузки процессора до 83 %. Другими словами, дополнительные 1 Гб увеличат производительность на 33 %.
- Еще 1 Гб могли бы повысить загрузку процессора с 83 до 93 %, таким образом, увеличив производительность всего лишь на 10 %. С помощью этой модели владелец компьютера может решить, что первые 1 Гб оперативной это хорошее вложение капитала, а вторые - нет.

Анализ производительности многозадачных систем

Задача	Время поступления	Количество минут работы процесса
1	10:00	4
2	10:10	3
3	10:15	2
4	10:20	2

а

Количество процесса

	1	2	3	4
Процессор бездействует	.80	.64	.51	.41
Процессор занят	.20	.36	.49	.59
Процессор/ процесс	.20	.18	.16	.15

б

в

Поток

Потоки

В обычных операционных системах каждому процессу соответствует адресное пространство и одиночный управляющий поток. Фактически это и определяет процесс.

Тем не менее часто встречаются ситуации, в которых предпочтительно иметь несколько квазипараллельных управляющих потоков в одном адресном пространстве, как если бы они были различными процессами (однако разделяющим одно адресное пространство).

Использование потоков

Основной причиной использование потоков является выполнение большинством приложений существенного числа действий.

В случае параллельных процессов используются прерывания, таймеры и переключатели контекста.

В случае потоков придется добавить еще один элемент: возможность совместного использования параллельными объектами адресного пространства и всех содержащихся в нем данных.

Легкость создания и уничтожения потоков. На создание потока уходит примерно в 100 раз меньше времени, чем на создание процесса.

Третьим аргументом является производительность.

Использование потоков (пример)

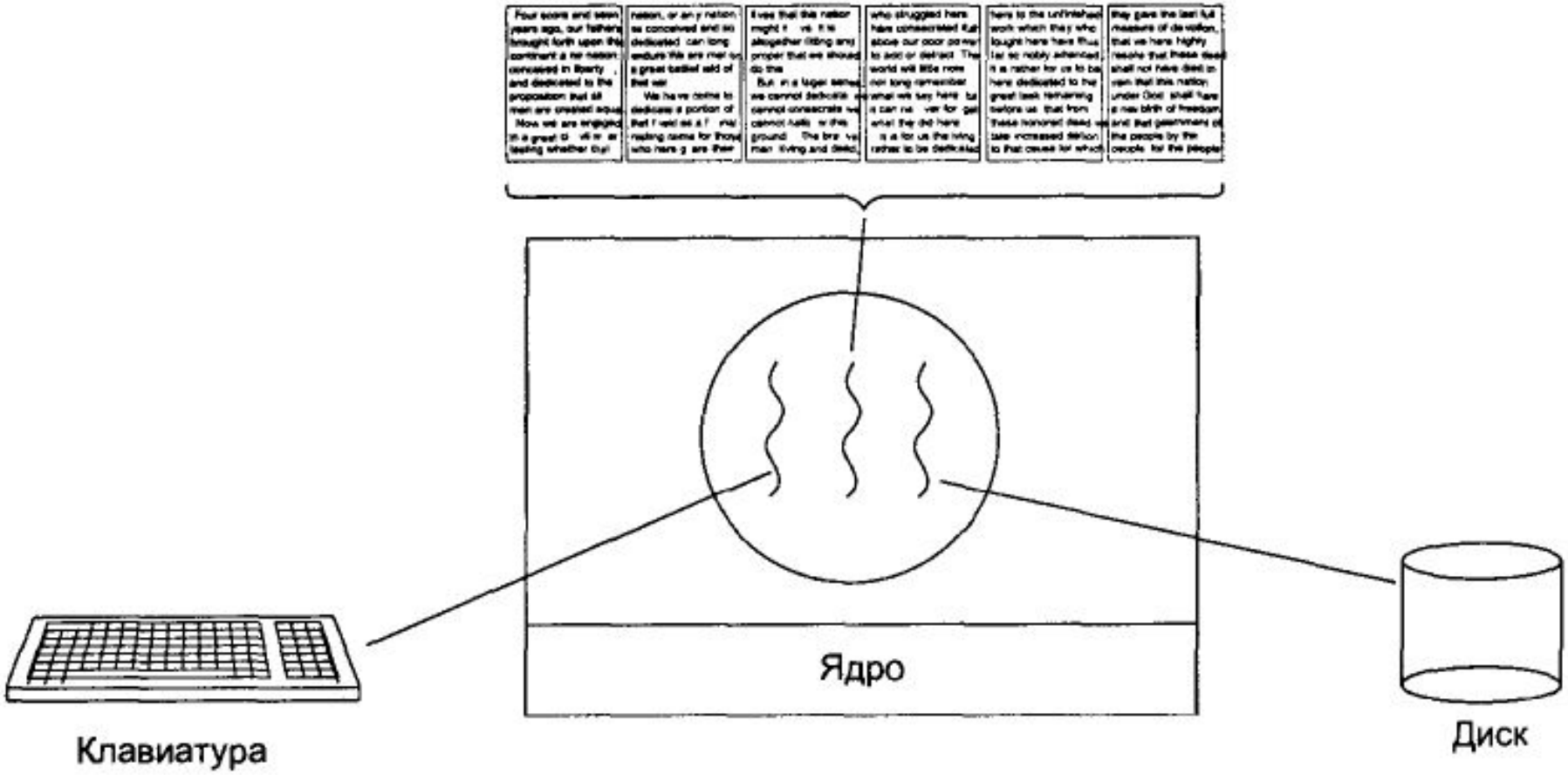
Пользователь пишет книгу.

С точки зрения автора проще всего хранить книгу в одном файле, чтобы легче было искать отдельные разделы, выполнять глобальную замену и т. п. С другой стороны, можно хранить каждую главу в отдельном файле. Но было бы крайне неудобно хранить каждый раздел и параграф в своем файле - в случае глобальных изменений пришлось бы редактировать сотни файлов.

Например, если предполагаемый стандарт xxx был утвержден только перед отправкой книги в печать, придется заменять «Черновой стандарт xxx» на «Стандарт xxx» в последнюю минуту. Эта операция делается одной командой в случае одного файла и, напротив, займет очень много времени, если придется редактировать каждый из 300 файлов, на которые разбита книга.

Теперь представьте себе, что произойдет, если пользователь удалит одно предложение на первой странице документа, в котором 800 страниц. Пользователь перечитал эту страницу и решил исправить предложение на 600-й странице. Он дает команду текстовому редактору перейти на страницу с номером 600 (например, задав поиск фразы, встречающейся только на этой странице). Текстовому редактору придется переформатировать весь документ вплоть до 600 страницы, поскольку до форматирования он не будет знать, где начинается эта страница. Это может занять довольно много времени и вряд ли обрадует пользователя.

Использование потоков (пример) (2)

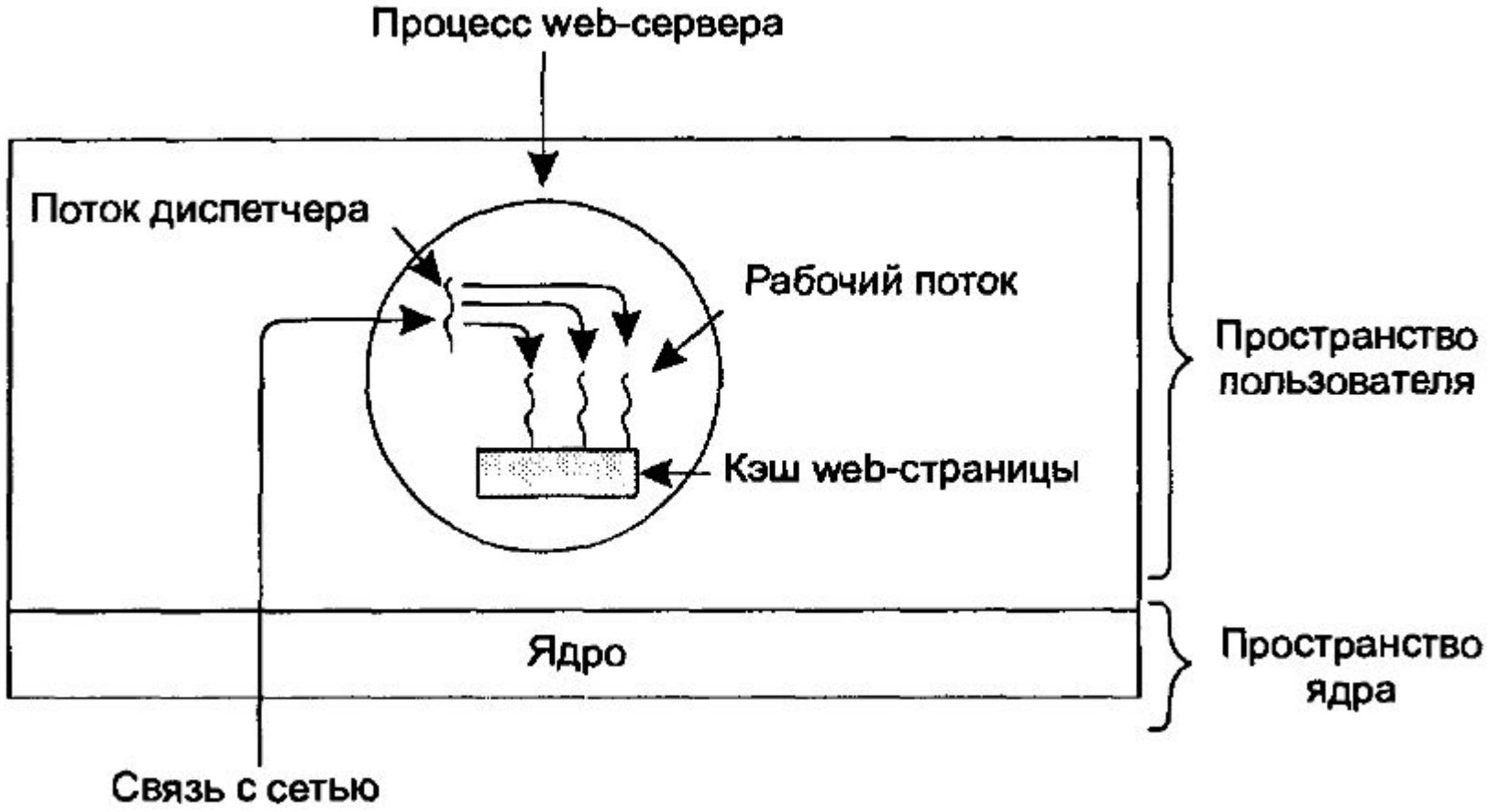


Использование потоков. Пример. WEB-сервер.

