НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ЯДЕРНЫЙ УНИВЕРСИТЕТ «МИФИ»
Кафедра информатики и процессов управления (№17)

Курс «Современные операционные системы»

Лекция 5

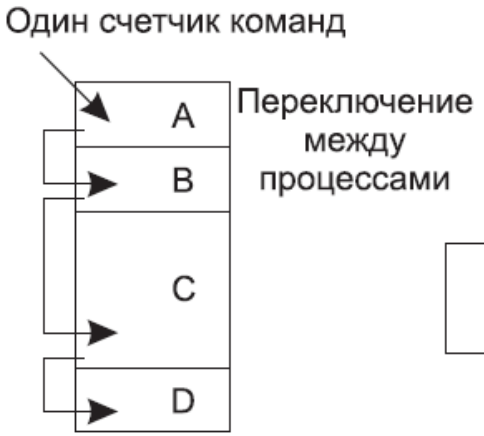
Процессы и потоки

Москва
2016

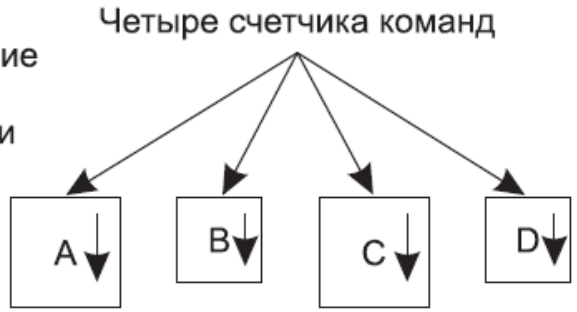
Содержание

1. Процессы

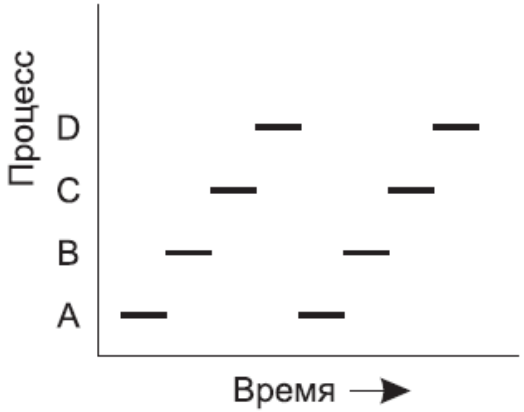
1.1. Модель процесса



Четыре программы, работающие в многозадачном режиме.



Модель четырех независимых друг от друга последовательных процессов.



В отдельно взятый момент активна только одна программа.

Разница между процессом и программой: процесс — это своего рода действия, у него есть программа, входные и выходные данные и состояние. Один процессор может совместно использоваться несколькими процессами в соответствии с неким алгоритмом планирования, который используется для определения того, когда остановить один процесс и обслужить другой. В отличие от процесса программа может быть сохранена на диске и вообще ничего не делать. Например, если программа запущена дважды, то считается, что ею заняты два процесса.

1.2. Создание процесса

Основные события, приводящие к созданию процессов.

1. Инициализация системы.
2. Выполнение работающим процессом системного вызова, предназначенного для создания процесса.
3. Запрос пользователя на создание нового процесса.
4. Инициация пакетного задания (только для ОС пакетной обработки данных).

При запуске ОС создаются, как правило, несколько процессов. Фоновые процессы, предназначенные для обработки какой-либо активной деятельности, называются ***демонами***. Обычно у больших ОС насчитываются десятки демонов.

В UNIX-подобных ОС для отображения списка запущенных процессов может быть использована программа **ps**. В Windows для этого используется ***диспетчер задач***.

В UNIX существует только один системный вызов для создания нового процесса — `fork`. Этот вызов создает точную копию вызывающего процесса. Теперь два процесса, родительский и дочерний, имеют одинаковый образ памяти, одинаковые строки описания конфигурации и одни и те же открытые файлы. Обычно после этого дочерний процесс изменяет свой образ памяти и запускает новую программу, выполняя системный вызов `execve` или ему подобный.

В Windows все происходит иначе: одним вызовом функции Win32 `CreateProcess` создается процесс, и в него загружается нужная программа. У этого вызова имеется 10 параметров, включая:

- выполняемую программу,
- параметры командной строки для этой программы,
- различные параметры безопасности,
- биты, управляющие наследованием открытых файлов,
- информацию о приоритетах,
- спецификацию окна, создаваемого для процесса (если оно используется),
- указатель на структуру, в которой вызывающей программе будет возвращена информация о только что созданном процессе.

