Операционные системы

Лекция 6

Управление ресурсами в ОС

Концепция процессов и потоков

Одним из основных понятий, связанных с операционными системами, является *процесс* – абстрактное понятие, описывающее работу программы. Все функционирующее на компьютере *программное обеспечение*, включая и операционную систему, можно представить набором процессов.

Задачей ОС является *управление процессами* и ресурсами компьютера или, точнее, организация рационального использования ресурсов в интересах наиболее эффективного выполнения процессов. Для решения этой задачи *операционная система* должна располагать информацией о текущем состоянии каждого процесса и ресурса. Универсальный подход к предоставлению такой информации заключается в создании и поддержке таблиц с информацией *по* каждому объекту управления.

Общее *представление* об этом можно получить из рисунка, на котором показаны таблицы, поддерживаемые операционной системой: для памяти, устройств ввода-вывода, файлов (программ и данных) и процессов. Хотя детали таких таблиц в разных ОС могут отличаться, *по* сути, все они поддерживают информацию *по* этим четырем категориям. Располагающий одними и теми же аппаратными ресурсами, но управляемый различными ОС, *компьютер* может работать с разной степенью эффективности. Наибольшие сложности в управлении ресурсами компьютера возникают в мультипрограммных ОС.

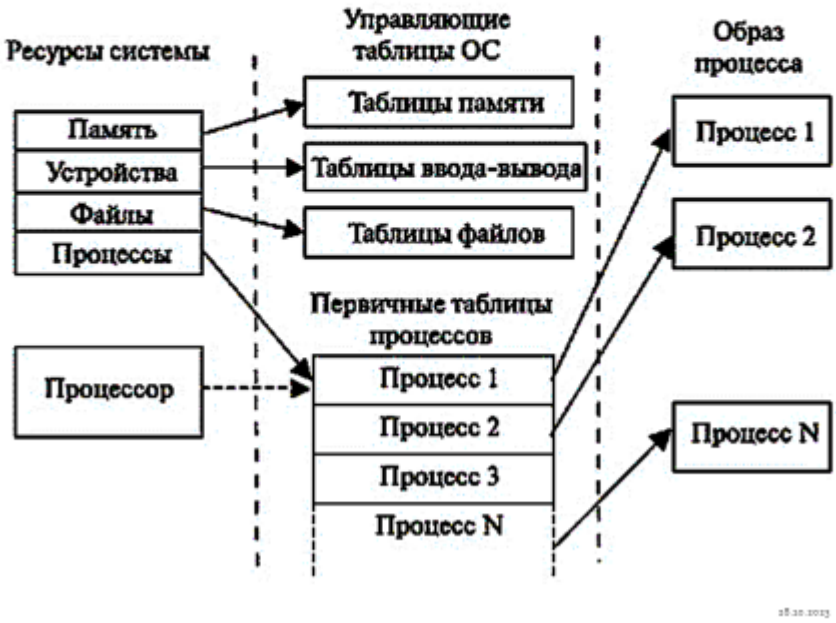


Рис. 5.1. Таблицы ОС

Мультипрограммирование (*многозадачность*, *multitasking*) – это такой способ организации вычислительного процесса, при котором на одном процессоре попеременно выполняются несколько программ. Чтобы поддерживать *мультипрограммирование*, ОС должна определить для себя внутренние единицы работы, между которыми будут разделяться *процессор* и другие ресурсы компьютера. В ОС пакетной обработки, распространенных в компьютерах второго и сначала и третьего поколения, такой единицей работы было задание. В настоящее время в большинстве операционных систем определены два типа единиц работы: более крупная *единица* – процесс, или задача, и менее крупная – *поток*, или *нить*. Причем процесс выполняется в форме одного или нескольких потоков.

Вместе с тем, в некоторых современных ОС вновь вернулись к такой единице работы, как *задание* (*Job*), например, в *Windows*. Задание в *Windows* представляет собой набор из одного или нескольких процессов, управляемых как единое целое. В частности, с каждым заданием ассоциированы *квоты* и *лимиты* ресурсов, хранящиеся в соответствующем объекте задания. Квоты включают такие пункты, как максимальное количество процессов (это не позволяет процессам задания создавать бесконтрольное количество дочерних процессов), суммарное время центрального процессора, доступное для каждого процесса в отдельности и для всех процессов вместе, а также максимальное количество используемой памяти для процесса и всего задания. Задания также могут ограничивать свои процессы в вопросах безопасности,

например, получать или запрещать *права* администратора (даже при наличии правильного пароля).

Процессы рассматриваются операционной системой как заявки или контейнеры для всех видов ресурсов, кроме одного – процессорного времени. Это важнейший *ресурс* распределяется операционной системой между другими единицами работы – потоками, которые и получили свое название благодаря тому, что они представляют собой последовательности (потоки выполнения) команд. Каждый процесс начинается с одного потока, но новые потоки могут создаваться (порождаться) процессом динамически. В простейшем случае процесс состоит из одного потока, и именно таким образом трактовалось понятие "процесс" до середины 80-х годов (например, в ранних версиях *UNIX*). В некоторых современных ОС такое положение сохранилось, т.е. понятие "*поток*" полностью поглощается понятием "процесс".

Как правило, *поток* работает в пользовательском режиме, но когда он обращается к системному вызову, то переключается в режим ядра. После завершения системного вызова *поток* продолжает выполняться в режиме пользователя. У каждого потока есть два стека, один используется в режиме ядра, другой – в режиме пользователя. Помимо состояния (текущие значения всех объектов потока) идентификатора и двух стеков, у каждого потока есть *контекст* (в котором сохраняются его регистры, когда он не работает), приватная область для его локальных переменных, а также может быть собственный маркер доступа (*информация* о защите). Когда *поток* завершает работу, он может прекратить свое существование. Процесс завершается, когда прекратит существование последний *активный поток*.

Взаимосвязь между заданиями, процессами и потоками показана на рисунке

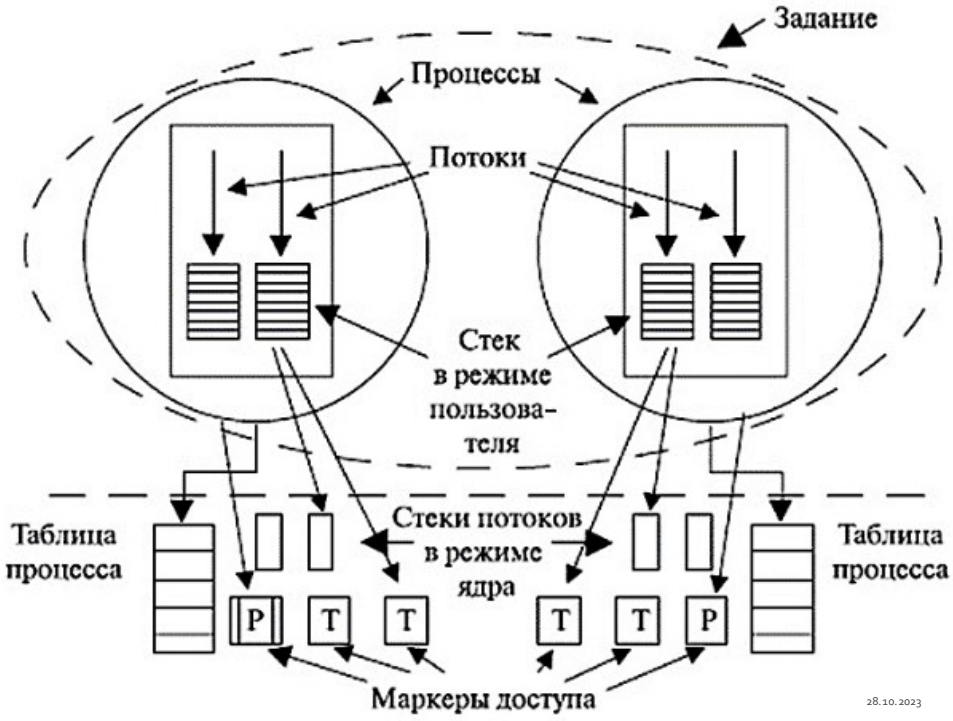


Рис. 5.2. Задания, процессы, потоки

Переключение потоков в ОС занимает довольно много времени, так как для этого необходимы переключение в режим ядра, а затем возврат в *режим пользователя*. Достаточно велики *затраты* процессорного времени на планирование и диспетчеризацию потоков. Для предоставления сильно облегченного *псевдопараллелизма* в *Windows 2000* (и последующих версиях) используются *волокна* (*Fiber*), подобные потокам, но планируемые в пространстве пользователя создавшей их программой. У каждого потока может быть несколько волокон, с той разницей, что когда волокно логически блокируется, оно помещается в *очередь* блокированных волокон, после чего для работы выбирается другое волокно в контексте того же потока. При этом ОС "не знает" о смене волокон, так как все тот же *поток* продолжает работу. Таким образом, существует *иерархия* рабочих единиц операционной системы, которая применительно к *Windows* выглядит следующим образом.

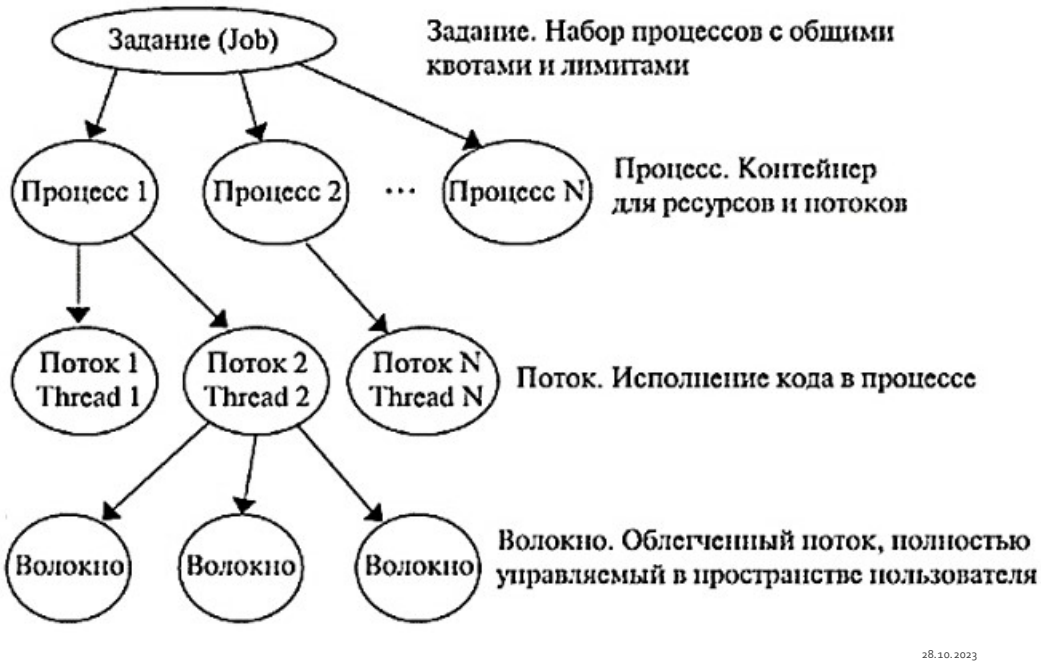


Рис. 5.3. Иерархия рабочих единиц ОС

Возникает вопрос: зачем нужна такая сложная организация *работ*, выполняемых операционной системой? Ответ нужно искать в развитии теории и практики мультипрограммирования, цель которой – в обеспечении максимально эффективного использования главного ресурса вычислительной системы – центрального процессора (нескольких центральных процессоров).