Способ организации web-сервера:

1. Один поток, называемый **диспетчером**, считывает приходящие по сети запросы;
2. После этого он находит свободный (то есть блокированный) **рабочий поток** и передает ему запрос, скажем, записывая указатель сообщения в специальное слово, связанное с каждым потоком;
3. Затем диспетчер активизирует ждущий поток, переводя его из состояния блокировки в состояние готовности.

Использование потоков. Пример. WEB-сервер.



Использование потоков. WEB-сервер. Модель сервера.

сервер создается в виде набора последовательных потоков.

Программа диспетчера состоит из бесконечного цикла, в который входит получение запроса и передача его рабочему потоку.

Программа каждого рабочего потока состоит из бесконечного цикла, включающего получение запроса от диспетчера и проверку кэша на наличие запрашиваемой страницы. При наличии страницы в кэше она отсылается клиенту, и рабочий процесс блокируется в ожидании нового запроса. При отсутствии страницы в кэше она считывается с диска, отсылается клиенту, и рабочий процесс блокируется в ожидании нового запроса.

Использование потоков. Пример (3) Сервер обработки данных.

Способы конструирования сервера:

Модель	**Характеристики**
Потоки	Параллелизм, системные запросы с блокировкой
Процесс с одним потоком	Нет параллелизма, системные запросы с блокировкой
Конечный автомат	Параллелизм, системные запросы без блокировки, прерывания

Классическая модель потоков

Модель потока

Модель процесса, базируется на двух независимых концепциях:

- группирование ресурсов;
- выполнение программы.

Модель потока. Группировка ресурсов.

Процесс можно рассматривать как способ объединения родственных ресурсов в одну группу.

У процесса есть адресное пространство, содержащее текст программы и данные, а также другие ресурсы.

Ресурсами являются:

- открытые файлы;
- дочерние процессы;
- необработанные аварийные сообщения;
- обработчики сигналов, и м. д.

Гораздо проще управлять ресурсами, объединив их в форме процесса.

Модель потока. Выполнение программы.

Процесс можно рассматривать как поток исполняемых команд или просто поток.

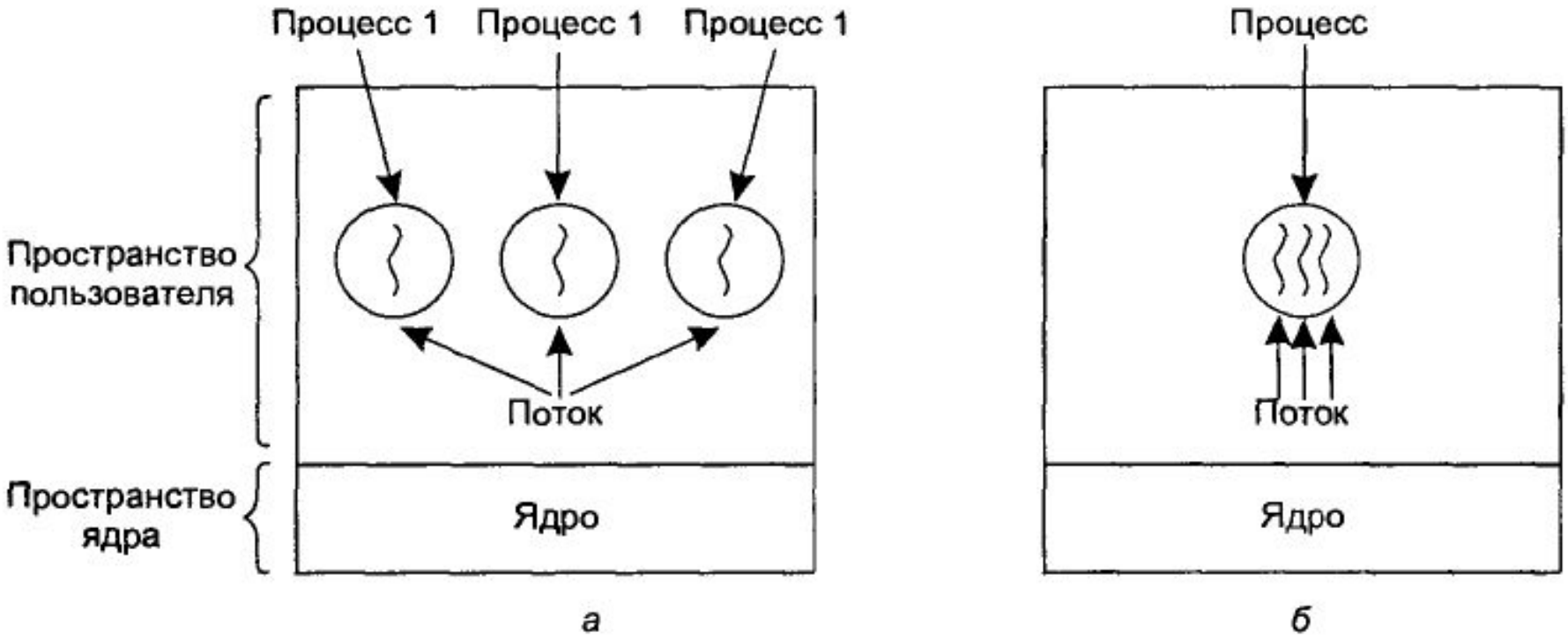
Компоненты потока:

- счетчик команд, отслеживающий порядок выполнения действий;
- регистры, в которых хранятся текущие переменные;
- стек, содержащий протокол выполнения процесса.

Отличия потока и процесса

Процессы используются для группирования ресурсов, а потоки являются объектами, поочередно исполняющимися на центральном процессоре.

Модель потока (2)



Модель потока (3)

Различные потоки в одном процессе не так независимы, как различные процессы. У всех потоков одно и то же адресное пространство, что означает совместное использование глобальных переменных.

Любой поток имеет доступ к любому адресу ячейки памяти в адресном пространстве процесса, один поток может считывать, записывать или даже стирать информацию из стека другого потока.

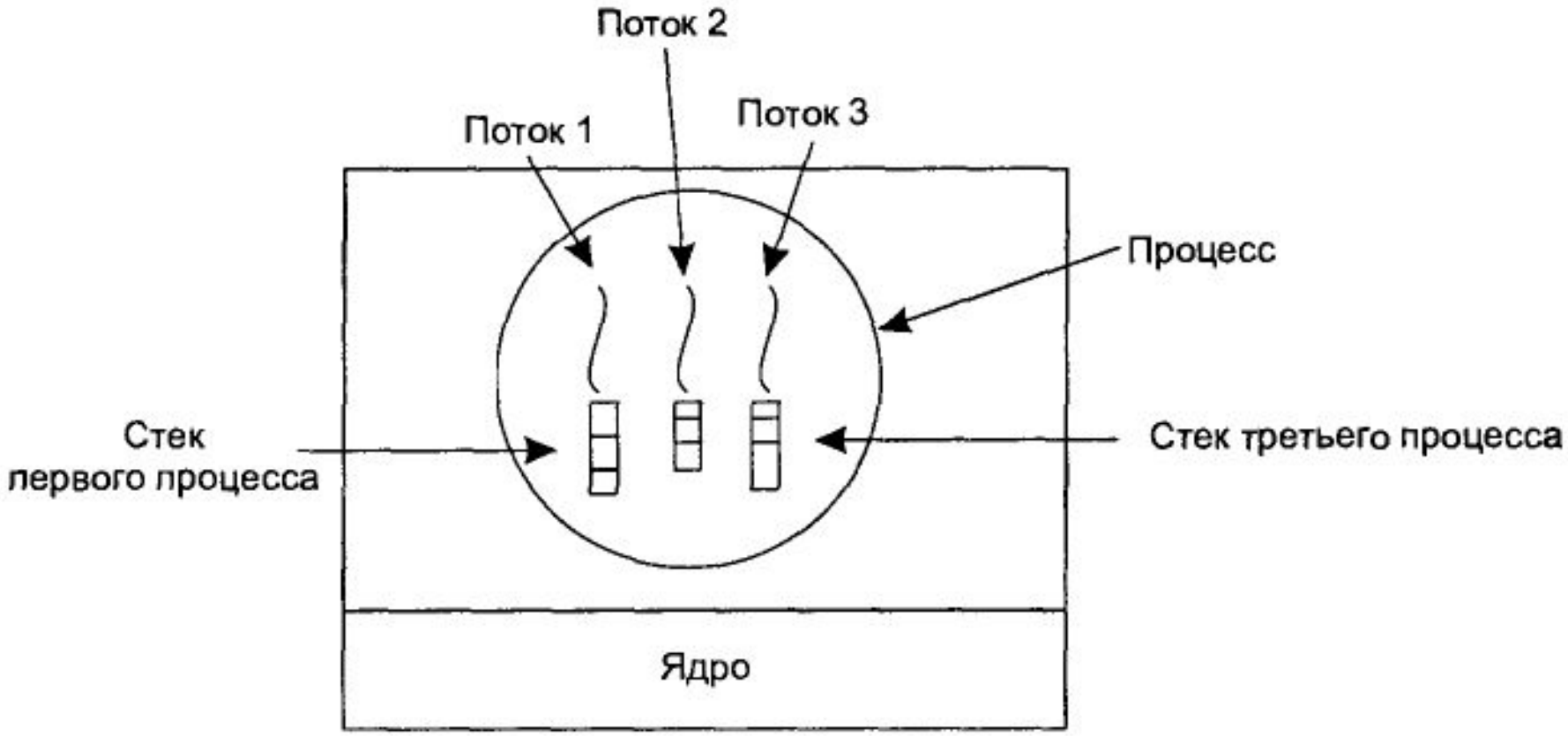
Защиты не существует, поскольку: - это невозможно и - это ненужно. В отличие от различных процессов, которые инициированы различными пользователями, один процесс всегда запущен одним пользователем, и потоки созданы, чтобы работать совместно.

Модель потока (4)

Элементы процесса	Элементы потока
Адресное пространство	Счетчик команд
Глобальные переменные	Регистры
Открытые файлы	Стек
Дочерние процессы	Состояние
Необработанные аварийные сигналы	
Сигналы и их обработчики	
Информация об использовании ресурсов	

Модель потока (5)

- Каждый поток обладает собственным стеком



Потоки в POSIX

- **IEEE standard 1003.1c** — стандарт создания переносимых многопоточных программ.
- **Пакет Pthreads**, реализует работу с потоками, поддерживается большинством UNIX-систем.
- В стандарте определено более 60 вызовов функций.
- Все потоки Pthreads имеют определенные свойства.
  - У каждого потока есть свой идентификатор, набор регистров (включая счетчик команд) и набор атрибутов, которые сохраняются в определенной структуре. Атрибуты включают размер стека, параметры планирования и другие элементы, необходимые при использовании потока.