Кроме того в Win32 имеется около 100 других функций для управления процессами и их синхронизации, а также выполнения всего, что с этим связано.

В UNIX и Windows, после создания процесса родительский и дочерний процессы обладают своими собственными, отдельными адресными пространствами. Если какой-нибудь процесс изменяет слово в своем адресном пространстве, другим процессам эти изменения не видны. Тем не менее вновь созданный процесс может делить со своим создателем часть других ресурсов, например открытые файлы.

1.3. Завершение процесса

Процессы завершаются, в силу следующих обстоятельств:

- обычного выхода (добровольно);
- выхода при возникновении ошибки (добровольно);
- возникновения фатальной ошибки (принудительно);
- уничтожения другим процессом (принудительно).

Системный вызов процесса, сообщающий ОС о завершении своей работы: в UNIX — `exit`, в Windows — `ExitProcess`. Системный вызов процесса, приказывающий ОС завершить некоторые другие процессы: в UNIX — `kill`, в Windows — `TerminateProcess`.

1.4. Состояния процессов

Три состояния, в которых может находиться процесс:

- ***выполняемый*** (в данный момент использующий центральный процессор);
- ***готовый*** (работоспособный, но временно приостановленный, чтобы дать возможность выполнения другому процессу);
- ***заблокированный*** (неспособный выполняться, пока не возникнет какое-нибудь внешнее событие).

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

Состояния процесса. Стрелками показаны переходы между состояниями

Обработку прерываний и действия, запускающие и останавливающие процессы, выполняет ***планировщик***. Остальная часть ОС может быть структурирована в виде процессов.

1.5. Выполнение процессов

Для реализации модели процессов ОС ведет ***таблицу процессов*** (состоящую из массива структур) в которой каждая запись соответствует конкретному процессу. Эти записи содержат важную информацию о состоянии процесса, включая:

- счетчик команд,
- указатель стека,
- распределение памяти,
- состояние открытых файлов,
- учетную и планировочную информацию

и все остальное, что должно быть сохранено, когда процесс переключается из состояния *выполнения* в состояние *готовности* или *блокировки*, чтобы позже он мог возобновить выполнение, как будто никогда не останавливался.

Некоторые из полей типичной записи таблицы процессов.

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

Схема обработки ***прерывания*** и планирования (низший уровень ОС). При возникновении ***прерывания***:

1) Оборудование помещает в стек счетчик команд и др.
2) Оборудование загружает новый счетчик команд из вектора прерывания
3) Процедура на ассемблере сохраняет регистры
4) Процедура на ассемблере устанавливает указатель на новый стек
5) Запускается процедура на языке С, обслуживающая прерывание (как правило, она считывает входные данные и помещает их в буфер)
6) Планировщик принимает решение, какой процесс запускать следующим
7) Процедура на языке С возвращает управление ассемблерному коду
8) Процедура на ассемблере запускает новый текущий процесс

(***Вектор прерывания*** – область памяти (обычно это фиксированная область в нижних адресах), связанная с каждым классом устройств ввода-вывода, в которой содержится адрес процедуры, обслуживающей прерывание.)