Поэтому прежде, чем переходить к рассмотрению современных принципов управления процессором, процессами и потоками, следует остановиться на основных принципах мультипрограммирования.

Мультипрограммирование. Формы многопрограммной работы

- *Мультипрограммирование* призвано повысить эффективность использования вычислительной системы. Однако эффективность может пониматься *по*-разному. Наиболее характерными показателями эффективности вычислительных систем являются:
  - пропускная способность – количество задач, выполняемых системой в единицу времени;
  - удобство работы пользователей, заключающихся, в частности, в том, что они могут одновременно работать в интерактивном режиме с несколькими приложениями на одной машине;

- *реактивность системы* – способность выдерживать заранее заданные (возможно, очень короткие) интервалы времени между запуском программы и получением конечного результата.

В зависимости от выбора одного из этих показателей эффективности ОС делятся на *системы пакетной обработки*, системы разделения времени и *системы реального времени* (некоторые ОС могут поддерживать одновременно несколько режимов).

Системы *пакетной обработки* предназначались для решения задач в основном вычислительного характера, не требующих быстрого получения результатов

Максимальная *пропускная способность* компьютера достигается в этом случае минимизацией простоев его устройств и прежде всего процессора. Для достижения этой цели *пакет заданий* формируется так, чтобы получающаяся мультипрограммная смесь сбалансированно загружала все устройства машины. Например, в такой смеси желательно присутствие задач вычислительного характера и с интенсивным вводом-выводом. Однако в этом случае трудно гарантировать сроки выполнения того или иного задания.

В благоприятных случаях общее *время выполнения* смеси задач меньше, чем суммарное время их последовательного выполнения. При этом времени выполнения отдельной задачи может быть затрачено больше, чем при монопольном ее выполнении.

В *системах разделения времени* пользователям (в частном случае – одному) предоставляется возможность интерактивной работы сразу с несколькими приложениями. Для этого каждое *приложение* должно регулярно получать возможность "общения" с пользователем. Эта проблема решается за счет того, что ОС принудительно периодически приостанавливает приложения, не дожидаясь, когда они "добровольно" освободят *процессор*.

ИЛЛЮСТРАЦИЯ ЭФФЕКТА МУЛЬТИПРОГРАММИРОВАНИЯ

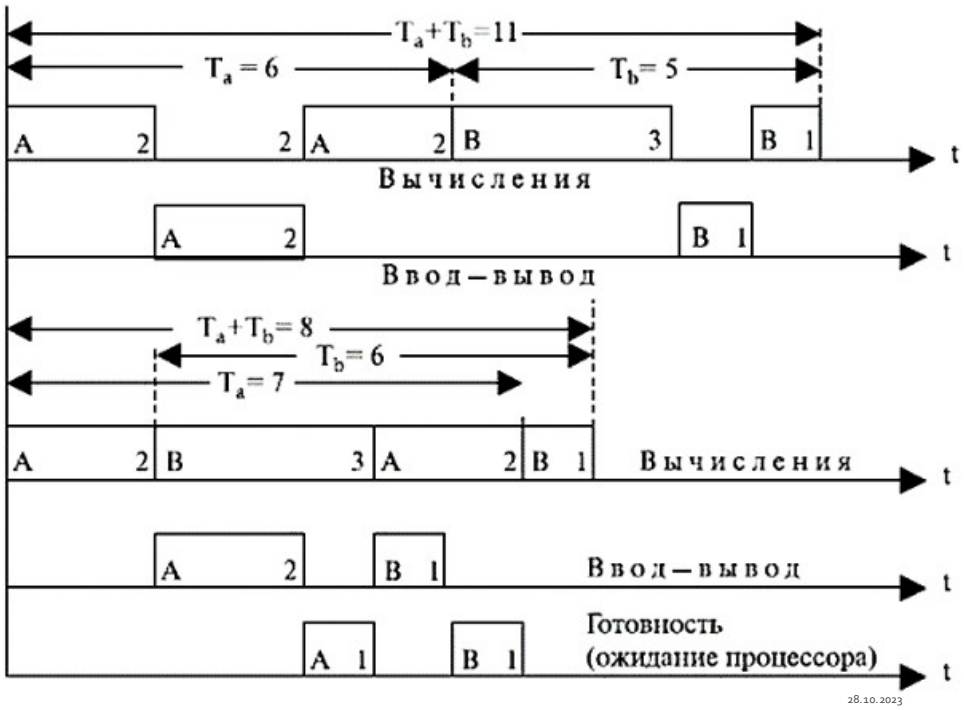
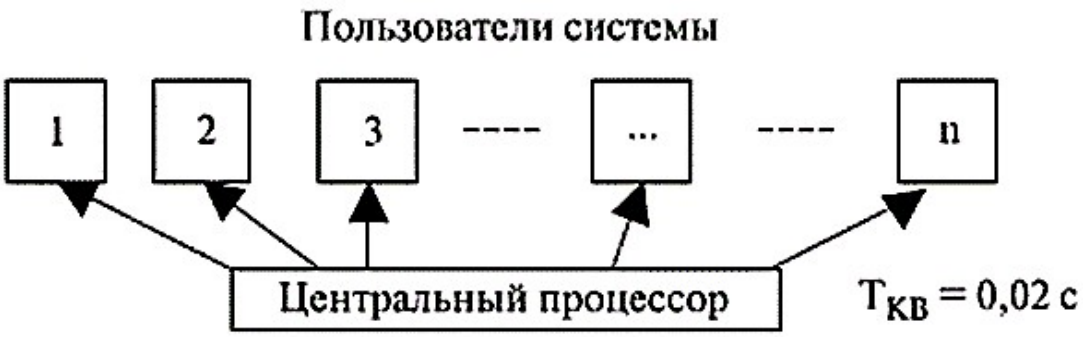


Рис. 5.4. Иллюстрация эффекта мультипрограммирования

Всем приложениям попеременно выделяются кванты времени процессора, таким образом, пользователи, запустившие программы на выполнение, получают возможность поддерживать с ними диалог со своего терминала. Если время кванта выбрано достаточно небольшим, то у всех пользователей складывается впечатление единоличной работы на машине.



28.10.2023

Рис. 5.5. Система разделения времени

Системы *реального времени* предназначены для управления техническими объектами (спутник, ракета, атомные электростанции, станок, научная установка и др.), технологическими процессами (гальваническая линия, доменный процесс и т.п.), системами обслуживания разного рода (резервирование авиабилетов, *оплата* покупок и счетов и др.). Во всех этих случаях существует предельно допустимое время, в течение которого должна быть выполнена та или иная *программа* управления объектом. В противном случае возможны нежелательные последствия вплоть до аварии.

Критерием эффективности ОС в этом случае является способность выдерживать заранее заданные интервалы времени между запуском программы и получением результата. Это время называется временем реакции системы, а соответствующее свойство – реактивностью. Требования ко времени реакции зависят от специфики управляемого объекта или процесса. В системах реального времени мультипрограммная смесь представляет собой фиксированный набор заранее разработанных программ решения функциональных задач управления объектом или процессом. Выбор программы на выполнение осуществляется *по* прерываниям (исходя из текущего состояния объекта) или в соответствии с расписанием плановых *работ*.

В системе реального времени обычно закладывается запас вычислительной мощности на случай пиковой нагрузки, а также принимаются

меры обеспечения высокой надежности работы системы (резервирование, дублирование, троирование с мажоритарным элементом и др.).

Интересная форма мультипрограммной работы связана с *мультипроцессорной обработкой*. Мультипроцессорная обработка – это способ организации вычислительного процесса в системе с несколькими процессорами, при котором несколько задач (процессов, потоков) могут одновременно выполняться на разных процессорах системы. Концепция мультипроцессирования не нова, она известна с 70-х годов, однако стала доступной в широком масштабе лишь в последнее десятилетие, особенно с появлением многопроцессорных ПК (часто в качестве серверов *ЛВС*).

В отличие от мультипрограммной обработки, в мультипроцессорных системах несколько задач выполняется одновременно, т.к. имеется несколько процессоров. Однако это не исключает мультипрограммной обработки на каждом процессоре. При этом резко усложняются все алгоритмы управления ресурсами, т.е. *операционная система*. Современные ОС, как правило, поддерживают мультипроцессирование (*Sun* Solaris 2.x, Santa Cruz *Operation Open Server* 3.x, *OS/2*, *Windows* NT/2000/2003/XP, NetWare, начиная с версии 4.1 и др.).

Мультипроцессорные системы часто характеризуют как *симметричные* и как *несимметричные*. Эти термины относятся, с одной стороны, к архитектуре вычислительной системы, а с другой – к способу организации вычислительного процесса.

Симметричная *архитектура* мультипроцессорной системы предполагает однотипность и единообразие включения процессоров и большую разделяемую между этими процессорами *память*. Масштабируемость, т.е. возможность наращивания числа процессоров, в данном случае ограничена, т.к. все они используют одну и ту же оперативную *память* и, следовательно, должны располагаться в одном корпусе. В *симметричных* архитектурах вычислительных систем легко реализуется *симметричное мультипроцессирование* общей для всех процессоров операционной системой. При этом все процессоры равноправно участвуют и в управлении вычислительным процессом, и в выполнении прикладных задач. Разные процессоры могут в какой-то момент времени одновременно обслуживать как разные, так и одинаковые модули общей ОС. Для этого программы ОС должны быть *реентерабельными* (повторновходимыми).

Операционная система полностью *децентрализована*. Ее модули выполняются на любом доступном процессоре. Как только *процессор* завершает выполнение очередной задачи, он передает управление планировщику задач. Последний выбирает из общей для всех процессоров *системной очереди* задачу, которая будет выполняться на данном процессоре следующей.

В вычислительных системах с *асимметричной* архитектурой процессоры могут быть различными как *по* характеристикам

(*производительность*, *система команд*), так и *по* функциональной роли в работе системы. Например, могут быть выделены процессоры для вычислений, ввода-вывода и др. Эта неоднородность ведет к структурным отличиям во фрагментах системы, содержащих разные процессоры (разные схемы подключения, наборы периферийных устройств, способы взаимодействия процессоров с устройствами и др.).