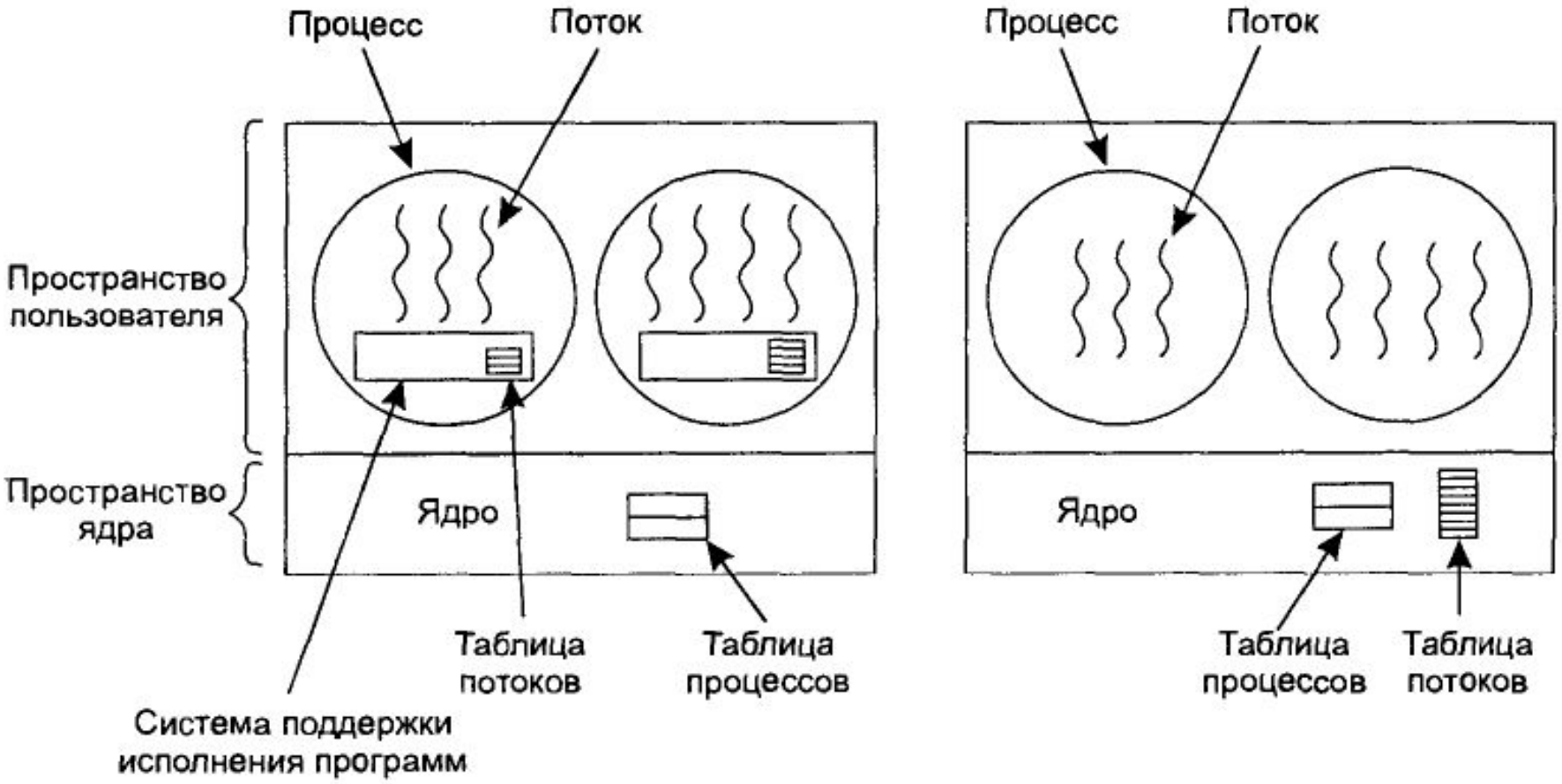
Потоки в POSIX

Функции пакета Ptheads

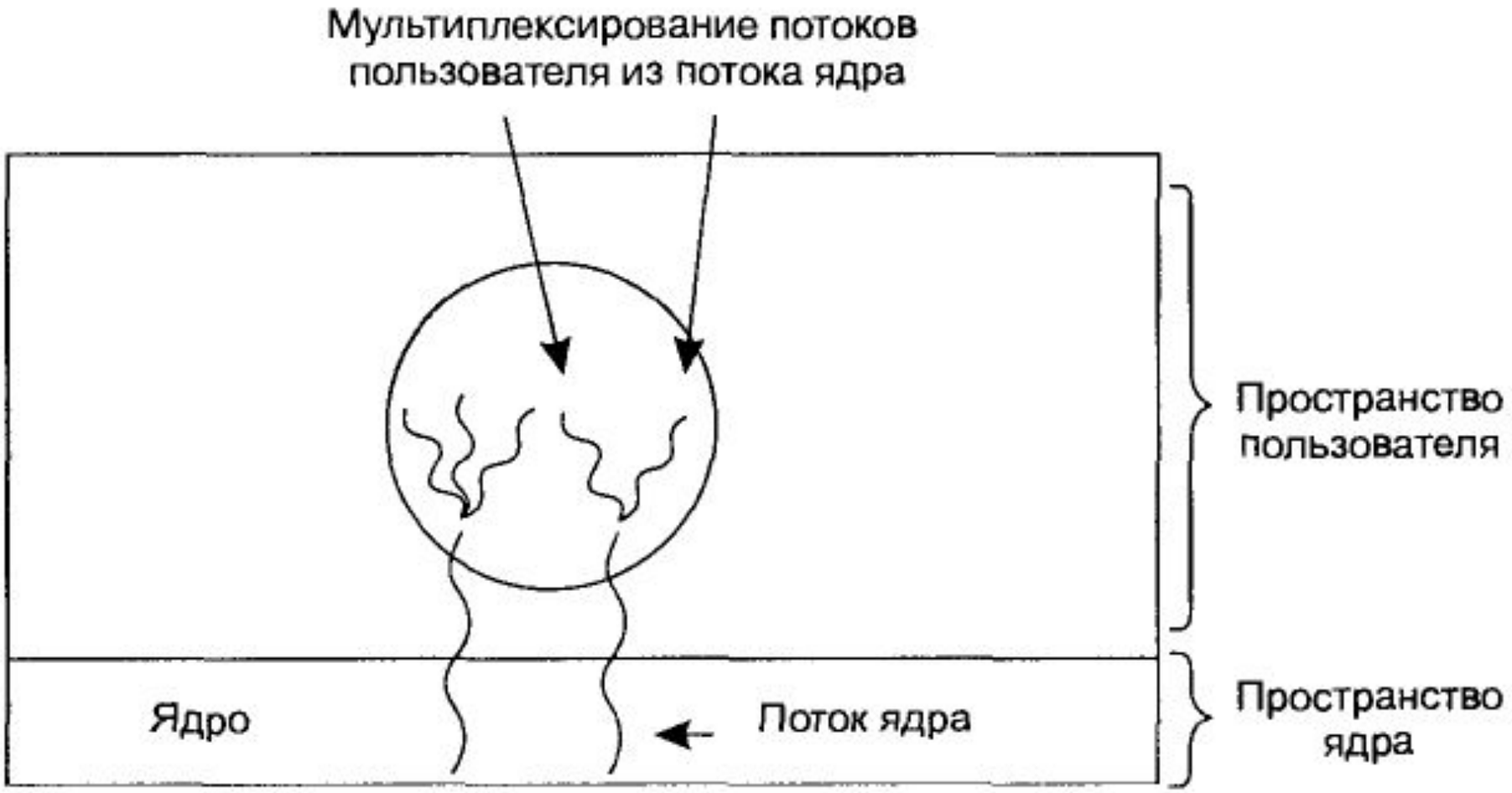
Вызовы, связанные с потоком	Описание
pthread_create	Создание нового потока
pthread_exit	Завершение работы вызвавшего потока
pthread_join	Ожидание выхода из указанного потока
pthread_yield	Освобождение центрального процессора, позволяющее выполняться другому потоку
pthread_attr_init	Создание и инициализация структуры атрибутов потока
pthread_attr_destroy	Удаление структуры атрибутов потока

Реализация потоков

Есть два основных способа реализации пакета потоков: в **пространстве пользователя** и **ядре**.



Реализация потоков (смешанная реализация)



Активация планировщика

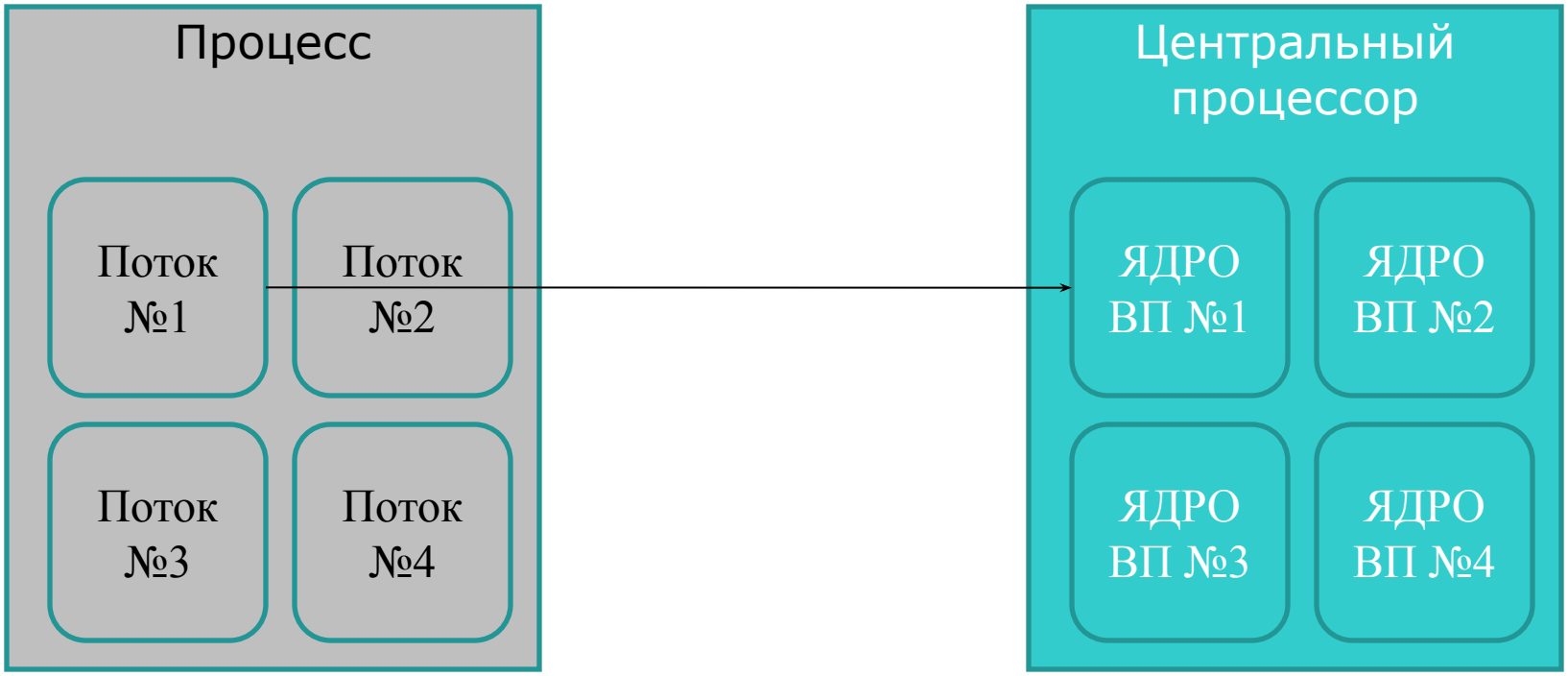
Цель активации планировщика заключается в имитации функциональных возможностей потоков на уровне ядра, но при лучшей производительности и более высокой гибкости, свойственной пакетам потоков, реализуемых в пользовательском пространстве.