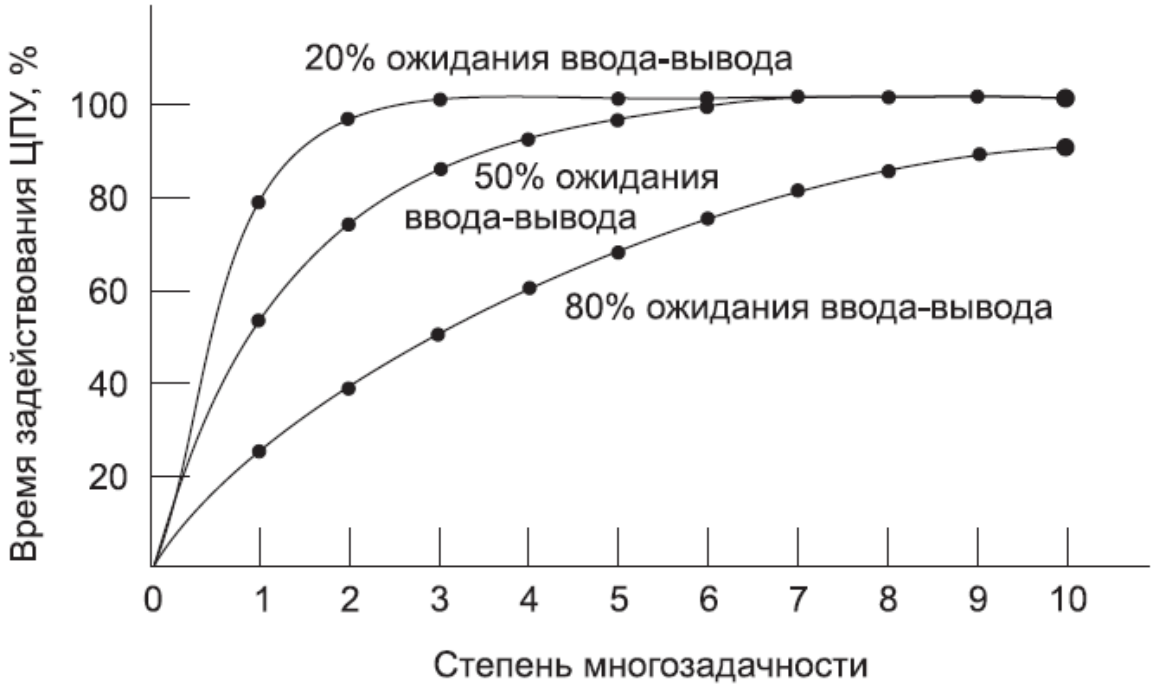
После каждого прерывания прерванный процесс возвращается в точности к такому же состоянию, в котором он был до того, как случилось прерывание.

1.6. Режим многозадачности

Пусть процесс проводит часть своего времени p в ожидании завершения операций ввода-вывода. При одновременном присутствии в памяти n процессов вероятность того, что все они ожидают завершения ввода-вывода (в случае чего процессор простаивает), равна p^n. Тогда время задействования процессора:

$$t = 1 - p^n.$$

Пример функции зависимости при $p = 0{,}2$; $0{,}3$ и $0{,}8$:



Степень многозадачности (n) – количество процессов, присутствующих в памяти.

Вероятностная модель построена на предположении, что все n процессов являются независимыми друг от друга. На самом деле, поскольку на одном ЦП процессы ожидают своего выполнения в очереди, они не обладают независимостью. Но график функции при этом имеет примерно такой же характер.

2. Потоки

У каждого процесса есть адресное пространство и ***поток (thread) управления***. Нередко возникают ситуации, когда хорошо иметь в одном и том же адресном пространстве несколько потоков управления, выполняемых так, как будто они являются обособленными процессами, но на общем адресном пространстве.

Потоки полезны, когда в одном приложении одновременно происходит несколько действий, часть которых может периодически быть заблокированной. Наличие потоков позволяет действиям приложения перекрываться по времени, ускоряя его работу.

Преимуществом является ***«легковесность»*** потоков (т.е. быстрота их создания и ликвидации) по сравнению с «тяжеловесными» процессами. Во многих системах создание потоков осуществляется в 10–100 раз быстрее, чем создание процессов.

Потоки очень эффективны для систем, имеющих несколько процессоров, где есть реальная возможность параллельных вычислений.

Пример. Редактирование документа в WYSIWYG-редакторе (What You See Is What You Get – «что видишь, то и получишь»), использующем три потока. Первый поток занят только взаимодействием с пользователем. Второй поток по необходимости занимается переформатированием документа. А третий поток периодически сохраняет документ на диск. При этом использовать три отдельных процесса невозможно, поскольку всем им необходимо работать с одним и тем же документом.

Пример. Структура процесса может включать входной поток, обрабатывающий поток и выходной поток. Входной поток считывает данные с диска во входной буфер. Обрабатывающий поток извлекает данные из входного буфера, обрабатывает их и помещает результат в выходной буфер. Выходной буфер записывает эти результаты обратно на диск. Таким образом, ввод, вывод и обработка данных могут осуществляться одновременно. Такая модель будет работать лишь при том условии, что системный вызов блокирует только вызывающий его поток, а не весь процесс.