Масштабирование в таких системах реализуется иначе, поскольку отсутствует требование единого корпуса. Система может состоять из нескольких устройств, каждое из которых содержит один или несколько процессоров. Масштабирование в данном случае называют горизонтальным, а мультипроцессорную систему – кластерной. В кластерной системе может быть реализовано только асимметричное мультипроцессирование с организацией вычислительного процесса *по* принципу "ведущий – ведомый". Этот наиболее простой способ может быть использован и в вычислительных системах с симметричной архитектурой. В таких системах ОС работает на одном процессоре, который называется ведущим и организует *централизованное управление* вычислительным процессом и распределением всех ресурсов системы.

Управление процессами и потоками

ОСНОВНЫЕ ФУНКЦИИ ПОДСИСТЕМЫ УПРАВЛЕНИЯ ПРОЦЕССАМИ И ПОТОКАМИ.

- создание процессов и потоков;
- обеспечение процессов и потоков необходимыми ресурсами;
- изоляция процессов;
- планирование выполнения процессов и потоков (вообще, следует говорить и о планировании заданий);
- диспетчеризация потоков;
- организация межпроцессного взаимодействия;
- синхронизация процессов и потоков;
- завершение и уничтожение процессов и потоков.

28.10.2023

Одной из основных подсистем любой современной мультипрограммной ОС, непосредственно влияющей на функционирование компьютера, является

подсистема управления процессами и потоками. Основные функции этой подсистемы:

- создание процессов и потоков;
- обеспечение процессов и потоков необходимыми ресурсами;
- изоляция процессов;
- планирование выполнения процессов и потоков (вообще, следует говорить и о планировании заданий);
- диспетчеризация потоков;
- организация межпроцессного взаимодействия;
- синхронизация процессов и потоков;
- завершение и уничтожение процессов и потоков.

К созданию процесса приводят пять основных событий:

1. инициализация ОС (загрузка);
2. выполнение запроса работающего процесса на создание процесса;
3. запрос пользователя на создание процесса, например, при входе в систему в интерактивном режиме;
4. инициирование пакетного задания;
5. создание операционной системой процесса, необходимого для работы каких-либо служб.

Обычно при загрузке ОС создаются несколько процессов. Некоторые из них являются высокоприоритетными процессами, обеспечивающими взаимодействие с пользователями и выполняющими заданную работу. Остальные процессы являются фоновыми, они не связаны с конкретными пользователями, но выполняют особые функции – например, связанные с электронной почтой, Web-страницами, выводом на *печать*, передачей файлов *по* сети, периодическим запуском программ (например, *дефрагментации дисков*) и т.д. Фоновые процессы называют демонами.

Новый процесс может быть создан *по* запросу текущего процесса. Создание новых процессов полезно в тех случаях, когда выполняемую задачу проще всего сформировать как набор связанных, но, тем не менее, независимых взаимодействующих процессов. В интерактивных системах *пользователь* может запустить программу, набрав на клавиатуре команду или дважды щелкнув на значке программы. В обоих случаях создается новый процесс и *запуск* в нем программы. В *системах пакетной обработки* на мэйнфреймах пользователи посылают задание (возможно, с использованием удаленного доступа), а ОС создает новый процесс и запускает следующее задание из очереди, когда освобождаются необходимые ресурсы.

С технической точки зрения во всех перечисленных случаях новый процесс формируется одинаково: текущий процесс выполняет системный *запрос* на создание нового процесса. Подсистема управления процессами и потоками отвечает за обеспечение процессов необходимыми ресурсами. ОС поддерживает в памяти специальные информационные структуры, в которые

записывает, какие ресурсы выделены каждому процессу. Она может назначить процессу ресурсы в единоличное пользование или совместное пользование с другими процессами. Некоторые из ресурсов выделяются процессу при его создании, а некоторые – динамически *по* запросам во *время выполнения*. Ресурсы могут быть выделены процессу на все время его жизни или только на определенный период. При выполнении этих функций подсистема управления процессами взаимодействует с другими подсистемами ОС, ответственными за *управление ресурсами*, такими как подсистема управления памятью, *подсистема ввода-вывода*, *файловая система*.

Для того чтобы процессы не могли вмешаться в *распределение ресурсов*, а также не могли повредить коды и данные друг друга, важнейшей задачей ОС является изоляция одного процесса от другого. Для этого *операционная система* обеспечивает каждый процесс отдельным виртуальным адресным пространством, так что ни один процесс не может получить прямого доступа к командам и данным другого процесса.

В ОС, где существуют процессы и потоки, процесс рассматривается как заявка на потребление всех видов ресурсов, кроме одного – процессорного времени. Этот важнейший *ресурс* распределяется операционной системой между другими единицами работы – потоками, которые и получили свое название благодаря тому, что они представляют собой последовательности (потоки выполнения) команд. Переход от выполнения одного потока к другому осуществляется в результате *планирования* и *диспетчеризации*. Работа *по* определению момента, в который необходимо прервать выполнение текущего потока, и потока, которому следует предоставить возможность выполняться, называется планированием. Планирование потоков осуществляется на основе информации, хранящейся в описателях процессов и потоков. При планировании принимается во внимание *приоритет потоков*, время их ожидания в очереди, накопленное *время выполнения*, интенсивность обращения к вводу-выводу и другие факторы.

Диспетчеризация заключается в реализации найденного в результате планирования решения, т.е. в переключении процессора с одного потока на другой. Диспетчеризация проходит в три этапа:

- сохранение контекста текущего потока;
- загрузка контекста потока, выбранного в результате планирования;
- запуск нового потока на выполнение.

Когда в системе одновременно выполняется несколько независимых задач, возникают дополнительные проблемы. Хотя потоки возникают и выполняются синхронно, у них может возникнуть необходимость во взаимодействии, например, при обмене данными. Для общения друг с другом процессы и потоки могут использовать широкий спектр возможностей: каналы (в *UNIX*), почтовые ящики (*Windows*), вызов удаленной процедуры, сокеты (в *Windows* соединяют процессы на разных машинах). Согласование скоростей потоков также очень важно для предотвращения эффекта "гонок" (когда

несколько потоков пытаются изменить один и тот же *файл*), взаимных блокировок и других коллизий, которые возникают при совместном использовании ресурсов.

Синхронизация потоков является одной из важнейших функций подсистемы управления процессами и потоками. Современные операционные системы предоставляют множество механизмов синхронизации, включая семафоры, мьютексы, критические области и события. Все эти *механизмы* работают с потоками, а не с процессами. Поэтому когда *поток* блокируется на семафоре, другие потоки этого процесса могут продолжать работу.

Каждый раз, когда процесс завершается, – а это происходит благодаря одному из следующих событий: обычный *выход, выход по* ошибке, *выход по* неисправимой ошибке, уничтожение другим процессом – ОС предпринимает шаги, чтобы "зачистить следы" его пребывания в системе. Подсистема управления процессами закрывает все файлы, с которыми работал процесс, освобождает области оперативной памяти, отведенные под коды, данные и системные информационные структуры процесса. Выполняется *коррекция* всевозможных очередей ОС и *список* ресурсов, в которых имелись ссылки на завершаемый процесс.

Как уже отмечалось, чтобы поддержать *мультипрограммирование*, ОС должна оформить для себя те внутренние единицы работы, между которыми будет разделяться *процессор* и другие ресурсы компьютера. Возникает вопрос: в чем принципиальное отличие этих единиц работы, какой эффект мультипрограммирования можно получить от их применения и в каких случаях эти единицы *работ* операционной системы следует создавать?

Очевидно, что любая работа вычислительной системы заключается в выполнении некоторой программы. Поэтому и с процессом, и с потоком связывается определенный программный код, который оформляется в виде исполняемого модуля. В простейшем случае процесс состоит из одного потока, и в некоторых современных ОС сохранилось такое положение. *Мультипрограммирование* в таких ОС осуществляется на уровне процессов. При необходимости взаимодействия процессы обращаются к операционной системе, которая, выполняя функции посредника, предоставляет им средства межпроцессной связи – каналы, почтовые акции, разделяемые секции памяти и др.

Однако в системах, в которых отсутствует понятие потока, возникают проблемы при организации параллельных вычислений в рамках процесса. А такая необходимость может возникать. Дело в том, что отдельный процесс никогда не может быть выполнен быстрее, чем в однопрограммном режиме. Однако *приложение*, выполняемое в рамках одного процесса, может обладать *внутренним параллелизмом*, который, в принципе, мог бы ускорить его решение. Если, например, в программе предусмотрено обращение к внешнему устройству, то на время этой *операции* можно не блокировать выполнение всего процесса, а продолжить вычисления *по* другой ветви программы.

Параллельное выполнение нескольких *работ* в рамках одного интерактивного приложения повышает эффективность работы пользователя. Так, при работе с текстовым редактором желательно иметь возможность совмещения набора нового текста с такими продолжительными операциями, как переформатирование значительной части текста, сохранение его на локальном или удаленном диске.

Нетрудно представить будущую версию компилятора, способную автоматически компилировать файлы исходного кода в паузах, возникающих при наборе текста программы. Тогда предупреждения и сообщения об ошибках появлялись бы в режиме реального времени, и *пользователь* тут же видел бы, в чем он ошибся. Современные электронные таблицы пересчитывают данные в фоновом режиме, как только *пользователь* что-либо изменил. Текстовые процессоры разбивают текст на страницы, проверяют его на орфографические и грамматические ошибки, печатают в фоновом режиме, сохраняют текст каждые несколько минут и т.д. Во всех этих случаях потоки используются как средство распараллеливания вычислений.

Эти задачи можно было бы возложить на программиста, который должен был бы написать программу-*диспетчер*, реализующую *параллелизм* в рамках одного процесса. Однако это весьма сложно, да и сама *программа* получилась бы весьма запутанной и сложной в отладке.

Другим решением является создание для одного приложения нескольких процессов для каждой из параллельных *работ*. Однако использование для создания процессов стандартных средств ОС не позволяет учесть тот факт, что процессы решают единую задачу и имеют много общего: работают с одними и теми же данными, используют один и тот же кодовый сегмент, имеют одни и те же *права* доступа к ресурсам вычислительной системы. А *операционная система* при таком подходе будет рассматривать эти процессы наравне со всеми остальными процессами и обеспечивать их изоляцию друг от друга. В данном случае это будет не только бесполезная, но и вредная работа, затрудняющая *обмен данными* между различными частями приложения. Кроме того, на создание каждого процесса ОС тратит определенные системные ресурсы, которые в данном случае неоправданно дублируются – каждому процессу выделяется собственное *виртуальное адресное пространство*, физическая *память*, закрепляются устройства ввода-вывода и т.п.

Из изложенного следует *вывод*, что операционной системе наряду с процессами нужен другой механизм распараллеливания вычислений, который учитывал бы тесные связи между отдельными ветвями вычислений одного и того же приложения. Для этих целей современные ОС предлагают механизм многопоточной обработки (*multithreading*).