- Пользовательские потоки не должны осуществлять специальные неблокирующие системные вызовы.
- Когда поток блокируется на системном вызове или на ошибке обращения к отсутствующей странице, должна оставаться возможность выполнения другого потока в рамках того же процесса.

Эффективность достигается путем уклонения от ненужных переходов между пространствами пользователя и ядра.

Активация планировщика

Виртуальные процессора



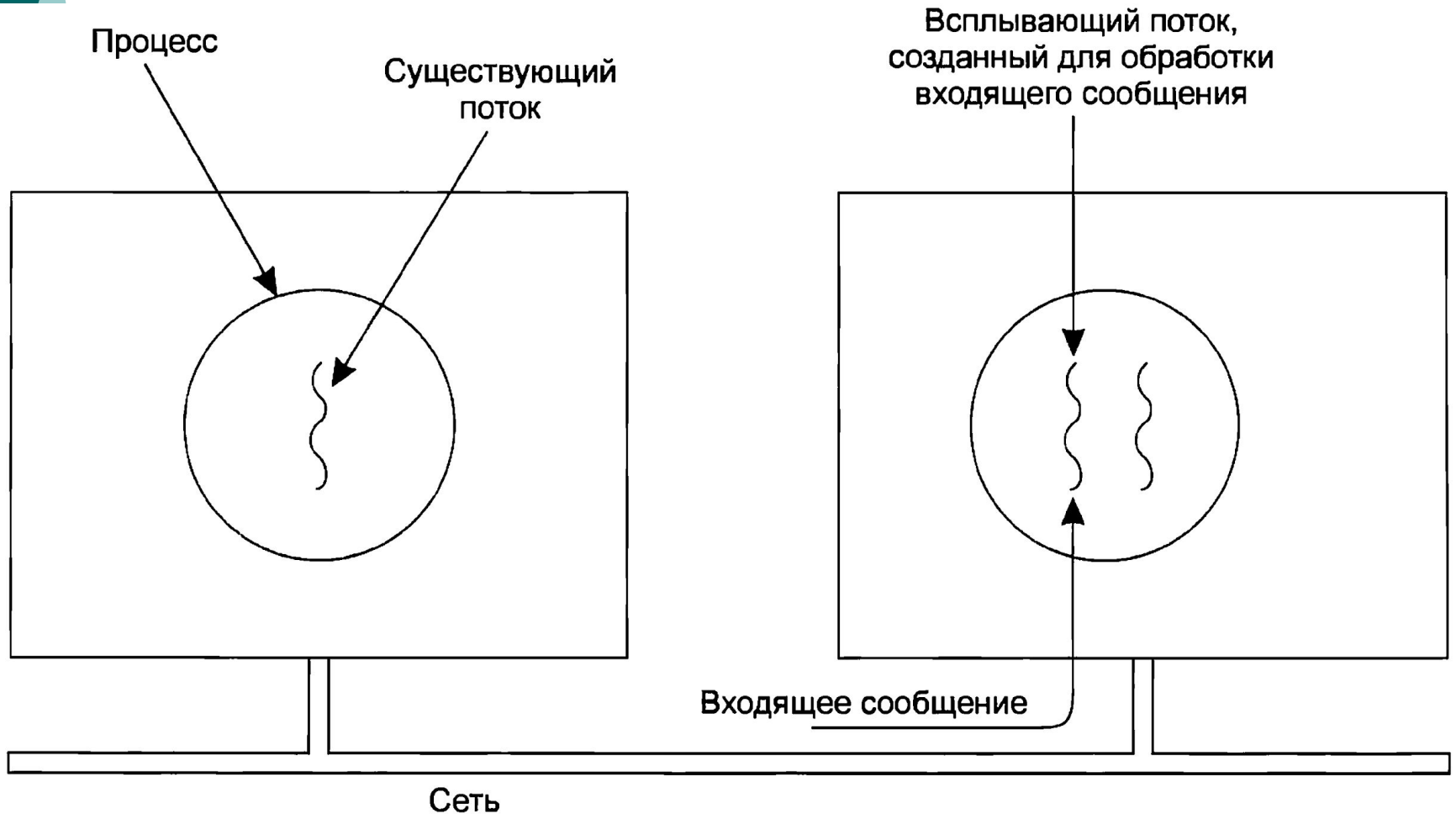
При использовании активации планировщика ядро назначает каждому процессу определенное количество виртуальных процессоров, а системе поддержки исполняемых программ (в пользовательском пространстве) разрешается распределять потоки по процессорам.

Всплывающие потоки

- Потоки часто используются в распределенных системах.
- Пример:
  - обработка входящих сообщений.
- При поступлении сообщения система создает новый поток для его обработки, называется **всплывающий поток.**
- Основное преимущество всплывающих потоков заключается в том, что они создаются заново и не имеют прошлого — никаких регистров, стека и всего остального, что должно быть восстановлено. Каждый такой поток начинается с чистого листа, и каждый их них идентичен всем остальным.
- Это позволяет создавать такие потоки довольно быстро.
- Новый поток получает сообщение для последующей обработки.
- В результате использования всплывающих потоков задержку между поступлением и началом обработки сообщения можно свести к минимуму.

Всплывающие потоки

- Создание нового потока при поступлении сообщения



Межпроцессорное взаимодействие

Межпроцессорное взаимодействие

Процессам часто бывает необходимо взаимодействовать между собой.

Например, в конвейере ядра выходные данные первого процесса должны передаваться второму по цепочке.

Проблема разбивается на три пункта.

Первое: передача информации от одного процесса другому.

Второе: контроль над деятельностью процессов: как гарантировать, что два процесса не пересекутся в критических ситуациях.

Третье: касается согласования действий процессов: если процесс А должен поставлять данные, а процесс В выводить их на печать, то процесс В должен подождать и не начинать печатать, пока не поступят данные от процесса А.

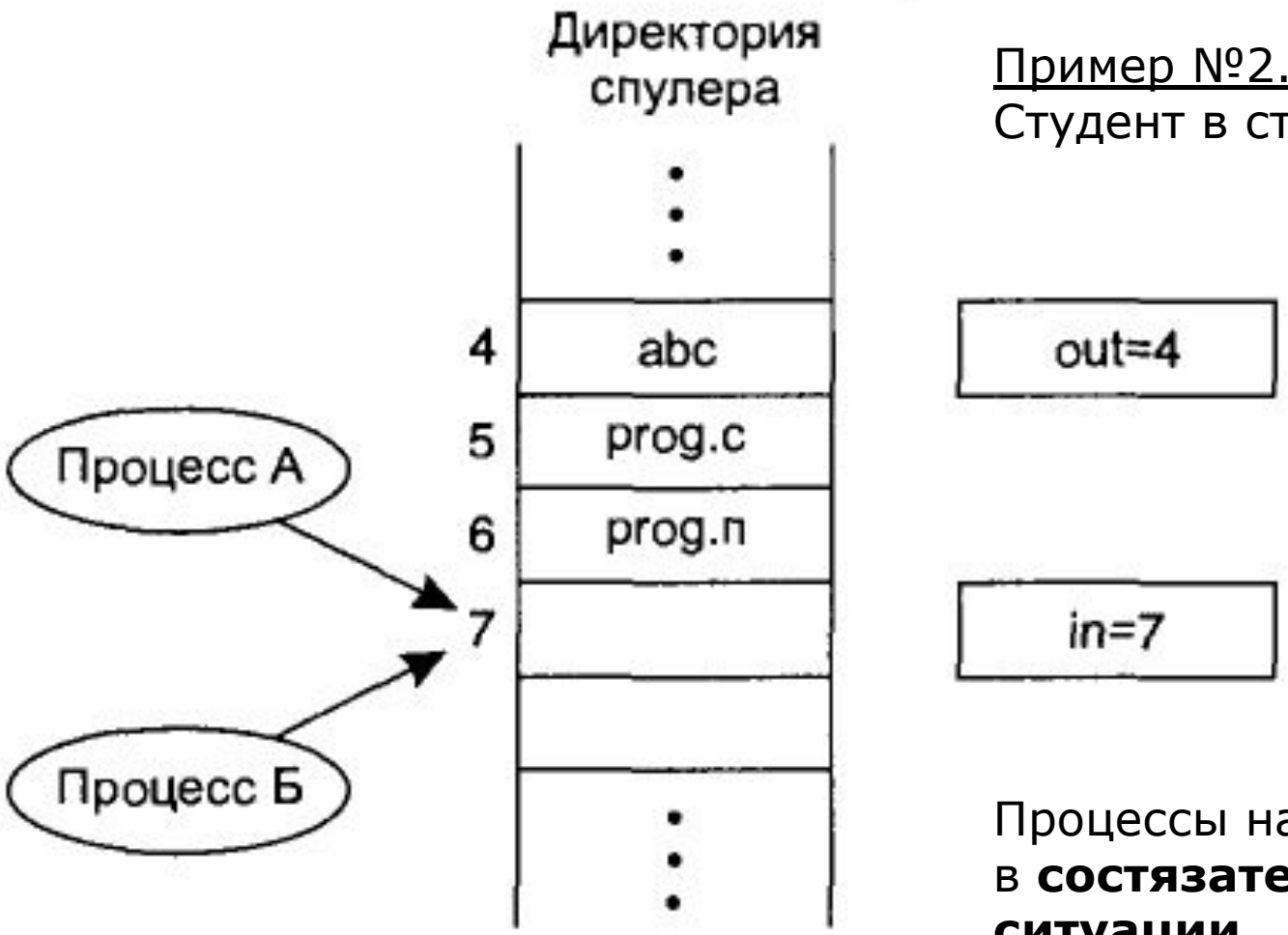
Состояние состязания

- В некоторых операционных системах процессы, работающие совместно, могут сообща использовать некое общее хранилище данных. Каждый из процессов может считывать из общего хранилища данных и записывать туда информацию. Это хранилище представляет собой участок в основной памяти или файл общего доступа.