2.1. Классическая модель потоков

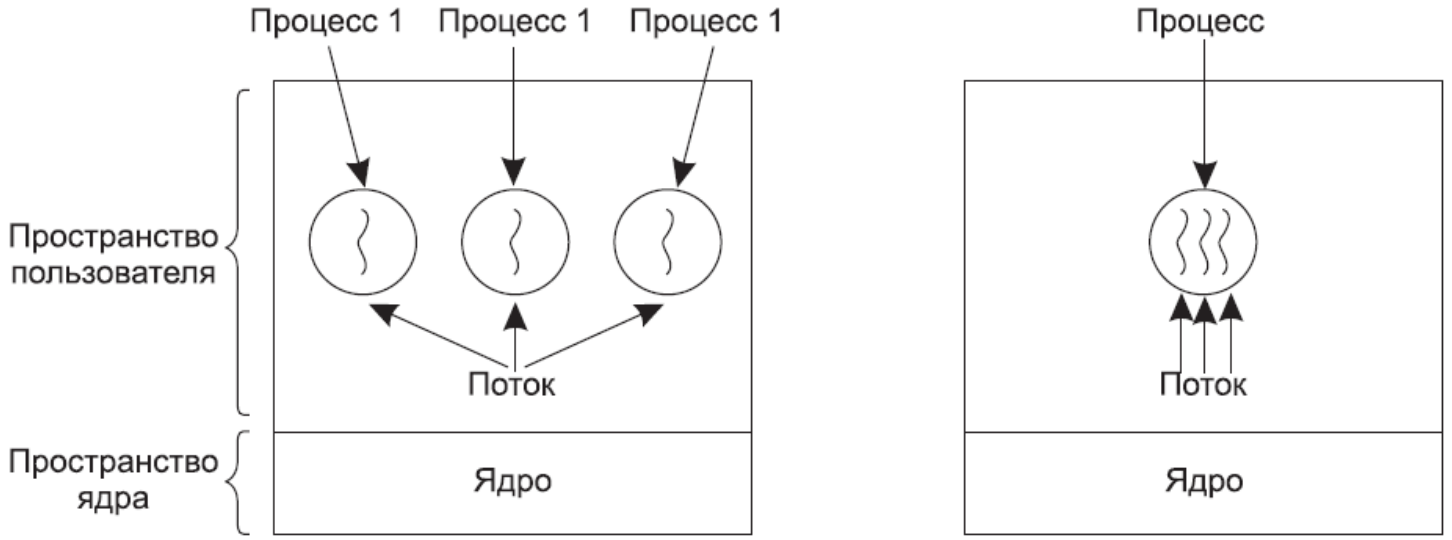
Процессы используются для **группировки взаимосвязанных ресурсов** в единое целое, а **потоки** являются «сущностью», распределяемой для **выполнения на центральном процессоре**.

Процесс содержит:

- адресное пространство (содержащее текст программы и данные);
- глобальные переменные;
- открытые файлы;
- дочерние процессы;
- необработанные аварийные сигналы;
- сигналы и обработчики сигналов;
- учетную информации.

Поток выполнения имеет:

- ***счетчик команд***, отслеживающий, какую очередную инструкцию нужно выполнять;
- ***регистры***, в которых содержатся текущие рабочие переменные;
- ***стек*** с протоколом выполнения, содержащим по одному фрейму для каждой вызванной, но еще не возвратившей управление процедуры.



Потоки добавляют к модели процесса возможность реализации нескольких выполняемых задач в единой среде процесса. Потоки иногда называют ***облегченными (легковесными) процессами***.

Многопоточный режим допускает работу нескольких потоков в одном и том же процессе.

