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:

- 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 ниже.

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 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 рис. 5.6.

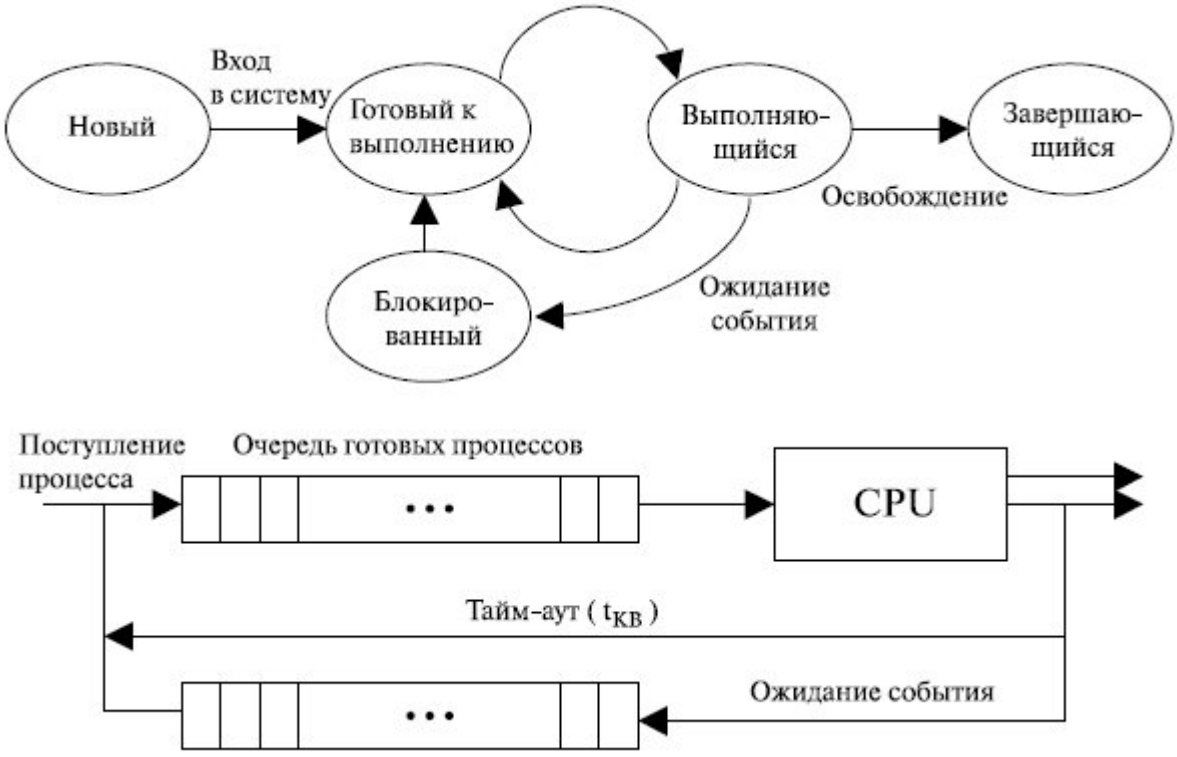

Рис. 5.6. Состояния процесса

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:

- 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).

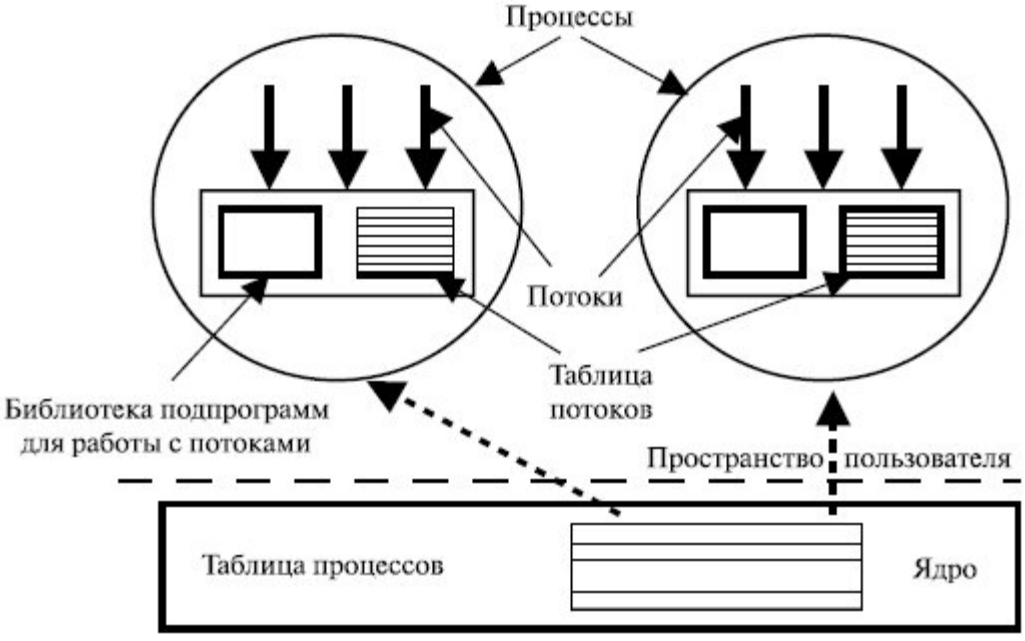

Рис. 5.7. Потоки в пространстве пользователя

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:

1. 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;
2. имеется возможность использования различных алгоритмов планирования потоков в различных приложениях (процессах) с учетом их специфики;
3. 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).

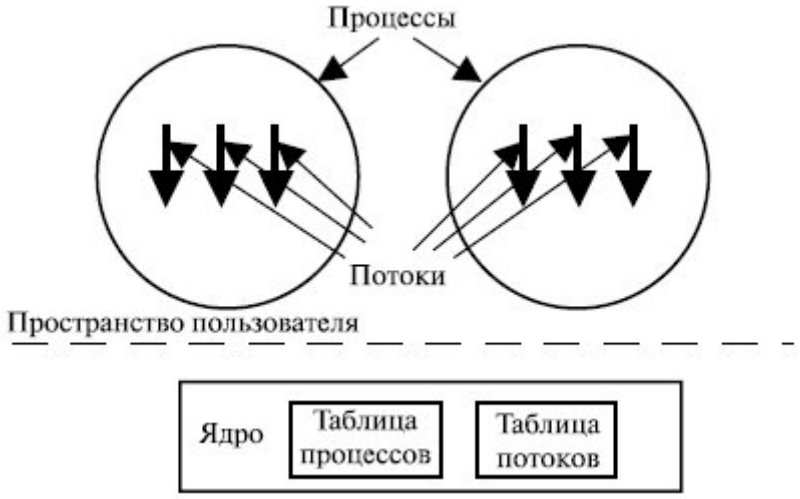

Рис. 5.8. Потоки в пространстве ядра

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.