Пример: **спулер печати**.

- Если процессу требуется вывести на печать файл, он помещает имя файла в специальный **каталог спулера**. Другой процесс, **демон печати**, периодически проверяет наличие файлов, которые нужно печатать, печатает файл и удаляет его имя из каталога.

Состояние состязания (2)



Пример №2.
Студент в столовой.

Процессы находятся в **состязательной ситуации**.

Критические области. Состязания между процессами.

Основным способом **предотвращения любой ситуации**, связанной с совместным использованием памяти, файлов и чего-либо еще, **является запрет одновременной записи и чтения** разделенных данных более чем одним процессом.

Взаимное исключение: один процесс использует разделенные данные, другому процессу это делать будет запрещено.

Выбор подходящей примитивной операции, реализующей взаимное исключение, является серьезным моментом разработки операционной системы.

Критические области. Состязания между процессами.

Формулировка состояния состязания:

1. Некоторый промежуток времени процесс занят внутренними расчетами и другими задачами, не приводящими к состояниям состязания.
2. В другие моменты времени процесс обращается к совместно используемым данным или выполняет действие, которое может привести к состязанию.
3. Часть программы, в которой есть обращение к совместно используемым данным, называется **критической областью** или **критической секцией**.
4. Если удастся избежать одновременного нахождения двух процессов в критических областях, можно избежать состязаний.

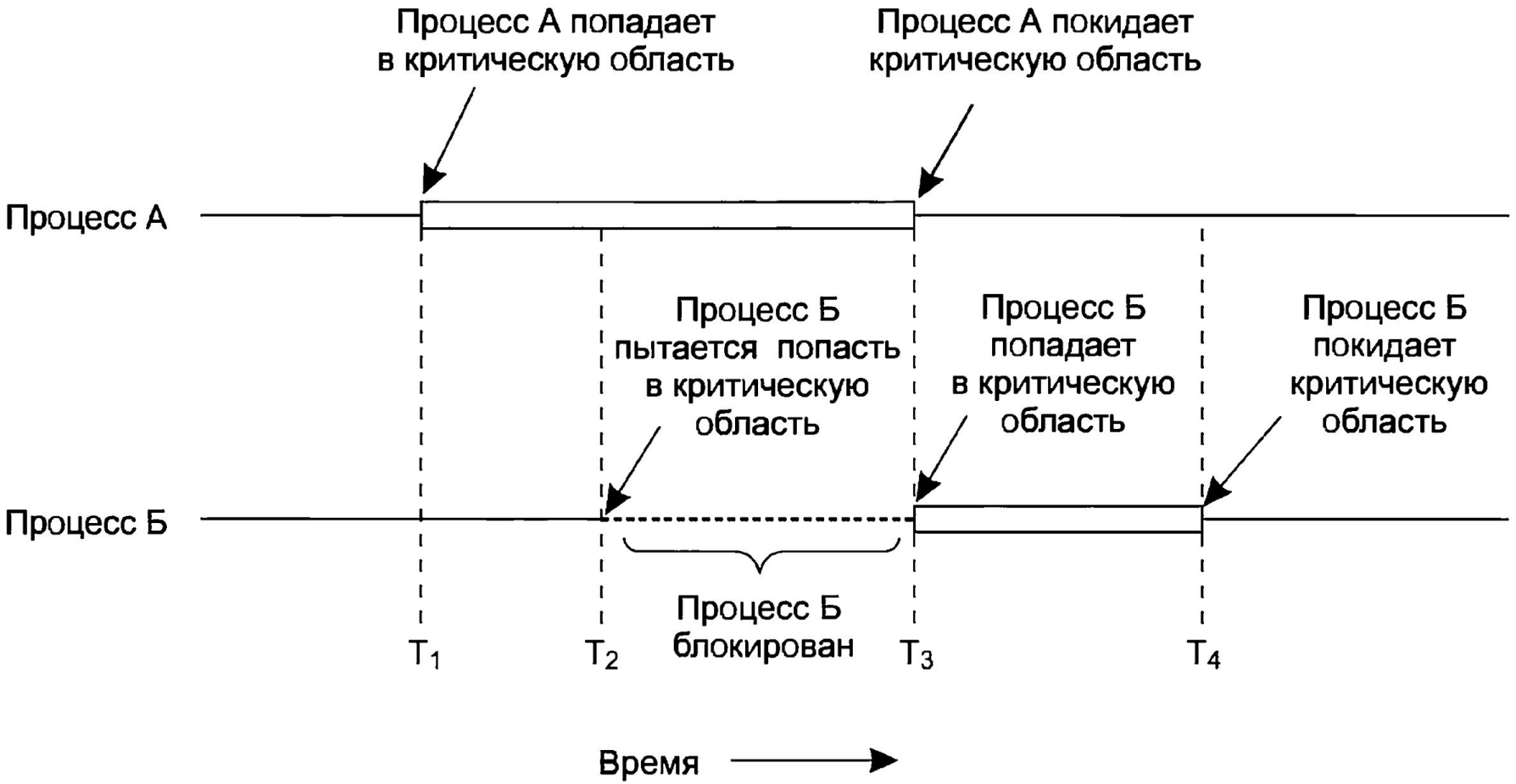
Критические области

Для правильной совместной работы параллельных процессов и эффективного использования общих данных необходимо выполнение четырех условий:

1. Два процесса не должны одновременно находиться в критических областях.
2. В программе не должно быть предположений о скорости или количестве процессоров.
3. Процесс, находящийся вне критической области, не может блокировать другие процессы.
4. Невозможна ситуация, в которой процесс вечно ждет попадания в критическую область.

Критические области

- Взаимное исключение использования критических областей



Взаимное исключение с активным ожиданием

- Запрещение прерываний
- Переменные блокировки
- Строгое чередование
- Алгоритм Петерсона
- Команда TSL
- Семафоры
- Мьютексы
- Мониторы
- Передача сообщений
- Барьеры

Взаимное исключение с активным ожиданием

способы реализации

Запрещение прерываний

- Запрет всех прерываний при входе процесса в критическую область и разрешение прерываний по выходе из области.
- Если прерывания запрещены, невозможно прерывание по таймеру. Поскольку процессор переключается с одного процесса на другой только по прерыванию, отключение прерываний исключает передачу процессора другому процессу.

Запрещение прерываний (пример)

Пример: процесс пользователя отключил все прерывания и в результате какого-либо сбоя не включил их обратно. Операционная система на этом может закончить свое существование.

Пример: для ядра характерно запрещение прерываний для некоторых команд при работе с переменными или списками.

Итак, запрет прерываний бывает полезным в самой операционной системе.

Переменные блокировки

Рассмотрим одну совместно используемую переменную блокировки, изначально равную 0.

- Если процесс хочет попасть в критическую область, он предварительно считывает значение переменной блокировки.
- Если переменная равна 0, процесс изменяет ее на 1 и входит в критическую область.
- Если же переменная равна 1, то процесс ждет, пока ее значение сменится на 0.

Переменные блокировки (недостатки)

1. Один процесс считывает переменную блокировки, обнаруживает, что она равна 0, но прежде, чем он успевает изменить ее на 1, управление получает другой процесс, успешно изменяющий ее на 1.
2. Когда первый процесс снова получит управление, он тоже заменит переменную блокировки на 1 и два процесса одновременно окажутся в критических областях.

Строгое чередование

Третий метод реализации взаимного исключения.

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

а

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

б

Строгое чередование

Переменная turn=0 отслеживает, чья очередь входить в критическую область.

1. Вначале процесс 0 проверяет значение turn, считывает 0 и входит в критическую область.
2. Процесс 1 также проверяет значение turn, считывает 0 и после этого входит в цикл, непрерывно проверяя, когда же значение turn будет равно 1.

Постоянная проверка значения переменной в ожидании некоторого значения называется **активным ожиданием**. Активное ожидание используется только в случае, когда есть уверенность в небольшом времени ожидания.

Блокировка, использующая активное ожидание, называется **спин-блокировкой**.

Алгоритм Петерсона

```
#define FALSE 0
#define TRUE  1
#define N     2                       /* Количество процессов */
int turn;                             /* Чья сейчас очередь? */
int interested[N];                    /* Все переменные изначально равны 0 (FALSE) */
void enter_region(int process);       /* Процесс 0 или 1 */
{
       int other;                     /* Номер второго процесса */
       other = 1 - process;           /* Противоположный процесс */
       interested[process] = TRUE;    /* Индикатор интереса*/
       turn = process;                /* Установка флага*/
       while (turn == process && interested[other] == TRUE) /* Пустой оператор */;
}
void leave_region(int process)        /* process: процесс, покидающий критическую область */
{
       interested[process] = FALSE; /* Индикатор выхода из критической области */
}
```

Команда TSL

Рассмотрим решение, требующее участия аппаратного обеспечения. Многие компьютеры, особенно разработанные с расчетом на несколько процессоров, имеют команду

TSL RX.LOCK (Test and Set Lock - проверить и заблокировать), которая действует следующим образом.

В регистр RX считывается содержимое слова памяти lock, а в ячейке памяти lock сохраняется некоторое ненулевое значение. Гарантируется, что операция считывания слова и сохранения неделима - другой процесс не может обратиться к слову в памяти, пока команда не выполнена.

Команда TSL

```
enter_region:
        TSL REGISTER,LOCK       | значение lock копируется в регистр, значение переменной
                                  устанавливается равным 1
        CMP REGISTER,#0         | Старое значение lock сравнивается с нулем
        JNE enter_region        | Если оно ненулевое, значит, блокировка уже была установлена,
                                  поэтому цикл завершается
        RET                     | Возврат к вызывающей программе, процесс вошел в критическую
                                  область

leave_region:
        MOVE LOCK,#0            | Сохранение 0 в переменной lock
        RET
```

Примитивы межпроцессного взаимодействия

Оба решения - Петерсона и с использованием команды TSL - корректны, но они обладают одним и тем же недостатком: использованием **активного ожидания**.

В сущности, оба они реализуют следующий алгоритм: перед входом в критическую область процесс проверяет, можно ли это сделать. Если нельзя, процесс входит в тугой цикл, ожидая возможности войти в критическую область.

Примитивы межпроцессного взаимодействия. Пример.

Этот алгоритм не только бесцельно расходует время процессора, но, кроме этого, он может иметь некоторые неожиданные последствия.