Понятию "*поток*" соответствует последовательный переход процессора от одной команды к другой. Процессу ОС назначают *адресное пространство* и набор ресурсов, которые совместно используются всеми его потоками. В отличие от процессов, которые принадлежат, вообще говоря, конкурирующим

приложениям, все потоки одного процесса всегда принадлежат одному приложению, поэтому ОС изолирует потоки в гораздо меньшей степени, чем процессы в традиционной мультипрограммной системе. Все потоки одного процесса используют общие файлы, таймеры, устройства, одну и ту же область оперативной памяти, одно и то же *адресное пространство*.

Это означает, что они разделяют одни и те же *глобальные переменные*. Поскольку каждый *поток* может иметь *доступ* к любому виртуальному адресу, один *поток* может задействовать *стек* другого потока. Между потоками одного процесса нет полной защиты, во-первых, потому что это невозможно, а во-вторых, потому что не нужно. Чтобы организовать взаимодействие и *обмен данными*, потокам 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.

Таким образом, *мультипрограммирование* более эффективно на уровне потоков, а не процессов. Еще больший эффект многопоточной обработки достигается в мультипроцессорных системах, в которых потоки могут выполняться на разных процессорах действительно параллельно.

Создание процессов и потоков. Модели процессов и потоков

Создать процесс – это, прежде всего, создать описатель процесса: несколько информационных структур, содержащих все сведения (атрибуты) о процессе, необходимые операционной системе для управления им. В число таких сведений могут входить: *идентификатор* процесса, данные о расположении в памяти исполняемого модуля, степень привилегированности процесса (приоритет и *права* доступа) и т.п.

Примерами таких описателей процесса являются [10, 17]:

- блок управления задачей (TCB – Task *Control Block*) в OS/360;
- управляющий блок процесса (*PCB – Process Control* Block) в OS/2;
- дескриптор процесса в UNIX;
- объект-процесс (object-process) в Windows NT/2000/2003.

Создание процесса включает загрузку кодов и данных исполняемой программы данного процесса с диска в операционную *память*. Для этого нужно найти эту программу на диске, перераспределить оперативную *память* и выделить *память* исполняемой программе нового процесса. Кроме того, при работе программы обычно используется *стек*, с помощью которого реализуются вызовы процедур и *передача параметров*.

Множество, в которое входят *программа*, данные, стеки и атрибуты процесса, называется образом процесса.

Типичные элементы образа процесса приведены на слайде.

Информация	Описание
Данные пользователя	Изменяемая часть пользовательского адресного пространства (данные программы, пользовательский стек, модифицируемый код)
Пользовательская программа	Программа, которую необходимо выполнить
Системный стек	Один или несколько системных стеков для хранения параметров и адресов вызова процедур и системных служб
Управляющий блок процесса	Данные, необходимые операционной системе для управления процессом

Местонахождение образа процесса зависит от используемой схемы управления памятью. В большинстве современных ОС с виртуальной памятью образ процесса состоит из набора блоков (*сегменты*, страницы или их комбинация), не обязательно расположенных последовательно. Такая *организация памяти* позволяет иметь в основной памяти лишь часть образа процесса (активная часть), в то время как во вторичной памяти находится полный образ. Когда в основную *память* загружается часть образа, она туда не переносится, а копируется. Однако если часть образа в основной памяти модифицируется, она должна быть скопирована на *диск*.

При управлении процессами ОС использует два основных типа информационных структур: *блок управления процессом* (*дескриптор* процесса) и *контекст процесса*. Дескрипторы процессов объединяются в таблицу процессов, которая размещается в области ядра. На основании информации, содержащейся в таблице процессов, ОС осуществляет планирование и синхронизацию процессов.

В дескрипторе (*блоке управления) процесса* содержится такая *информация* о процессе, которая необходима ядру в течение всего жизненного *цикла* процесса независимо от того, находится он в активном или пассивном состоянии и находится ли образ в оперативной памяти или на диске. Эту информацию можно разделить на три категории:

- информация по идентификации процесса;
- информация по состоянию процесса;
- информация, используемая при управлении процессом.

Каждому процессу присваивается числовой *идентификатор*, который может быть просто индексом в первичной таблице процессов. В любом случае должно существовать некоторое *отображение*, позволяющее операционной системе найти *по* идентификатору процесса соответствующие ему таблицы.

При создании нового процесса идентификаторы указывают родительский и дочерние процессы. В операционных системах, не поддерживающих иерархию процессов, например, в Windows 2000, все созданные процессы равноправны, но один из 18-ти параметров, возвращаемых вызывающему (родительскому) процессу, представляет собой дескриптор нового процесса. Кроме того, процессу может быть присвоен идентификатор пользователя, который указывает, кто из пользователей отвечает за данное задание.

ИНФОРМАЦИЯ ПО СОСТОЯНИЮ И УПРАВЛЕНИЮ ПРОЦЕССОМ

- состояние процесса, определяющее готовность процесса к выполнению (выполняющийся, готовый к выполнению, ожидающий какогелибо события, приостановленный);
- данные о приоритете (текущий приоритет, по умолчанию, максимально возможный);
- информация о событиях– идентификация события, наступление которого позволит продолжить выполнение процесса;
- указатели, позволяющие определить расположение образа процесса в оперативной памяти и на диске;
- указатели на другие процессы (в частности, находящиеся в очереди на выполнение);
- флаги, сигналы и сообщения, имеющие отношение к обмену информацией между двумя независимыми процессами;
- данные о привилегиях, определяющих права доступа к определенной области памяти или возможности выполнять определенные виды команд, использовать системные утилиты и службы;
- указатели на ресурсы, которыми управляет процесс (например, перечень открытых файлов);
- сведения по истории использования ресурсов и процессора;
- информация, связанная с планированием.

28.10.2023

Информация по состоянию и управлению процессом включает следующие основные данные:

- состояние процесса, определяющее готовность процесса к выполнению (выполняющийся, готовый к выполнению, ожидающий какого-либо события, приостановленный);
- данные о приоритете (текущий приоритет, по умолчанию, максимально возможный);
- информация о событиях – идентификация события, наступление которого позволит продолжить выполнение процесса;
- указатели, позволяющие определить расположение образа процесса в оперативной памяти и на диске;
- указатели на другие процессы (в частности, находящиеся в очереди на выполнение);

- флаги, сигналы и сообщения, имеющие отношение к обмену информацией между двумя независимыми процессами;
- данные о привилегиях, определяющих права доступа к определенной области памяти или возможности выполнять определенные виды команд, использовать системные утилиты и службы;
- указатели на ресурсы, которыми управляет процесс (например, перечень открытых файлов);
- сведения по истории использования ресурсов и процессора;
- информация, связанная с планированием. Эта информация во многом зависит от алгоритма планирования. Сюда относятся, например, такие данные, как время ожидания или время, в течение которого процесс выполнялся при последнем запуске, количество выполненных операций ввода-вывода и др.

Контекст процесса содержит информацию, позволяющую системе приостанавливать и возобновлять выполнение процесса с прерванного места.

В *контексте процесса* содержится следующая основная *информация*:

- содержимое регистров процессора, доступных пользователю;
- содержимое счетчика команд;
- состояние управляющих регистров и регистров состояния;
- коды условий, отражающие результат выполнения последней арифметической или логической операции (например, знак равенства нулю, переполнения);
- указатели вершин стеков, хранящие параметры и адреса вызова процедур и системных служб.

Следует заметить, что часть этой информации, известная как "*слово* состояния программы" (Program *Status Word – PSW*), фиксируется в специальном регистре процессора (например, в регистре EFLAGS в процессорах Pentium).

Самую простую модель процесса можно построить исходя из того, что в любой момент времени процесс либо выполняется, либо не выполняется, т.е. имеет только два состояния. Если бы все процессы были бы всегда готовы к выполнению, то *очередь по* этой схеме могла бы работать вполне эффективно. Такая *очередь* работает *по* принципу обработки в порядке поступления, а *процессор* обслуживает имеющиеся в наличии процессы круговым методом (*Round-robin*). Каждому процессу отводится определенный промежуток времени, *по* истечении которого он возвращается в *очередь*.

Однако в таком простом примере подобная реализация не является адекватной: часть процессов готова к выполнению, а часть заблокирована, например, *по* причине ожидания ввода-вывода. Поэтому при наличии одной очереди *диспетчер* не может просто выбрать для выполнения первый процесс из очереди. Перед этим он должен будет просматривать весь *список*, отыскивая незаблокированный процесс, который находится в очереди дальше других.

Отсюда представляется естественным разделить все невыполняющиеся процессы на два типа: готовые к выполнению и заблокированные. Полезно добавить еще два состояния, как показано на слайде

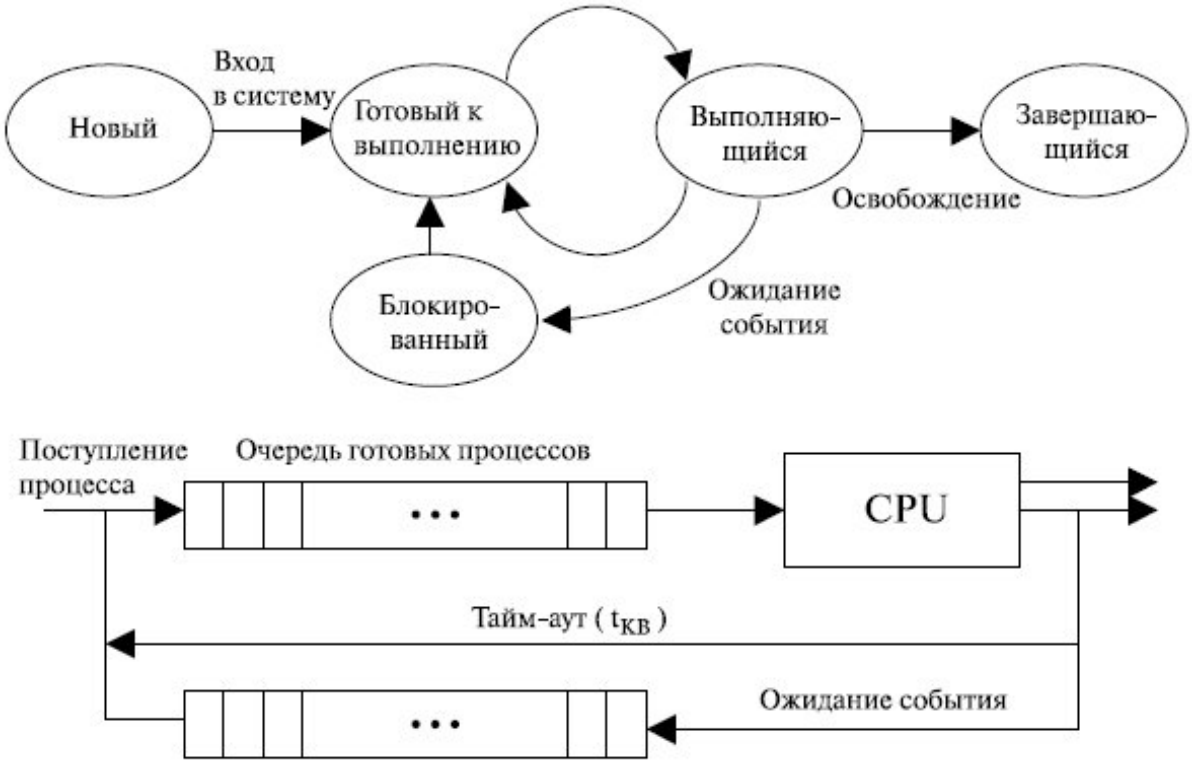


Рис. 5.6. Состояния процесса

В чем достоинства и недостатки такой модели и как устранить эти недостатки? Поскольку *процессор* работает намного быстрее выполнения операций ввода-вывода, то вскоре все находящиеся в памяти процессы оказываются в состоянии ожидания ввода-вывода. Таким образом, *процессор* может простаивать даже в многозадачной системе. Что делать? Можно увеличить емкость основной памяти, чтобы в ней умещалось больше процессов.