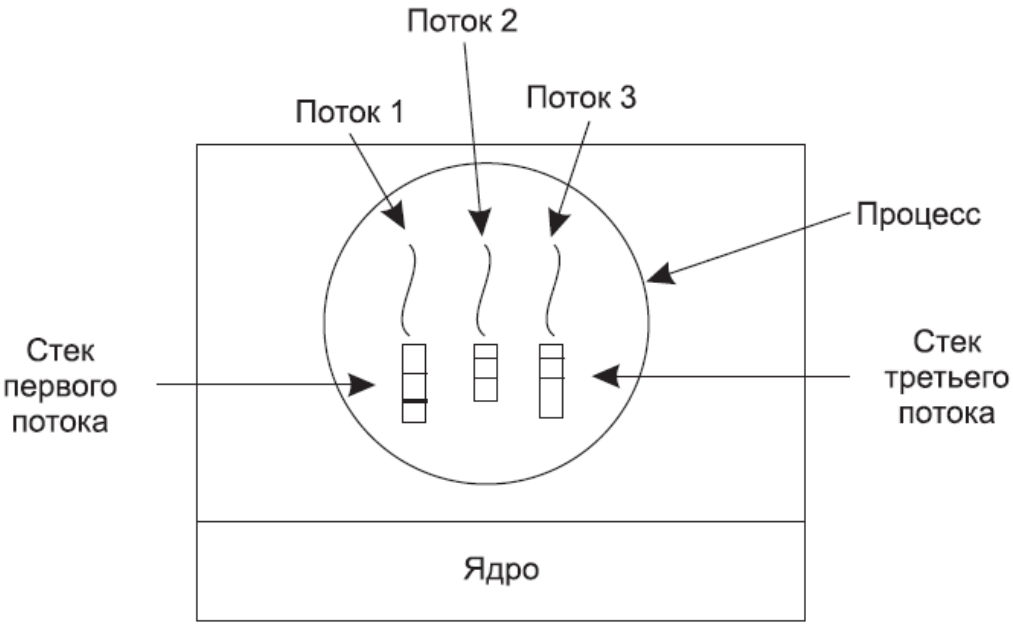
Различные потоки в процессе не обладают той независимостью, которая есть у различных процессов. У потоков одно и то же адресное пространство, один поток может считывать данные из стека другого потока, записывать туда свои данные и даже стирать оттуда данные. Защита между потоками отсутствует, в ней нет необходимости.

Различные **процессы** могут сильно **конкурировать** друг с другом, а **потоки** внутри одного процесса служат для **совместной** работы, активно и тесно **сотрудничают** друг с другом.

Подобно процессу (с одним потоком), поток может быть в одном из следующих состояний:

- выполняемый,
- заблокированный,
- готовый,
- завершенный.

Каждый поток имеет собственный **стек**, который содержит по одному фрейму для каждой уже вызванной, но еще не возвратившей управление процедуры.



2.2. Потоки в POSIX

Для многопоточных программ определен стандарт IEEE standard 1003.1c, в котором имеется пакет ***Pthreads*** (содержит более 60 вызовов функций), поддерживаемый большинством UNIX-систем. Основные вызовы функций:

Вызовы	Описание
`pthread_create`	Создание нового потока
`pthread_exit`	Завершение работы вызвавшего потока
`pthread_join`	Ожидание выхода из указанного потока
`pthread_yield`	Освобождение ЦП, позволяющее выполняться другому потоку
`pthread_attr_init`	Создание и инициализация структуры атрибутов потока
`pthread_attr_destroy`	Удаление структуры атрибутов потока (но не самого потока)

У каждого потока есть

- идентификатор,
- набор регистров (включая счетчик команд),
- набор атрибутов, которые сохраняются в определенной структуре, и включают:
  - размер стека,
  - параметры планирования,
  - другие элементы.

Ожидающий поток сам вызывает функцию `pthread_join`, чтобы ждать завершения другого потока, идентификатор которого указан в качестве параметра этой функции. Для процессов подобных вызовов функций не существует, поскольку предполагается, что процессы сильно конкурируют друг с другом.