Рассмотрим два процесса: H, с высоким приоритетом, и L, с низким приоритетом. Правила планирования в этом случае таковы, что процесс H запускается немедленно, как только он оказывается в состоянии ожидания. В какой-то момент, когда процесс L находится в критической области, процесс H оказывается в состоянии ожидания. Процесс H попадает в состояние активного ожидания, но поскольку процессу L во время работающего процесса H никогда не будет предоставлено процессорное время, у процесса L не будет возможности выйти из критической области, и процесс H навсегда останется в цикле. Эту ситуацию иногда называют **проблемой инверсии приоритета**.

Примитивы межпроцессного взаимодействия

Теперь рассмотрим некоторые примитивы межпроцессного взаимодействия, применяющиеся вместо циклов ожидания, в которых лишь напрасно расходуется процессорное время. Эти примитивы блокируют процессы в случае запрета на вход в критическую область. Одной из простейших является пара примитивов sleep и wakeup.

Примитив **sleep** - системный запрос, в результате которого вызывающий процесс блокируется, пока его не запустит другой процесс.

Примитив **wakeup** есть один параметр - процесс, который следует запустить. Также возможно наличие одного параметра у обоих запросов - адреса ячейки памяти, используемой для согласования запросов ожидания и запуска.

Семафоры

В 1965 году Дейкстра (E. W. Dijkstra) предложил использовать целую переменную для подсчета сигналов запуска, сохраненных на будущее.

Им был предложен новый тип переменных, так называемые **семафоры**, значение которых может быть нулем (в случае отсутствия сохраненных сигналов активизации) или некоторым положительным числом, соответствующим количеству отложенных активизирующих сигналов.

Семафоры.
Операции: **down** и **up.**

Операция down сравнивает значение семафора с нулем.

Если значение семафора больше нуля, операция down уменьшает его и просто возвращает управление. Если значение семафора равно нулю, процедура down не возвращает управление процессу, а процесс переводится в состояние ожидания.

Все операции проверки значения семафора, его изменения и перевода процесса в состояние ожидания выполняются как единое и неделимое элементарное действие. Тем самым гарантируется, что после начала операции ни один процесс не получит доступа к семафору до окончания или блокирования операции.

Мьютексы

Иногда используется упрощенная версия семафора, называемая **мьютексом** (mutex, сокращение от mutual exclusion - взаимное исключение).

Мьютекс не способен считать, он может лишь управлять взаимным исключением доступа к совместно используемым ресурсам или кодам.

Реализация мьютекса проста и эффективна, что делает использование мьютексов особенно полезным в случае потоков, действующих только в пространстве пользователя.

Мьютексы

Мьютекс - переменная, которая может находиться в одном из двух состояний: блокированном или неблокированном.

Для описания мьютекса требуется всего один бит, хотя чаще используется целая переменная, у которой 0 означает неблокированное состояние, а все остальные значения соответствуют блокированному состоянию.

Значение мьютекса устанавливается двумя процедурами. Если поток собирается войти в критическую область, он вызывает процедуру **mutex_lock**. Если мьютекс не заблокирован, запрос выполняется и вызывающий поток может попасть в критическую область.

Мониторы

- В 1974 году Хоар (Hoare) и Бринч Хансен (Brinch Hansen) предложили примитив синхронизации более высокого уровня, называемый **монитором**.
- **Монитор** - набор процедур, переменных и других структур данных, объединенных в особый модуль или пакет. Процессы могут вызывать процедуры монитора, но у процедур, объявленных вне монитора, нет прямого доступа к внутренним структурам данных монитора.

Передача сообщений

Этот метод межпроцессного взаимодействия использует два примитива: send и receive, которые скорее являются системными вызовами, чем структурными компонентами языка.

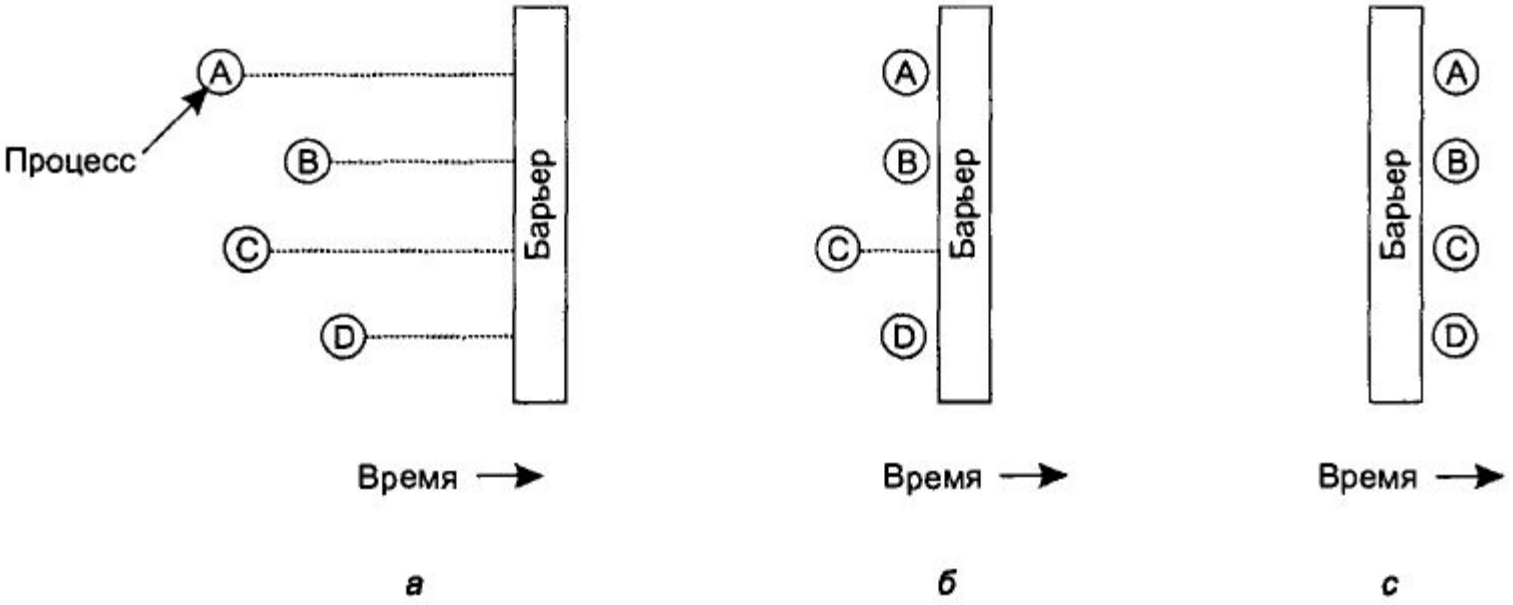
Например:

```
send(destination, Smessage);
receive(source, &message);
```

Первый запрос посылает сообщение заданному адресату, а второй получает сообщение от указанного источника. Если сообщения нет, второй запрос блокируется до поступления сообщения либо немедленно возвращает код ошибки.

Барьеры

Последний из рассмотренных нами механизмов синхронизации предназначался скорее для групп процессов, нежели для ситуаций с двумя процессами. Некоторые приложения делятся на фазы, и существует правило, что процесс не может перейти в следующую фазу, пока к этому не готовы все остальные процессы. Этого можно добиться, разместив в конце каждой фазы барьер.

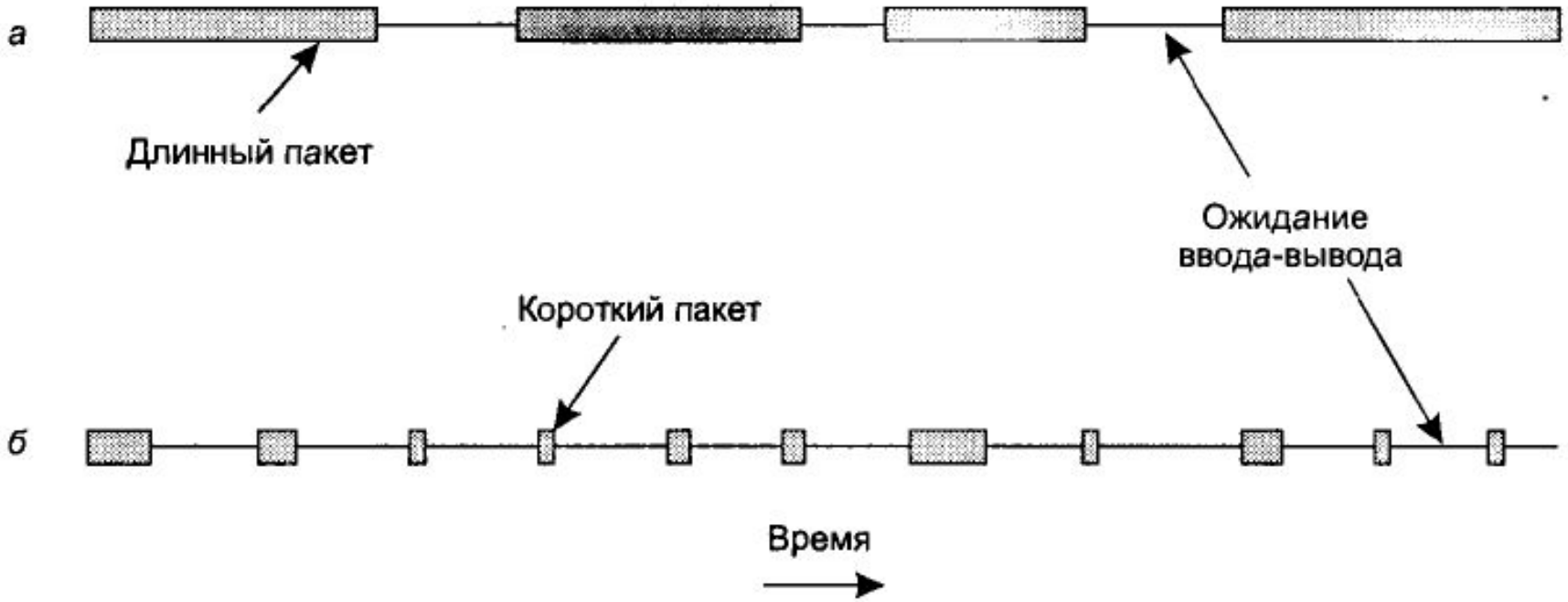


а *б* *с*

Планирование

Поведение процесса

Практически все процессы чередуют периоды вычислений с операциями ввода-вывода. Обычно процессор некоторое время работает без остановки, затем происходит системный вызов на чтение из файла или запись в файл. После выполнения системного вызова процессор опять считает, пока ему не понадобятся новые данные или не потребуется записать полученные данные и т. д.



Когда планировать ?

Ключевым вопросом планирования является выбор момента принятия решений. Оказывается, существует множество ситуаций, в которых необходимо планирование.

Во-первых, когда создается новый процесс, необходимо решить, какой процесс запустить: родительский или дочерний.