Но такой подход имеет два недостатка: во-первых, возрастает *стоимость* памяти, а во-вторых, "аппетит" программиста в использовании памяти возрастает пропорционально ее объему, так что увеличение объема памяти приводит к увеличению размера процессов, а не к росту их числа. Другое решение проблемы – *свопинг*, перенос части процессов из оперативной памяти на *диск* и *загрузка* другого процесса из очереди приостановленных (но не *блокированных!) процессов*, находящихся во внешней памяти. На этом мы прервем рассмотрение модели процессов и их выполнения. Как уже отмечалось, более эффективными являются *многопоточные системы*. В таких системах при создании процесса ОС создается для каждого процесса *минимум* один *поток* выполнения.

При создании потоков, так же как и при создании процессов, ОС генерирует специальную информационную структуру – *описатель потока*, который содержит *идентификатор* потока, данные о правах доступа и

приоритете, о состоянии потока и другую информацию. Описатель потока можно разделить на две части: *атрибуты* блока управления и *контекст* потока. Заметим, что в случае многопоточной системы процессы контекста не имеют, так как им не выделяется *процессор*.

Есть два способа реализации пакета потоков [17]:

в пространстве пользователя или на уровне пользователя (User-level threads – ULT);

в ядре или на уровне ядра (kernel-level threads – KLT).

Рассмотрим эти способы, их преимущества и недостатки.

В программе, полностью состоящей из ULT-потоков, все действия *по* управлению потоками выполняются самим приложением. *Ядро* о потоках ничего не знает и управляет обычными однопоточными процессами.

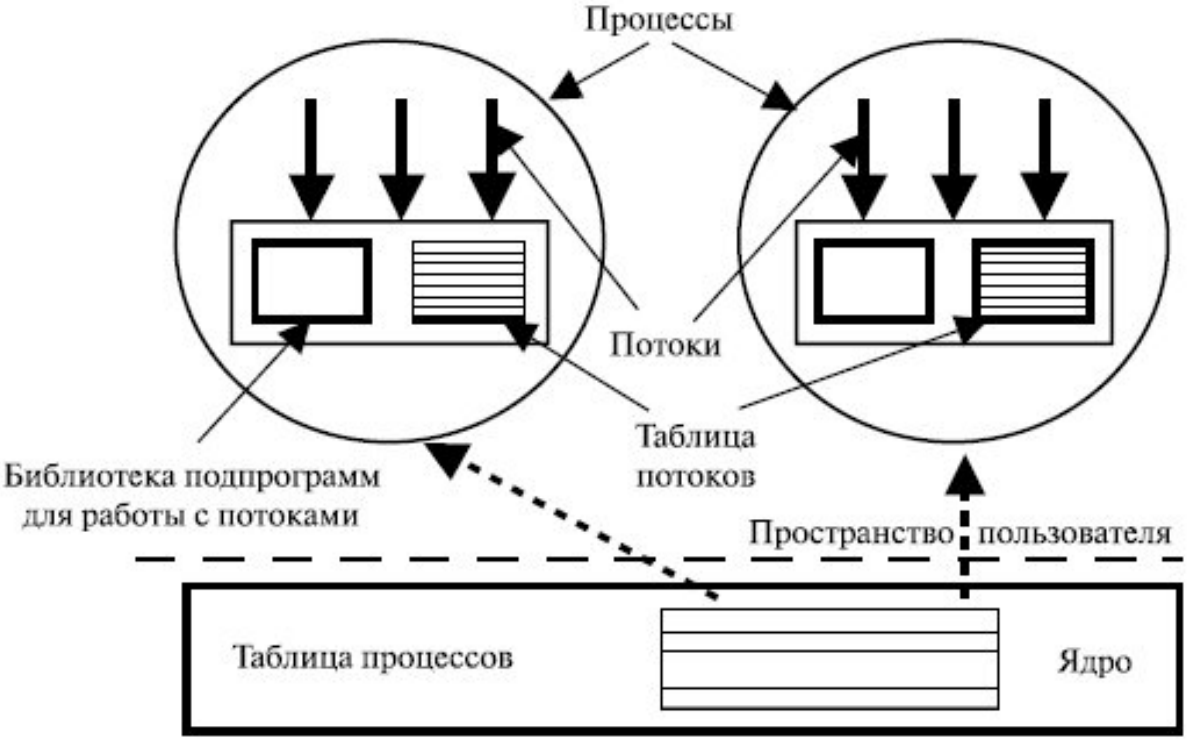


Рис. 5.7. Потоки в пространстве пользователя

Наиболее явное преимущество этого подхода состоит в том, что пакет потоков на уровне пользователя можно реализовать даже в ОС, не поддерживающей потоки.

Если управление потоками происходит в пространстве пользователя, каждому процессу необходима собственная *таблица* потоков. Она аналогична таблице процессов с той лишь разницей, что отслеживает такие характеристики потоков, как *счетчик* команд, *указатель* вершины стека, регистры состояния и т. п. Когда *поток* переходит в состояние готовности или блокировки, вся *информация*, необходимая для повторного запуска, хранится в таблице потоков.

По умолчанию *приложение* в начале своей работы состоит из одного потока и его выполнение начинается как выполнение этого потока. Такое *приложение* вместе с составляющим его потоком размещается в одном процессе, который управляется ядром. Выполняющийся *поток* может породить новый *поток*, который будет выполняться в пределах того же процесса. Новый *поток* создается с помощью вызова специальной подпрограммы из библиотеки, предназначенной для работы с потоками. Передача управления этой программе происходит в результате вызова соответствующей процедуры.

Таких процедур может быть *по* крайней мере четыре: thread-create, thread-exit, thread-wait и thread-*yield,* но обычно их больше. Библиотека подпрограмм для работы с потоками создает структуру данных для нового потока, а потом передает управление одному из готовых к выполнению потоков данного процесса, руководствуясь некоторым алгоритмом планирования. Когда управление переходит к библиотечной программе, *контекст* текущего процесса сохраняется в таблице потоков, а когда управление возвращается к потоку, его *контекст* восстанавливается. Все эти события происходят в пользовательском пространстве в рамках одного *процесса. Ядро* даже "не подозревает" об этой деятельности и продолжает осуществлять планирование процесса как единого целого и приписывать ему единое состояние выполнения.

Использование потоков на уровне пользователя имеет следующие преимущества:

высокая производительность, поскольку для управления потоками процессу не нужно переключаться в режим ядра и обратно. Процедура, сохраняющая информацию о потоке, и планировщики являются локальными процедурами, их вызов существенно более эффективен, чем вызов ядра;

имеется возможность использования различных алгоритмов планирования потоков в различных приложениях (процессах) с учетом их специфики;

использование потоков на пользовательском уровне применимо для любой операционной системы. Для их поддержки в ядро системы не требуется вносить каких-либо изменений.

Однако имеются и недостатки *по* сравнению с использованием потоков на уровне ядра:

в типичной ОС многие системные вызовы являются блокирующими. Когда в потоке, работающем на пользовательском уровне, выполняется системный вызов, блокируется не только этот поток, но и все потоки того процесса, к которому он относится;

в стратегии с наличием потоков только на пользовательском уровне приложение не может воспользоваться преимуществом многопроцессорной системы, так как ядро закрепляет за каждым процессом только один

процессор. Поэтому несколько потоков одного и того же процесса не могут выполняться одновременно. В сущности, получается мультипрограммирование в рамках одного процесса;

при запуске одного потока ни один другой поток не будет запущен, пока первый добровольно не отдаст процессор. Внутри одного процесса нет прерываний по таймеру, в результате чего невозможно создать планировщик для поочередного выполнения потоков.

Рассмотрим теперь потоки на уровне ядра. В этом случае в области приложения система поддержки исполнения программ не нужна, нет необходимости и в таблицах потоков в каждом процессе. Вместо этого есть единая *таблица* потоков, отслеживающая все потоки в системе. Если потоку необходимо создать новый *поток* или завершить имеющийся, он выполняет *запрос* ядра, который создает или завершает *поток*, внося изменения в таблицу потоков.