Пример программы, использующей потоки

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

#define NUMBER_OF_THREADS 10
void *print_hello_world(void *tid) {
     /* Эта функция выводит идентификатор потока, а затем осуществляет выход */
     printf("Привет, мир. Тебя приветствует поток № %d\n", tid);
     pthread_exit(NULL);
}
int main(int argc, char *argv[]) {
     /* Основная программа создает 10 потоков, а затем осуществляет выход. */
     pthread_t threads[NUMBER_OF_THREADS];
     int status, i;
     for(i=0; i < NUMBER_OF_THREADS; i++) {
       printf("Это основная программа. Создание потока № %d\n", i);
       status = pthread_create(&threads[i], NULL, print_hello_world, (void *)i);
       if (status != 0) {
         printf("Функция pthread_create вернула код ошибки %d\n", status);
         exit(-1);
       }
     }
     exit(NULL);
}
```

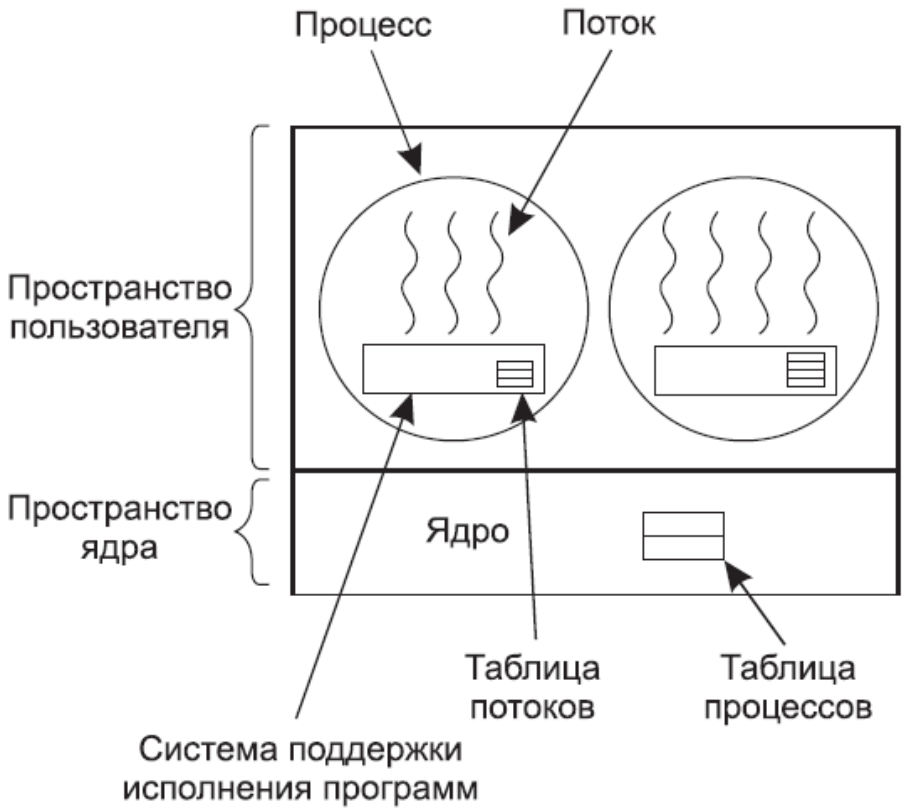
Основная программа (`main`) работает в цикле столько раз, сколько указано в константе `NUMBER_OF_THREADS` (количество потоков), создавая при каждой итерации новый поток и предварительно сообщив о своих намерениях. Если создать поток не удастся, она выводит сообщение об ошибке и выполняет выход. После создания всех потоков осуществляется выход из основной программы.

При создании поток выводит однострочное сообщение, объявляя о своем существовании, после чего осуществляет выход. Порядок, в котором выводятся различные сообщения, не определен и при нескольких запусках программы может быть разным.

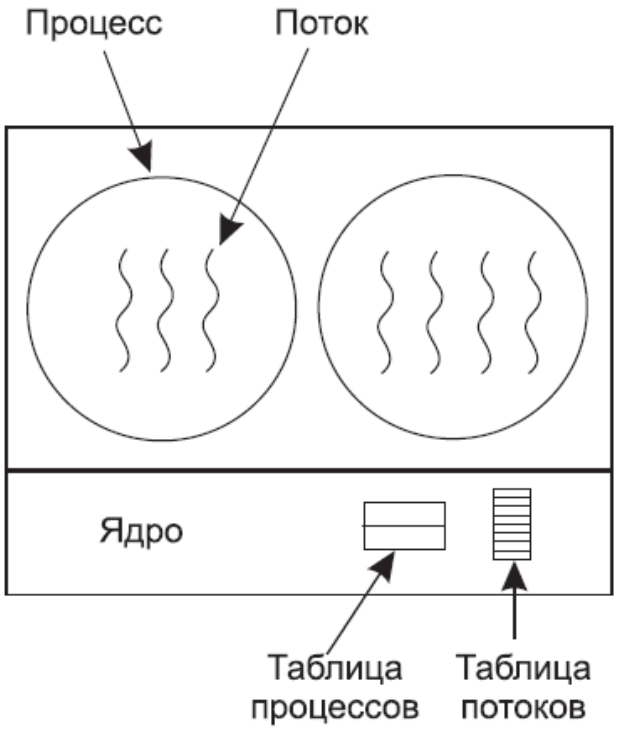
2.3. Реализация потоков

2.3.1. На пользовательском уровне.

Весь набор потоков помещается в пользовательском пространстве. Ядру об этом наборе ничего не известно, оно управляет однопотоковыми процессами. Этот способ может быть реализован в ОС, не поддерживающей потоки, которые реализуются с помощью библиотеки.



Потоки на пользовательском уровне



Потоки, управляемые ядром

Потоки запускаются поверх ***системы поддержки исполнения программ (run-time system)***, которая представляет собой набор процедур, управляющих потоками.