Во-вторых, планирование необходимо, когда процесс завершает работу.

В-третьих, когда процесс блокируется на операции ввода-вывода, семафоре, или по какой-либо другой причине, необходимо выбрать и запустить другой процесс.

В-четвертых, необходимость планирования может возникнуть при появлении прерывания ввода-вывода. Если прерывание пришло от устройства ввода-вывода, закончившего работу, можно запустить процесс, который был блокирован в ожидании этого события.

Когда планировать ? (2)

Алгоритмы планирования можно разделить на две категории согласно их поведению после прерываний.

Алгоритмы планирования без переключений, иногда называемого также неприоритетным планированием.

Алгоритмы планирования с переключениями, называемого также приоритетным планированием, выбирают процесс и позволяют ему работать некоторое максимально возможное фиксированное время.

Категории алгоритмов планирования

В различных средах требуются различные алгоритмы планирования. Это связано с тем, что различные операционные системы и различные приложения ориентированы на разные задачи. Другими словами, то, для чего следует оптимизировать планировщик, различно в разных системах.

Можно выделить три среды:

1. Системы пакетной обработки данных.
2. Интерактивные системы.
3. Системы реального времени.

Категории алгоритмов планирования

В системах пакетной обработки нет пользователей, сидящих за терминалами и ожидающих ответа. В таких системах приемлемы алгоритмы без переключений или с переключениями, но с большим временем, отводимым каждому процессу. Такой метод уменьшает количество переключений между процессами и улучшает эффективность.

Категории алгоритмов планирования

В интерактивных системах необходимы алгоритмы планирования с переключениями, чтобы предотвратить захват процессора одним процессом. Даже если ни один процесс не захватывает процессор на неопределенно долгий срок намеренно, из-за ошибки в программе один процесс может заблокировать остальные. Для исключения подобных ситуаций используется планирование с переключениями.

Категории алгоритмов планирования

В системах с ограничениями реального времени приоритетность, как это ни странно, не всегда обязательна, поскольку процессы знают, что их время ограничено, и быстро выполняют работу, а затем блокируются. Отличие от интерактивных систем в том, что в системах реального времени работают только программы, предназначенные для содействия конкретным приложениям. Интерактивные системы являются универсальными системами.

Задачи алгоритмов планирования

Все системы

- **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.

Пакетные системы

- **Производительность** — выполнение максимального количества заданий в час.
- **Оборотное время** — минимизация времени между представлением задачи и ее завершением.
- **Использование центрального процессора** — поддержка постоянной загруженности процессора.

Интерактивные системы

- **Время отклика** — быстрый ответ на запросы.
- **Пропорциональность** — оправдание пользовательских надежд.

Системы реального времени

- **Соблюдение предельных сроков** — предотвращение потери данных.
- **Предсказуемость** — предотвращение ухудшения качества в мультимедийных системах.

Планирование в системах пакетной обработки заданий

«Первым пришел – первым ушел»

«Первым пришел – первым ушел»

Алгоритм без переключений «первым пришел - первым обслужен» является, пожалуй, самым простым из алгоритмов планирования. Процессам предоставляется доступ к процессору в том порядке, в котором они его запрашивают. Чаще всего формируется единая очередь ждущих процессов. Как только появляется первая задача, она немедленно запускается и работает столько, сколько необходимо.

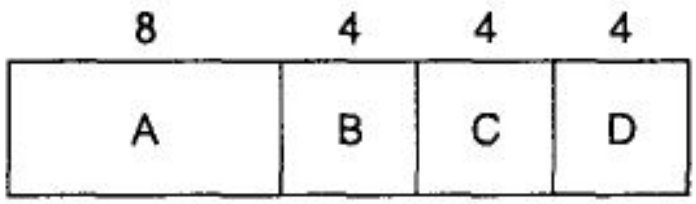
Остальные задачи ставятся в конец очереди. Когда текущий процесс блокируется, запускается следующий в очереди, а когда блокировка снимается, процесс попадает в конец очереди. Основным преимуществом этого алгоритма является то, что его легко понять и столь же легко программировать. Он справедлив в том же самом смысле, в каком справедливо распределение дефицитных билетов на концерт или соревнования среди всех желающих стоять в очереди с двух часов ночи. В этом алгоритме все процессы в состоянии готовности контролируются одним связным списком. Чтобы выбрать процесс для запуска, нужно всего лишь взять первый элемент списка и удалить его. Появление нового процесса приводит к помещению его в конец списка - что может быть проще?

«Кратчайшая задача - первая»

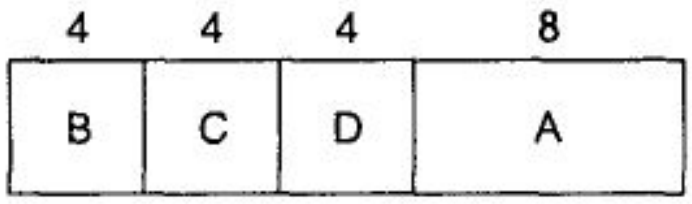
Рассмотрим еще один алгоритм без переключений для систем пакетной обработки, предполагающий, что временные отрезки работы известны заранее.

Например, работники страховой компании могут довольно точно предсказать, сколько времени займет обработка Пакета из 1000 исков, поскольку они делают это каждый день. Если в очереди есть несколько одинаково важных задач, планировщик выбирает первой самую короткую задачу.

Есть четыре задачи: A, B, C и D, со временем выполнения 8, 4, 4 и 4 мин соответственно.



а



б

Наименьшее оставшееся время выполнения

Версией предыдущего алгоритма с переключениями является алгоритм наименьшего оставшегося времени выполнения.

В соответствии с этим алгоритмом планировщик каждый раз выбирает процесс с наименьшим оставшимся временем выполнения.

В этом случае также необходимо заранее знать время выполнения задач.

Когда поступает новая задача, ее полное время выполнения сравнивается с оставшимся временем выполнения текущей задачи.

Если время выполнения новой задачи меньше, текущий процесс приостанавливается и управление передается новой задаче.

Эта схема позволяет быстро обслуживать короткие запросы.

Трехуровневое планирование

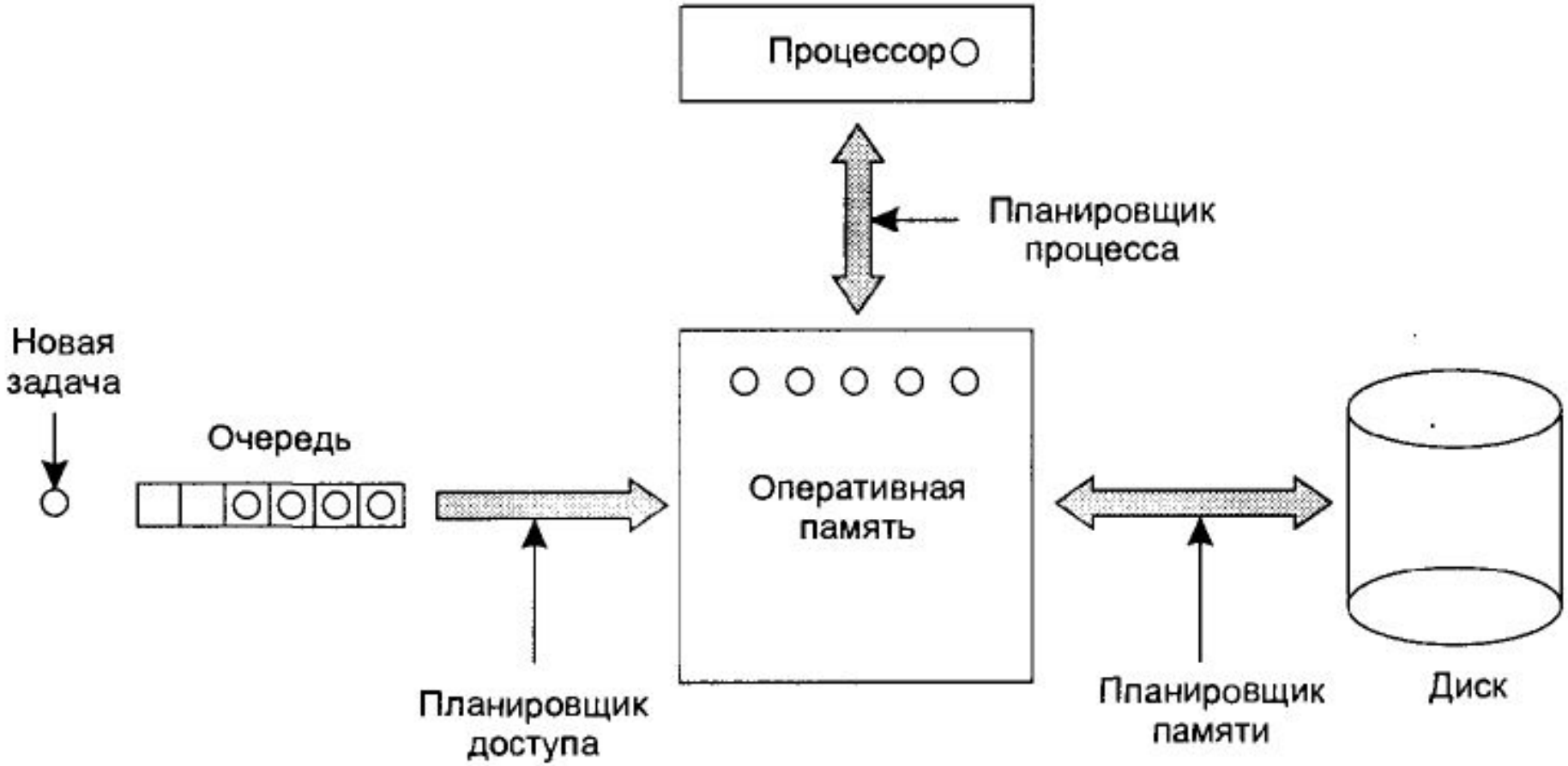
Системы пакетной обработки позволяют реализовать трехуровневое планирование. По мере поступления в систему новые задачи сначала помещаются в очередь, хранящуюся на диске.

Впускной планировщик выбирает задание и передает его системе. Остальные задания остаются в очереди. Характерный алгоритм входного контроля может заключаться в выборе смеси из процессов, ограниченных возможностями процессора, и процессов, ограниченных возможностями устройств ввода-вывода.

Также возможен алгоритм, в котором устанавливается приоритет коротких задач перед длинными.

Впускной планировщик волен придержать некоторые задания во входной очереди, а пропустить задание, поступившее позже остальных.