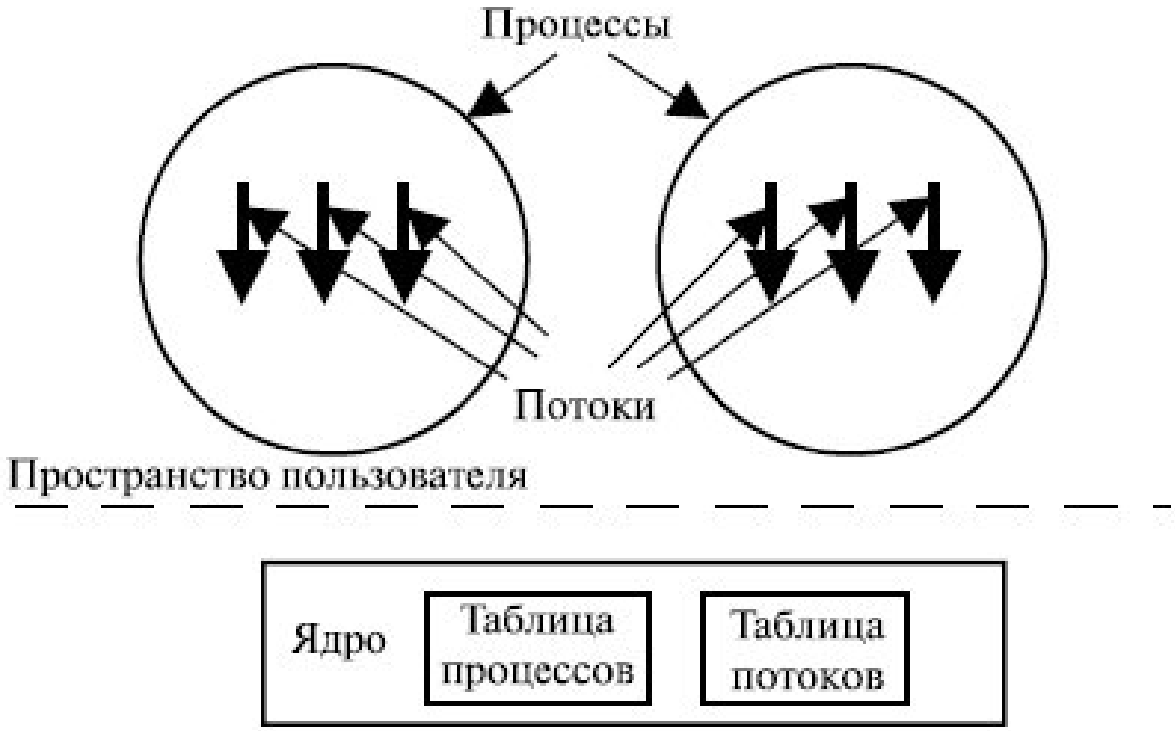


Рис. 5.8. Потоки в пространстве ядра

Любое *приложение* можно запрограммировать как многопоточное, при этом все потоки приложения поддерживаются в рамках единого *процесса. Ядро* поддерживается информацией контекста процесса как единого целого, а также контекстами каждого отдельного потока процесса. Планирование осуществляется ядром, исходя из состояния потоков. С помощью такого подхода удается избавиться от основных недостатков потоков пользовательского уровня.

Возможно планирование работы нескольких потоков одного и того же процесса на нескольких процессорах:

1. реализуется мультипрограммирование в режимах нескольких процессов (вообще – всех);
2. при блокировке одного из потоков процесса ядро может выбрать для выполнения другой поток этого же процесса;
3. процедуры ядра могут быть многопоточными.

Главный недостаток связан с необходимостью двукратного переключения режимов пользовательский – *ядро*, *ядро* – пользовательский для передачи одного потока к другому в рамках одного и того же процесса.

Планирование заданий, процессов и потоков

Основная цель планирования вычислительного процесса заключается в распределении времени процессора (нескольких процессоров) между выполняющимися заданиями пользователей таким образом, чтобы удовлетворять требованиям, предъявляемым пользователями к вычислительной системе. Такими требованиями могут быть, как это уже отмечалось, *пропускная способность*, время отклика, *загрузка* процессора и др.

Все виды планирования, используемые в современных ОС, в зависимости от временного масштаба, делятся на долгосрочное, среднесрочное, краткосрочное и планирование ввода-вывода. Рассматривая частоту работы планировщика, можно сказать, что долгосрочное планирование выполняется сравнительно редко, среднесрочное несколько чаще. Краткосрочный *планировщик*, называемый часто *диспетчер* (*dispatcher*), обычно работает, определяя, какой процесс или *поток* будет выполняться следующим. Ниже приведен перечень функций, выполняемых планировщиком каждого вида.

Вид планирования	Выполняемые функции
Долгосрочное	Решение о добавлении задания (процесса) в пул выполняемых в системе
Среднесрочное	Решение о добавлении процесса к числу процессов, полностью или частично размещенных в основной памяти
Краткосрочное	Решение о том, какой из доступных процессов (потоков) будет выполняться процессором
Планирование ввода-вывода	Решение о том, какой из запросов процессов (потоков) на операцию ввода-вывода будет выполняться свободным устройством ввода-вывода

Место планирования в графе состояний и переходов процессов показано на следующем слайде.

МЕСТО ПЛАНИРОВАНИЯ В ГРАФЕ ПРОЦЕССОВ

Рис. 5.9. Место планирования в графе процессов

В большинстве операционных систем универсального назначения планирование осуществляется динамически (on-line), т.е. решения

принимаются во время работы системы на основе анализа текущей ситуации, не используя никаких предложений о мультипрограммной смеси. Найденное оперативно решение в таких условиях редко бывает оптимальным.

Другой тип планирования – *статический* (предварительный), может быть использован только в специализированных системах с заданным набором задач (заранее определенным), например, в управляющих вычислительных системах или системах реального времени. В этом случае статический *планировщик* (или предварительный *планировщик*) принимает решение не во время работы системы, а заранее (off-line). Результатом его работы является расписание – *таблица*, в которой указано, какому процессу, когда и на какое время должен быть предоставлен *процессор*. При этом накладные *расходы* ОС на *исполнение* расписания значительно меньше, чем при динамическом планировании.

Краткосрочный *планировщик* (*диспетчер*) реализует найденное решение, т.е. переключает *процессор* с одного процесса (потока) на другой. Он вызывается при наступлении события, которое может приостановить текущий *процессор* или предоставить возможность прекратить выполнение данного процесса (потока) в пользу другого. Примерами этих событий могут быть:

- прерывание таймера;
- прерывание ввода-вывода;
- вызовы операционной системы;
- сигналы.

Среднесрочное планирование является частью системы свопинга. Обычно решение о загрузке процесса в *память* принимается в зависимости от степени многозадачности (например, OS *MFT*, OS MVT). Кроме того, в системе с отсутствием виртуальной памяти среднесрочное планирование тесно связано с вопросами управления памятью.

Диспетчеризация сводится к следующему:

- сохранение контекста текущего потока, который требуется сменить;
- загрузка контекста нового потока, выбранного в результате планирования;
- запуск нового потока на выполнение.

Поскольку *переключение контекстов* существенно влияет на *производительность* вычислительной системы, программные модули ОС выполняют эту операцию при поддержке аппаратных средств процессора.

В мультипрограммной системе *поток* (процесс, если *операционная система* работает только с процессами) может находиться в одном из трех основных состояний:

- выполнение – активное состояние потока, во время которого поток обладает всеми необходимыми ресурсами и непосредственно выполняется процессором;
- ожидание – пассивное состояние потока, находясь в котором, поток заблокирован по своим внутренним причинам (ждет осуществления некоторого события, например, завершения операции ввода-вывода, получения сообщения от другого потока или освобождения какого-либо необходимого ему ресурса);
- готовность – также пассивное состояние потока, но в этом случае поток заблокирован в связи с внешним по отношению к нему обстоятельством (имеет все требуемые ресурсы, готов выполняться, но процессор занят выполнением другого потока).

В течение своей жизни каждый *поток* переходит из одного состояния в другое в соответствии с алгоритмом планирования потоков, принятом в данной операционной системе.

В состоянии выполнения в однопроцессорной системе может находиться не более одного потока, а в остальных состояниях – несколько. Эти потоки образуют очереди, соответственно, ожидающих и готовых потоков. Очереди организуются путем объединения в списки описателей отдельных потоков. С самых общих позиций все множество алгоритмов планирования можно разделить на два класса: вытесняющие и не *вытесняющие* алгоритмы планирования.

Не вытесняющие (non-*preemptive*) алгоритмы основаны на том, что активному потоку позволяется выполняться, пока он сам, *по* своей инициативе, не отдает управление операционной системе, для того чтобы она выбрала из очереди готовый к выполнению *поток*.

Вытесняющие (*preemptive*) алгоритмы – это такие способы планирования потоков, в которых решение о переключении процессора с выполнения одного потока на выполнение другого потока принимается операционной системой, а не активной задачей.

В первом случае механизм планирования распределяется между операционной системой и прикладными программами. Во втором случае функции планирования потоков целиком сосредоточены в операционной системе.

Недостатком первого типа алгоритмов планирования является необходимость разработки такого приложения, которое будет "дружественным" *по* отношению к другим выполняемым одновременно с ним программам. Для этого в приложении должны быть предусмотрены частные передачи управления операционной системе. Крайним проявлением недружественности приложения является его зависание, которое приводит к общему краху системы. Потому *распределение функций* планирования между ОС и приложениями достаточно сложно в программистском отношении и

используется, как правило, в специализированных системах с фиксированным набором задач. В то же время существенным преимуществом невытесняющего планирования является более высокая скорость переключения потоков.

Однако почти во всех ОС (*UNIX*, *Windows* NT/2000/2003, *OS/2*, VAX/VMS и др.) реализованы вытесняющие алгоритмы планирования. В основе многих таких алгоритмов лежит концепция квантования. В соответствии с ней каждому потоку поочередно для выполнения предоставляется ограниченный непрерывный период процессорного времени – квант.

Смена активного потока происходит, если:

- поток завершается и покинул систему;
- произошла ошибка;
- поток перешел в состояние ожидания;
- исчерпан квант времени, отведенный данному потоку.

Поток, который исчерпал свой квант, переводится в состояние готовности и ожидает, когда ему будет предоставлен новый *квант процессорного времени*, а на выполнение в соответствии с определенным правилом выбирается новый *поток* из очереди готовых потоков.

Кванты, выделяемые потокам, могут быть равными или различными (типичное *значение* десятки – сотни мс). Например, первоначально каждому потоку назначается достаточно большой квант, а величина каждого следующего кванта уменьшается до некоторой заранее заданной величины. В таком случае преимущество получают короткие задачи, которые успевают выполняться в течение первого кванта (второго и т.д.), а длительные вычисления будут проводиться в фоновом режиме.

Некоторые потоки, получив квант времени, используют его не полностью, например, из-за необходимости выполнить ввод или *вывод* данных. В результате может возникнуть ситуация, когда потоки с интенсивным вводом-выводом используют только небольшую часть выделенного им процессорного времени. Можно исправить эту "несправедливость", изменив *алгоритм* планирования, например, так: создать две очереди потоков, *очередь* 1 – для потоков, которые пришли в состояние готовности в результате исчерпания кванта времени, и *очередь* 2 – для потоков, у которых завершилась операция ввода-вывода. При выборе потока для выполнения сначала просматривается вторая *очередь*, и если она пуста, квант выделяется потоку из первой очереди.

Отметим три замечания об алгоритмах, основанных на квантовании.

Первое. Переключение контактов потоков связано с потерями процессорного времени, которые не зависят от величины кванта, но зависят от частоты переключения. Поэтому чем больше квант, тем меньше суммарные *затраты* процессорного времени на переключение потоков.

Второе. С увеличением кванта может быть ухудшено *качество обслуживания* пользователей, связанное с ростом времени реакции системы.

Третье. В алгоритмах, основанных на квантовании, ОС не имеет никаких сведений о решаемых задачах (длинные или короткие, интенсивен "ввод-*вывод*" или нет, важно быстрое *исполнение* или нет и т.п.). Дифференциация обслуживания при квантовании базируется на "истории существования" потока в системе.