Каждый процесс имеет собственную ***таблицу потоков*** (аналог таблицы процессов ядра), в которой содержатся свойства, принадлежащие каждому потоку (счетчик команд потока, указатель стека, регистры, состояние и т. д.). Таблица потоков управляется системой поддержки исполнения программ. У каждого процесса имеются собственные настройки планировщика.

При переключении поток вызывает локальные процедуры системы поддержки исполнения программ и планировщик потоков, что работает на порядок быстрее, чем перехват управления ядром, осуществляемый инструкцией TRAP.

Но при такой реализации потоков возникают существенные проблемы.

1) Проблема **блокирующих системных вызовов**.

Как позволить каждому потоку использовать блокирующие вызовы, но при этом предотвратить влияние одного заблокированного потока на выполнение других потоков?

Можно заранее сообщить, будет ли вызов блокирующим. Например, в UNIX существует системный вызов `select`, позволяющий сообщить вызывающей программе, будет ли предполагаемый системный вызов `read` блокирующим. Тогда библиотечная процедура `read` заменяется новой процедурой, которая сначала осуществляет вызов процедуры `select` и только потом — вызов `read`, если он безопасен. Если вызов `read` будет блокирующим, он не осуществляется. Вместо этого запускается выполнение другого потока.

Для реализации такого подхода требуется переписать некоторые части библиотеки системных вызовов, сделать системные вызовы ***неблокирующими***. Код, который помещается вокруг системного вызова с целью проверки блокировки, называется ***конвертом (jacket)*** или ***оберткой (wrapper)***.

2) Проблема **ошибки вызова отсутствующей страницы**.

Все программы могут находиться в опеативной памяти в одно и то же время. Если программа вызывает инструкции, отсутствующие в памяти, возникает ошибка обращения к отсутствующей странице, тогда ОС обращается к диску и получает отсутствующие инструкции.

Процесс блокируется ядром до тех пор, пока не будет найдена и считана необходимая инструкция (т.е. пока не завершится дисковая операция ввода-вывода), даже если другие потоки будут готовы к выполнению.

3) Проблема **ошибки вызова отсутствующей страницы**.

Пока выполняющийся поток добровольно не уступит центральный процессор, никакой другой поток, принадлежащий этому же процессу, не сможет выполняться. В рамках единого процесса нет прерываний по таймеру, позволяющих планировать поочередную работу процессов. Издержки на реализацию таких прерываний по таймеру могут быть неприемлемыми.