Трехуровневое планирование (2)



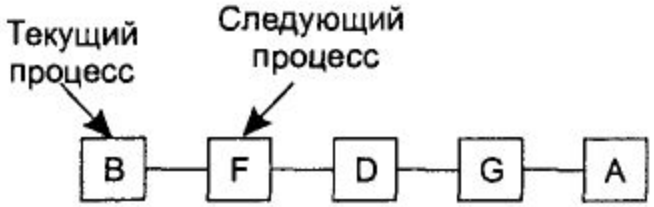
Планирование в интерактивных системах

- Циклическое планирование
- Приоритетное планирование
- Несколько очередей
- «Самый короткий процесс – следующий»
- Гарантированное планирование
- Лотерейное планирование
- Справедливое планирование

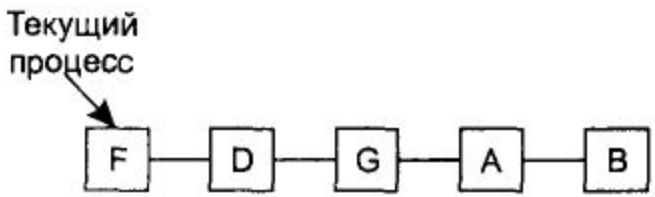
Циклическое планирование

Одним из наиболее старых, простых, справедливых и часто используемых является алгоритм циклического планирования. Каждому процессу предоставляется некоторый интервал времени процессора, так называемый квант времени.

Если к концу кванта времени процесс все еще работает, он прерывается, а управление передается другому процессу. Разумеется, если процесс блокируется или прекращает работу раньше, переход управления происходит в этот момент. Реализация циклического планирования проста. Планировщику нужно всего лишь поддерживать список процессов в состоянии готовности. Когда процесс исчерпал свой лимит времени, он отправляется в конец списка.



а



б

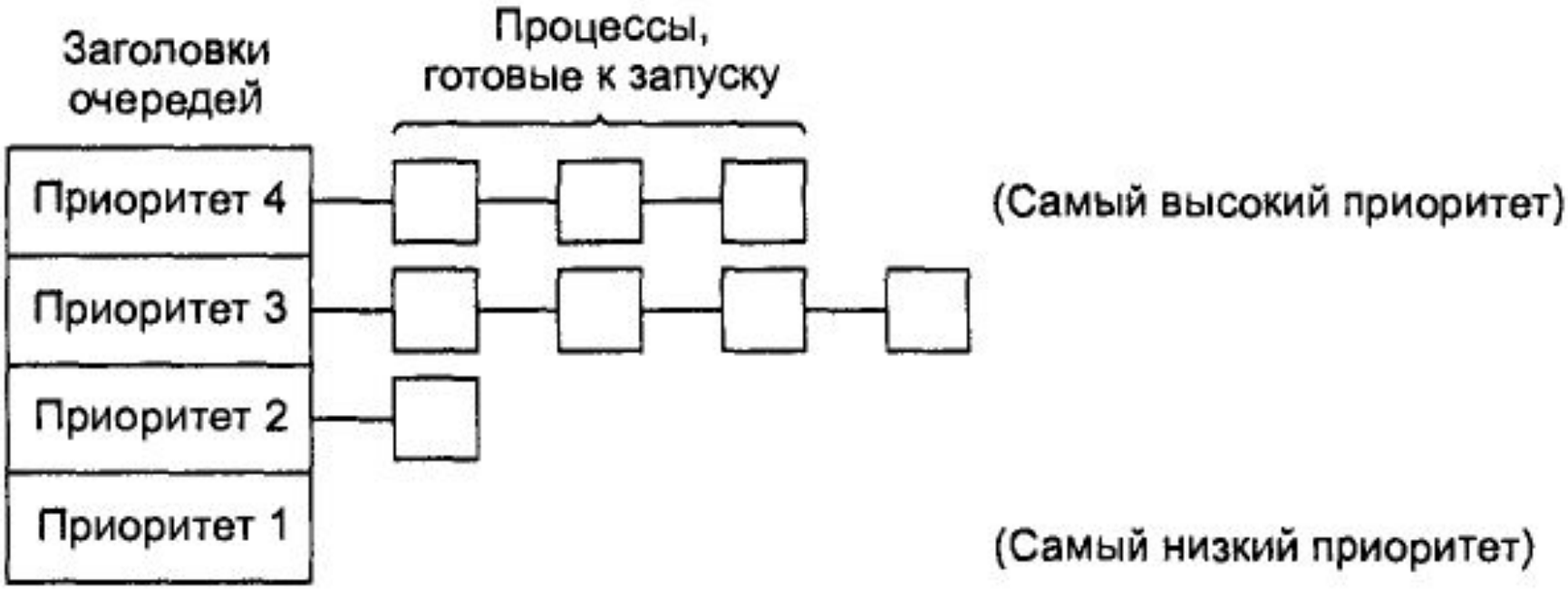
Приоритетное планирование

Необходимость принимать во внимание подобные внешние факторы приводит к приоритетному планированию. Основная идея проста: каждому процессу присваивается приоритет, и управление передается готовому к работе процессу с самым высоким приоритетом.

Даже на персональном компьютере с одним пользователем может происходить несколько процессов, отдельные из которых являются более важными, чем другие. Демон, отвечающий за пересылку электронной почты в фоновом режиме, имеет более низкий приоритет, чем процесс, отображающий на экране видеофильм в реальном времени.

Приоритеты процессам могут присваиваться **статически** или **динамически**.

Приоритетное планирование (2)



Несколько очередей

Один из первых приоритетных планировщиков был реализован в системе CTSS (compatible time-shared system - совместимая система с разделением времени).

В результате было разработано решение с классами приоритетов. Процессам класса с высшим приоритетом выделялся один квант, процессам следующего класса - два кванта, следующего - четыре кванта и т. д. Когда процесс использовал все отведенное ему время, он перемещался на класс ниже. В качестве примера рассмотрим процесс, которому необходимо производить вычисления в течение 100 квантов. Вначале ему будет предоставлен один квант, затем он будет перекачан на диск. В следующий раз ему достанется 2 кванта, затем 4, 8,16, 32, 64, хотя из 64 он использует только 37.

«Самый короткий процесс – следующий»

Поскольку алгоритм «Кратчайшая задача – первая» минимизирует среднее оборотное время в системах пакетной обработки, хотелось бы использовать его и в интерактивных системах. Интерактивные процессы чаще всего следуют схеме «ожидание команды, исполнение команды, ожидание команды, исполнение команды...» Если рассматривать выполнение каждой команды как отдельную задачу, можно минимизировать общее среднее время отклика, запуская первой самую короткую задачу.

Один из методов основывается на оценке длины процесса, базирующейся на предыдущем поведении процесса. При этом запускается процесс, у которого оцененное время самое маленькое. Допустим, что предполагаемое время исполнения команды равно *T0* и предполагаемое время следующего запуска равно *T1*. Можно улучшить оценку времени, взяв взвешенную сумму этих времен $aT0 + (1 - a)T1$.

Метод оценки следующего значения серии через взвешенное среднее предыдущего значения и предыдущей оценки часто называют **старением**.

Гарантированное планирование

Принципиально другим подходом к планированию является предоставление пользователям реальных обещаний и затем их выполнение. Вот одно обещание, которое легко произнести и легко выполнить: если вместе с вами процессором пользуются n пользователей, вам будет предоставлено *1/n* мощности процессора.

В системе с одним пользователем и *n* запущенными процессорами каждому достанется *1/n* циклов процессора. Чтобы выполнить это обещание, система должна отслеживать распределение процессора между процессами с момента создания каждого процесса. Затем система рассчитывает количество ресурсов процессора, на которое процесс имеет право, например время с момента создания, деленное на *n*. Теперь можно сосчитать отношение времени, предоставленного процессу, к времени, на которое он имеет право. Полученное значение 0.5 означает, что процессу выделили только половину положенного, а 2.0 означает, что процессу досталось в два раза больше, чем положено. Затем запускается процесс, у которого это отношение наименьшее, пока оно не станет больше, чем у его ближайшего соседа.

Лотерейное планирование

Хотя идея обещаний пользователям и их выполнения хороша, но ее трудно реализовать. Для более простой реализации предсказуемых результатов используется другой алгоритм, называемый **лотерейным планированием**.

В основе алгоритма лежит раздача процессам лотерейных билетов на доступ к различным ресурсам, в том числе и к процессору. Когда планировщику необходимо принять решение, выбирается случайным образом лотерейный билет, и его обладатель получает доступ к ресурсу. Что касается доступа к процессору, «лотерея» может происходить 50 раз в секунду, и победитель получает 20 мс времени процессора.

Справедливое планирование

До сих пор мы предполагали, что каждый процесс управляется независимо от того, кто его хозяин. Поэтому если пользователь 1 создаст 9 процессов, а пользователь 2 - 1 процесс, то с использованием циклического планирования или в случае равных приоритетов пользователю 1 достанется 90 % процессора, а пользователю 2 всего 10.

Чтобы избежать подобных ситуаций, некоторые системы обращают внимание на хозяина процесса перед планированием. В такой модели каждому пользователю достается некоторая доля процессора, и планировщик выбирает процесс в соответствии с этим фактом. Если в нашем примере каждому из пользователей было обещано по 50 % процессора, то им достанется по 50 % процессора, независимо от количества процессов.

В качестве примера рассмотрим систему и двух пользователей, каждому из которых отведено по 50 % процессора. У первого пользователя четыре процесса: A, B, C и D, у второго один процесс E. Если используется циклическое планирование, цепочка процессов, удовлетворяющая всем требованиям, будет выглядеть следующим образом:

A E B E C E D E A E B E C E D E ...

Планирование в системах реального времени

Системы реального времени делятся на **жесткие системы реального времени**, что означает наличие жестких сроков для каждой задачи (в них обязательно надо укладываться), и **гибкие системы реального времени**, в которых нарушения временного графика нежелательны, но допустимы. В обоих случаях реализуется разделение программы на несколько процессов, каждый из которых предсказуем. Эти процессы чаще всего бывают короткими и завершают свою работу в течение секунды. Когда появляется внешний сигнал, именно планировщик должен обеспечить соблюдение графика.

Планирование в системах реального времени (2)

Внешние события, на которые система должна реагировать, можно разделить на:

- периодические (возникающие через регулярные интервалы времени);
- непериодические (возникающие непредсказуемо).

Возможно наличие нескольких периодических потоков событий, которые система должна обрабатывать. В зависимости от времени, затрачиваемого на обработку каждого из событий, может оказаться, что система не в состоянии своевременно обработать все события.

Если в систему поступает *m* периодических событий, событие с номером *i* поступает с периодом *Pi* и на его обработку уходит *Ci*, секунд работы процессора, все потоки могут быть своевременно обработаны только при выполнении условия

$$\sum_{i=1}^{m} \frac{C_i}{P_i} \le 1.$$

Системы реального времени, удовлетворяющие этому условию, называются **планируемыми**.

Проблема обедающих философов