Важной концепцией, лежащей в основе многих вытесняющих алгоритмов планирования, является приоритетное обслуживание. Оно предполагает наличие у потоков некоторой изначально известной характеристики – приоритета, на основании которого определяется порядок выполнения потоков. Чем выше приоритет, тем выше привилегии потока, тем меньше времени *поток* находится в очередях. Приоритет задается числом (целым или дробным, положительным или отрицательным).

В большинстве ОС, поддерживающих потоки, *приоритет потока* связан с *приоритетом процесса*, в рамках которого выполняется *поток*. *Приоритет процесса* назначается операционной системой при его создании, его *значение* включается в описатель процесса и используется при назначении приоритета потоком этого процесса. При назначении приоритетов вновь созданному процессу ОС учитывается, является ли этот процесс системным или прикладным, каков статус пользователя, запустившего процесс (*администратор*, *пользователь*, часть и т.п.), было ли явное указание пользователя на присвоение процессу определенного уровня приоритета. *Поток* может быть инициирован не только *по* команде пользователя, но и в результате выполнения системного вызова другим потоком. В этом случае ОС учитывает значения параметров системного вызова.

Изменения приоритета могут происходить *по* инициативе самого потока, когда он обращается с соответствующим вызовом к ОС, или *по* инициативе пользователя, когда он выполняет соответствующую команду. Кроме этого, сама ОС может изменить приоритеты потоков в зависимости от ситуации, складывающейся в системе. В последнем случае приоритеты называются динамическими в отличие от неизменяемых, фиксированных приоритетов. Возможности пользователей влиять на приоритеты процессов и потоков ограничены ОС. Обычно это разрешается администраторам, и то в определенных пределах. В большинстве случаев ОС присваивает приоритеты потокам *по* умолчанию.

Существует две разновидности *приоритетного планирования*: с относительными и абсолютными приоритетами. В обоих случаях на выполнение выбирается *поток*, имеющий наивысший приоритет. Но *определение* момента смены активного потока решается *по*-разному. В системах с относительными приоритетами *активный поток* выполняется до тех пор, пока он сам не покинет *процессор*, перейдя в состояние ожидания (*по* вводу-выводу, например), или не завершится, или не произойдет ошибка. В

системах с абсолютными приоритетами выполнение активного потока прерывается еще и *по* причине появления потока, имеющего более высокий приоритет, чем у активного потока. В этом случае прерванный *поток* переходит в состояние готовности.

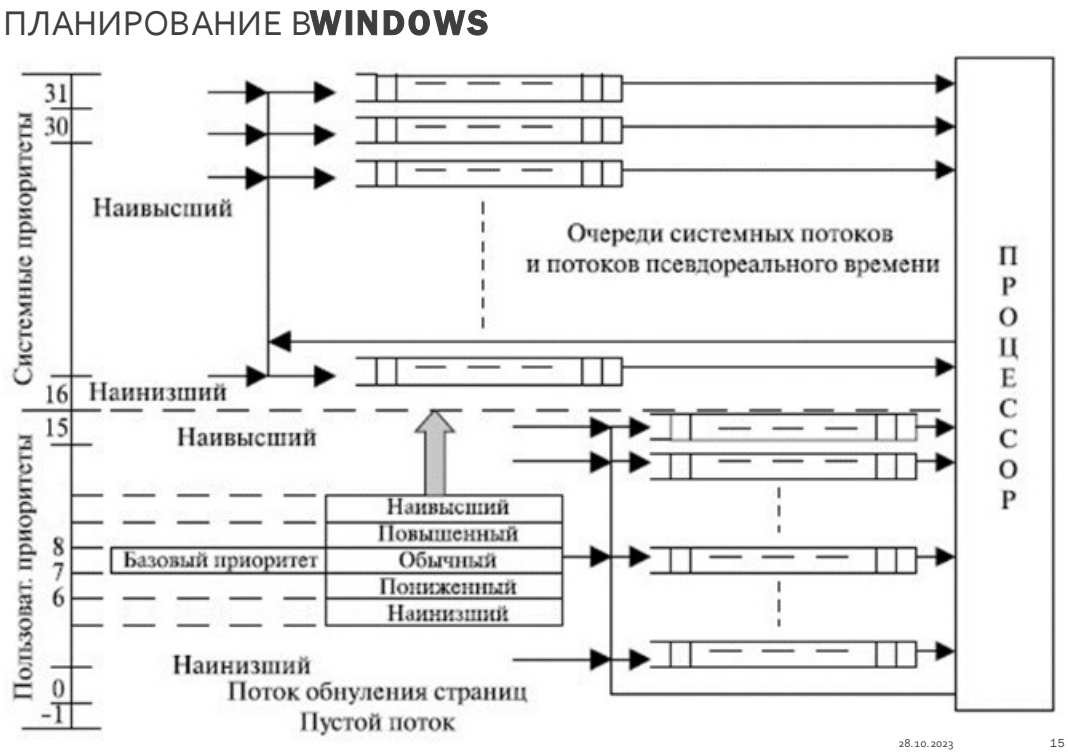


Рис. 5.10. Планирование в Windows

В качестве примера рассмотрим организацию приоритетного обслуживания в *Windows 2000*/2003/XP/Vista. Здесь приоритеты организованны в виде двух групп, или классов: реального времени и переменные. Каждая из групп состоит из 16 уровней приоритетов.

Потоки, требующие немедленного внимания, находятся в классе реального времени, который включает такие функции, как осуществление коммуникаций и задачи реального времени.

В целом, поскольку *W2K* использует вытесняющий *планировщик* с учетом *приоритетов, потоки* с приоритетами реального времени имеют преимущество *по* отношению к прочим потокам. В однопроцессорной системе, когда становится готовым к выполнению *поток* с более высоким приоритетом, чем выполняющийся в настоящий момент, текущий *поток* вытесняется и начинает выполняться *поток* с более высоким приоритетом.

Приоритеты из разных классов обрабатываются несколько *по*-разному. В классе приоритетов реального времени все потоки имеют фиксированный приоритет (от 16 до 31), который никогда не изменяется, и все активные потоки с определенным уровнем приоритета располагаются в круговой очереди

данного класса (tкв=20 мс для *W2K Professional*, 120 мс – для однопроцессорных серверов).

В классе переменных приоритетов *поток* начинает работу с базового приоритета процесса, который может принимать *значение* от 1 до 15. Каждый *поток*, связанный с процессом имеет, свой базовый приоритет, равный базовому приоритету процесса, или отличающийся от него не более чем на 2 уровня в большую или меньшую сторону. После активации потока его динамический приоритет может колебаться в определенных пределах – он не может упасть ниже наименьшего базового приоритета данного класса, т.е. 15 (для потоков с приоритетом 16 и выше никогда не делается никаких изменений приоритетов).

Когда же увеличивается *приоритет потока*? Во-первых, когда завершается операция ввода-вывода и освобождается ожидающий ее *поток*, его приоритет увеличивается, чтобы дать шанс этому потоку запуститься быстрее и снова запустить операцию ввода-вывода. Суть в том, чтобы поддержать занятость устройств ввода-вывода. Величина, на которую увеличивается приоритет, зависит от устройства ввода-вывода. Как правило, это 1 – для диска, 2 – для последовательной линии, 6 – для клавиатуры и 8 – для звуковой карты.

Во-вторых, если *поток* ждал семафора, мьютекса или другого события, то когда он отпускается, к его приоритету добавляется 2 единицы, если это *поток* переднего плана (т.е. управляет окном, к которому направляется ввод с клавиатуры), и одна *единица* – в противном случае. Таким образом, *интерактивный процесс* получает преимущество перед большим количеством других процессов. Наконец, если *поток* графического интерфейса пользователя просыпается, потому что стал доступен оконный ввод, он также получает прибавку к приоритету.

Эти увеличения приоритета не вечны. Они незамедлительно вступают в силу, но если *поток* использует полностью свой следующий квант, он теряет один *пункт* приоритета. Если он использует еще квант, то перемещается еще на уровень ниже и т.д. вплоть до своего базового уровня.

Последняя модификация алгоритма планирования *W2K* заключается в том, что когда окно становится окном переднего плана, все его потоки получают более длительные кванты времени (величина прибавки хранится в системном реестре).

Поток, созданный в системе, может находиться в одном из 6 состояний в соответствии с графом, приведенным на слайде

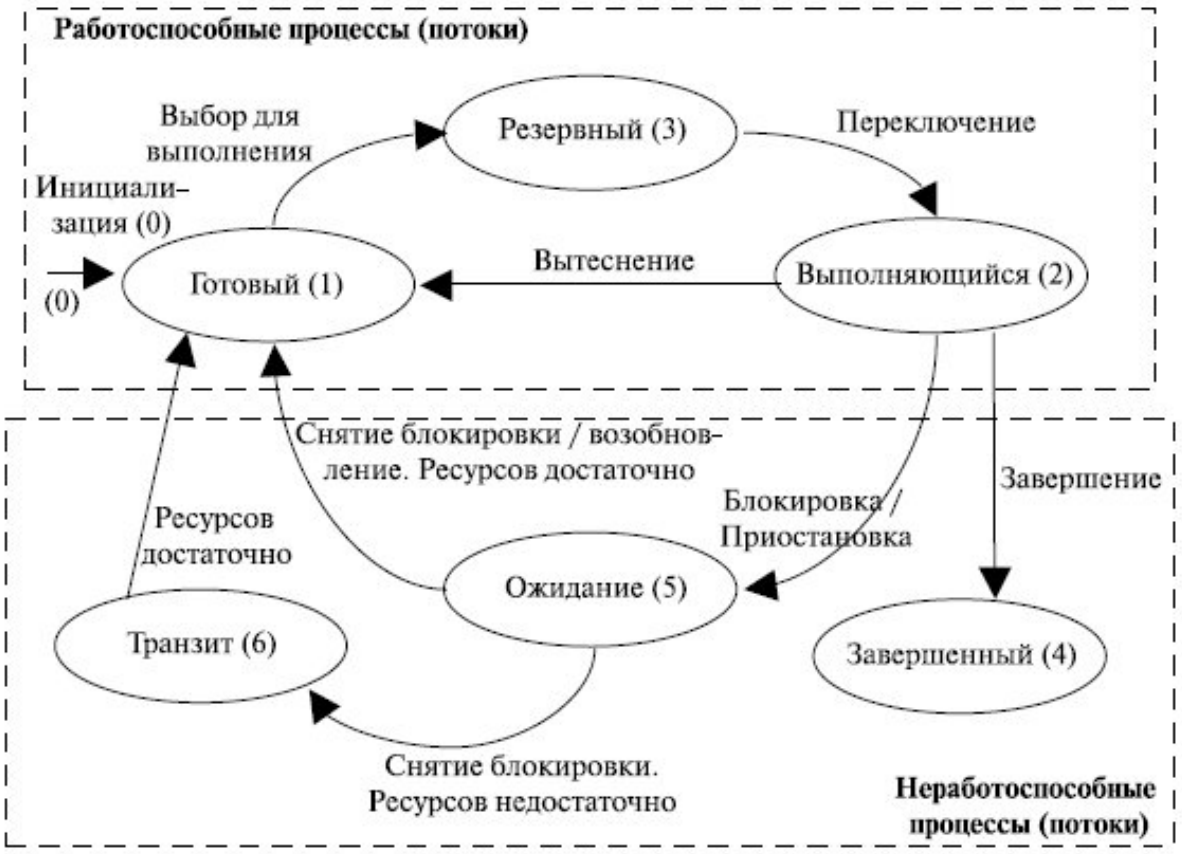


Рис. 5.11. Состояния потоков в Windows

Взаимодействие и синхронизация процессов и потоков

В мультипрограммных однопроцессорных системах процессы чередуются, обеспечивая эффективное выполнение программ. В многопроцессорных системах возможно не только *чередование*, но и перекрытие процессов. Обе эти технологии, которые можно рассматривать как примеры параллельных вычислений, порождают одинаковые проблемы. Выполнение процессов и потоков в мультипрограммной среде всегда имеет асинхронный характер – невозможно предсказать относительную скорость выполнения процессов. Момент прерывания потоков, время нахождения их в очередях к разделяемым ресурсам, порядок выбора потоков для выполнения – все эти события являются результатом стечения многих обстоятельств и являются случайными, это справедливо как *по* отношению к потокам одного процесса, выполняющим общий программный код, так и *по* отношению к потокам разных процессов, каждый из которых выполняет собственную программу.