Программистам потоки обычно требуются в тех приложениях, где они часто блокируются. Как только для выполнения системного вызова ядро осуществит перехват управления, ему не составит особого труда заняться переключением потоков.

2.3.2. На уровне ядра.

У ядра имеется ***таблица потоков***, в которой отслеживаются все потоки в системе. Когда потоку необходимо создать новый или уничтожить существующий поток, он обращается к ядру.

Информация в таблице потоков ядра является подмножеством той информации, которую поддерживает ядро в отношении своих однопоточных процессов, т.е. подмножеством состояния процесса. Кроме того, ядро поддерживает также ***таблицу процессов***.

Когда поток блокируется, ядро по своему выбору может запустить либо другой поток из этого же процесса (если имеется готовый к выполнению поток), либо поток из другого процесса.

Для потоков на уровне ядра не требуется никаких неблокирующих системных вызовов.

При ошибке вызова отсутствующей страницы, ядро может запустить один из готовых к выполнению потоков, пока длится ожидание извлечения запрошенной страницы с диска.

Главный недостаток потоков на уровне ядра – существенные затраты времени на системный вызов (создание, удаление потоков и т.п.)

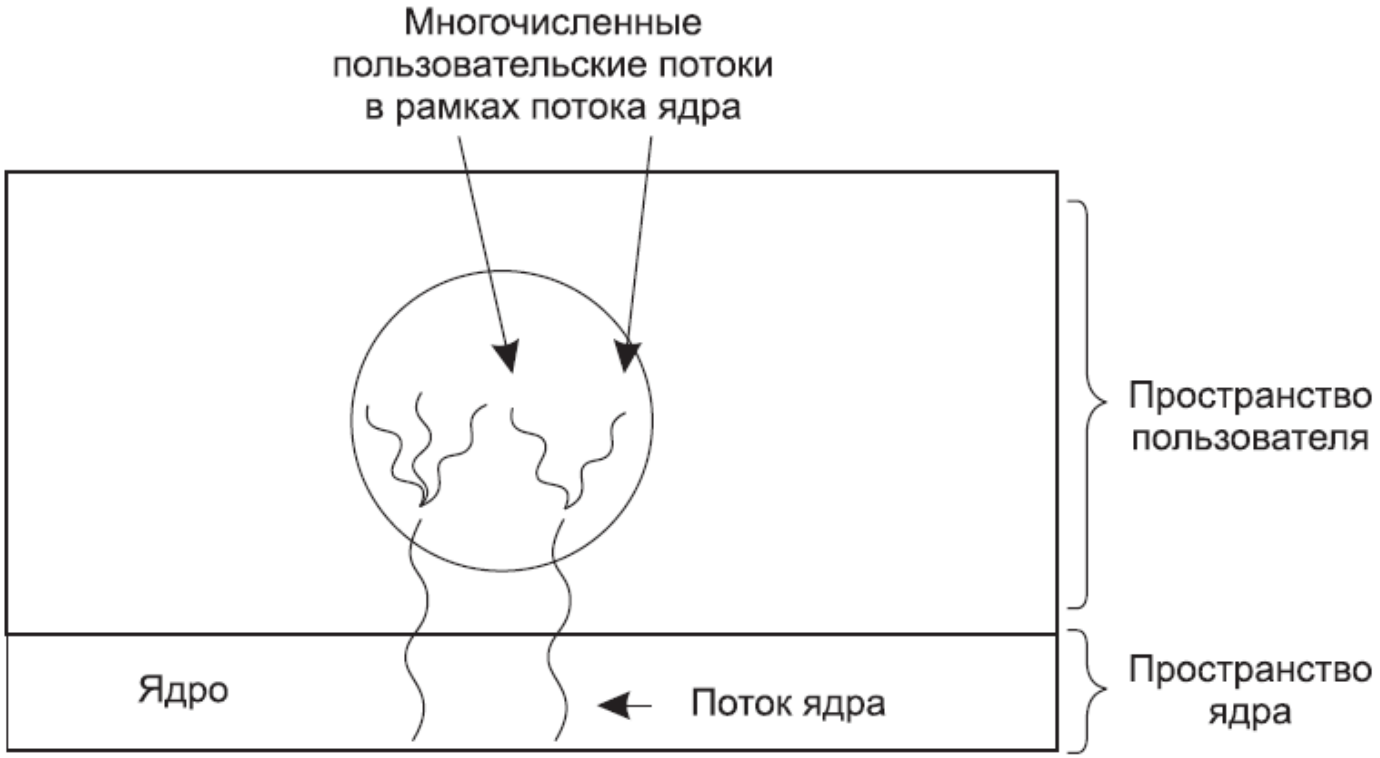
Возникают также другие проблемы.

Например, при разветвлении многопоточного процесса будет ли у нового процесса столько же потоков, сколько у старого, или только один поток.

Другой пример, когда сигналы посылаются процессам, а не потокам, какой из потоков должен обработать поступающий сигнал.

2.3.3. Гибридная реализация.

Создаются потоки на уровне ядра, которые затем разделяются на уровне пользователя. Гибкость достигается, когда программист сам может определить, сколько потоков использовать на уровне ядра и на сколько потоков каждый из них разделяется на уровне пользователя.



Разделение на пользовательские потоки в рамках потока ядра

Ядро планирует работу только потоков ядра, У каждого из них могут быть несколько потоков на пользовательском уровне, которые используют этот поток ядра по очереди.