Способы взаимодействия процессов (потоков) можно классифицировать *по* степени осведомленности одного процесса о существовании другого [10].

1. Процессы *не осведомлены* о наличии друг друга (например, процессы разных заданий одного или различных пользователей). Это независимые процессы, не предназначенные для совместной работы. Хотя эти процессы и не работают совместно, ОС должна решать вопросы конкурентного использования ресурсов. Например, два независимых

приложения могут затребовать доступ к одному и тому же диску или принтеру. ОС должна регулировать такие обращения.

2. Процессы *косвенно осведомлены* о наличии друг друга (например, процессы одного задания). Эти процессы не обязательно должны быть осведомлены о наличии друг друга с точностью до идентификатора процесса, однако они разделяют доступ к некоторому объекту, например, буферу ввода-вывода, файлу или БД. Такие процессы демонстрируют сотрудничество при разделении общего объекта.

3. Процессы *непосредственно осведомлены* о наличии друг друга (например, процессы, работающие последовательно или поочередно в рамках одного задания). Такие процессы способны общаться один с другим с использованием идентификаторов процессов и изначально созданы для совместной работы. Эти процессы также демонстрируют сотрудничество при работе.

При необходимости использовать один и тот же *ресурс* параллельные процессы вступают в *конфликт* (конкурируют) друг с другом. Каждый из процессов не подозревает о наличии остальных и не повергается никакому воздействию с их стороны. Отсюда следует, что каждый процесс не должен изменять состояние любого ресурса, с которым он работает. Примерами таких ресурсов могут быть устройства ввода-вывода, *память*, *процессорное время*, часы.

Между конкурирующими процессами не происходит никакого обмена информацией. Однако выполнение одного процесса может повлиять на поведение конкурирующего процесса. Это может, например, выразиться в замедлении работы одного процесса, если ОС выделит *ресурс* другому процессу, поскольку первый процесс будет ждать завершения работы с этим ресурсом. В предельном случае *блокированный процесс* может никогда не получить *доступ* к нужному ресурсу и, следовательно, никогда не сможет завершиться.

В случае конкурирующих процессов (потоков) возможно возникновение трех проблем. Первая из них – необходимость взаимных исключений (*mutual exclusion*). Предположим, что два или большее количество процессов требуют *доступ* к одному неразделяемому ресурсу, как например принтер.

О таком ресурсе будем говорить как о *критическом ресурсе*, а о части программы, которая его использует, – как о критическом разделе (*critical section*) программы. Крайне важно, чтобы в критической ситуации в любой момент могла находиться только одна *программа*. Например, во время печати файла требуется, чтобы отдельный процесс имел полный *контроль* над принтером, иначе на бумаге можно получить *чередование* строк двух файлов.

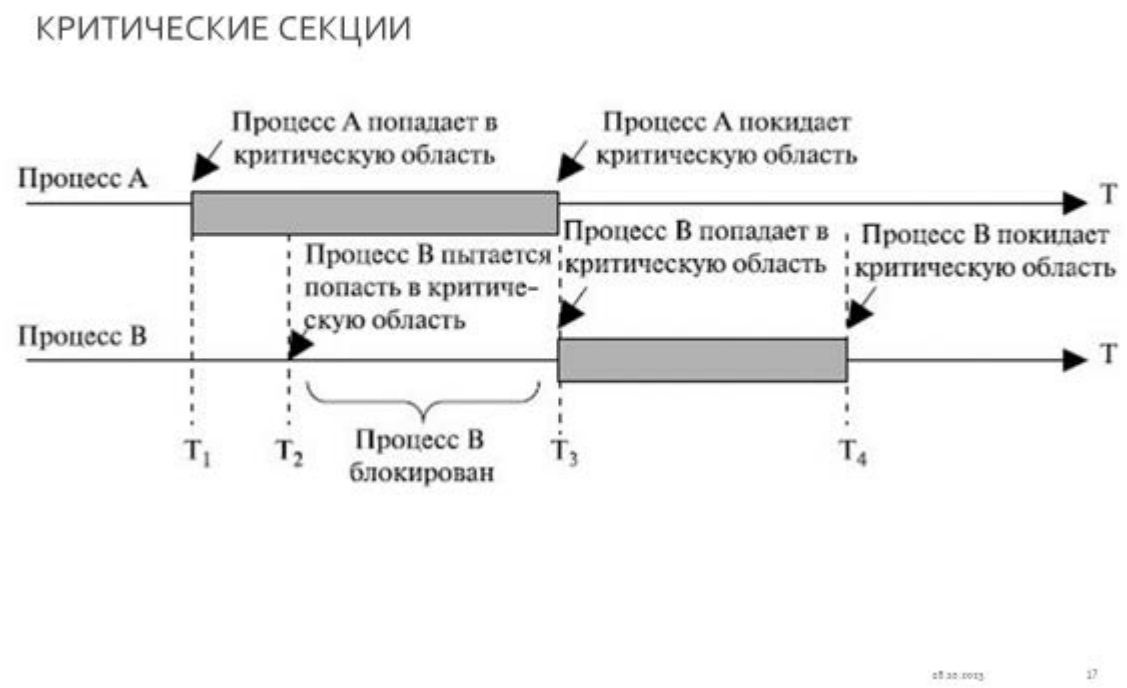


Рис. 5.12. Критические секции

Осуществление взаимных исключений создает две дополнительные проблемы. Одна из них – *взаимоблокировки* (*deadlock*) или тупики. Рассмотрим, например, два процесса – P1 и P2, и два ресурса – R1 и R2. Предположим, что каждому процессу для выполнения части своих функций требуется *доступ* к общим ресурсам. Тогда возможно возникновение следующей ситуации: ОС выделяет *ресурс* R1 процессу P2, а *ресурс* R2 – процессу P1. В результате каждый процесс ожидает получения одного из двух ресурсов. При этом ни один из них не освобождает уже имеющийся *ресурс*, ожидая получения второго ресурса для выполнения функций, требующих наличие двух ресурсов. В результате процессы оказываются взаимно заблокированными.

Очень удобно моделировать *условия возникновения тупиков*, используя *направленные графы* (предложено Holt, 1972). Графы имеют 2 вида узлов: процессы-кружочки и ресурсы-квадратики. *Ребро*, направленное от квадрата (ресурса) к кружку (процессу), означает, что *ресурс* был запрошен, получен и используется. В нашем примере это будет изображено так, как показано на следующем слайде. а)

ТУПИКОВАЯ СИТУАЦИЯ

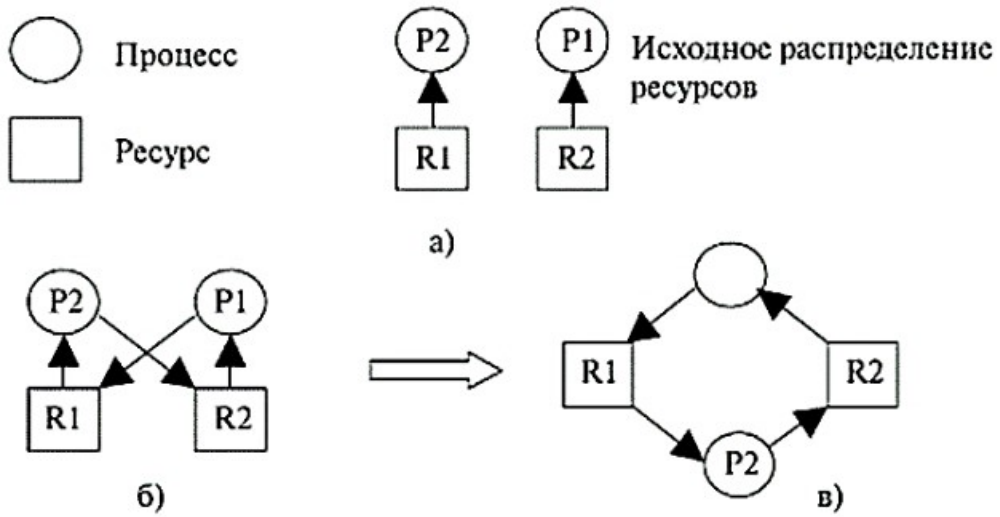


Рис. 5.13. Тупиковая ситуация

Ребро, направленное от процесса (кружка) к ресурсу (квадрату), означает, что процесс в данный момент заблокирован и находится в состоянии ожидания доступа к этому ресурсу. В нашем примере *граф* надо достроить, как показано на слайде б) или в). Цикл в графе означает наличие взаимной блокировки процессов.

Существует еще одна проблема у конкурирующих процессов – голодание. Предположим, что имеется 3 процесса (P1, P2, P3), каждому из которых периодически требуется *доступ* к ресурсам R. Представим ситуацию, в которой P1 обладает ресурсом, а P2 и P3 приостановлены в ожидании освобождения ресурса R. После выхода P1 из критического раздела *доступ* к ресурсу будет получен одним из процессов P2 или P3.

Пусть ОС предоставила *доступ* к ресурсу процессу P3. Пока он работает с ресурсом, *доступ* к ресурсу вновь требуется процессу P1. В результате *по* освобождении ресурса R процессом P3 может оказаться, что ОС вновь предоставит *доступ* к ресурсу процессу P1. Тем временем процессу P3 вновь требуется *доступ* к ресурсу R. Таким образом, теоретически возможна ситуация, в которой процесс P2 никогда не получит *доступ* к требуемому ему ресурсу несмотря на то, что никакой взаимной блокировки в данном случае нет.