Процессы и потоки. Межпроцессное и межпоточное взаимодействие. Часть 1

Лекция

Ревизия: 0.2

Последняя модификация: Vlad Kovtun

Дата последней модификации: 15.10.2014 22:29:00

История изменений

09.02.2010 – Версия 0.1. Первичный документ. Ковтун В.Ю.

19.07.2014 – Версия 0.2. Замена рисунков. Ковтун В.Ю.

Содержание

Лекция 3. Процессы и потоки. Межпроцессное и межпоточное взаимодействие

Вопросы

1. Процессы.

2. Потоки.

Процессы

Современные компьютеры обладают CPU, которые могут параллельно выполнять несколько потоков команд. Например, одновременно с запущенной пользователем программой может выполняться чтение с диска и вывод текста на экран монитора или на принтер. В многозадачной системе CPU переключается между программами, предоставляя каждой от десятков до сотен миллисекунд. При этом в каждый конкретный момент времени CPU занят только одной программой, но за секунду он успевает поработать с несколькими программами, создавая у пользователей иллюзию параллельной работы со всеми программами. Иногда в этом контексте говорят о **псевдопараллелизме**, в отличие от настоящего параллелизма в **многопроцессорных** системах (в которых установлено два и более CPU, разделяющих между собой общую физическую память). Следить за работой параллельно идущих процессов достаточно трудно, поэтому со временем разработчики ОС разработали концептуальную модель последовательных процессов, упрощающую эту работу.

Модель процесса

В этой модели все функционирующее на компьютере ПО, иногда включая собственно ОС, организовано в виде набора **последовательных процессов**, или, для краткости, просто **процессов**. **Процессом** является выполняемая программа, включая: текущие значения счетчика команд, регистров и переменных. С позиций данной **абстрактной модели**, у каждого процесса есть собственный виртуальный CPU. На самом деле, подразумевается, реальный CPU переключается с процесса на процесс, но для лучшего понимания системы значительно проще рассматривать набор процессов, идущих параллельно (псевдопараллельно), чем пытаться представить себе CPU, переключающийся от программы к программе. Как уже знаем из предыдущих лекций, это переключение и называется **многозадачностью или мультипрограммированием**. На Рис. 1(а) представлена схема single-CPU компьютера, работающего с четырьмя программами. На Рис. 1(б) представлены четыре процесса, каждый со своей управляющей логикой (то есть логическим счетчиком команд), идущие независимо друг от друга.

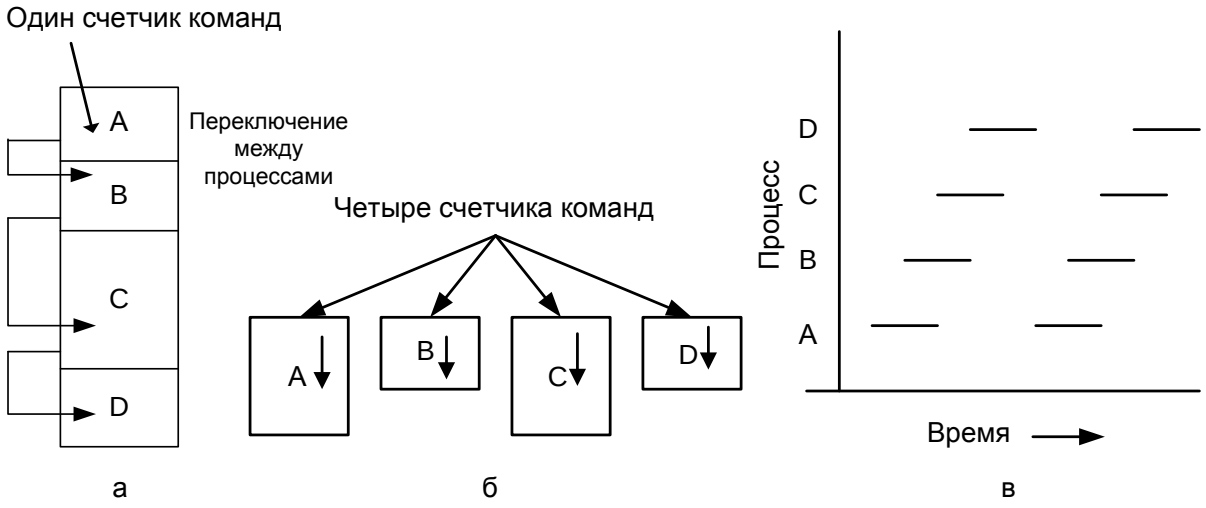

Рис. 1. Четыре программы в многозадачном режиме (а); принципиальная модель четырех независимых последовательных процессов (б); в каждый момент времени активна только одна программа (в)

Разумеется, на самом деле существует только один физический счетчик команд, в который загружается логический счетчик команд текущего процесса. Когда время,

отведенное текущему процессу, заканчивается, физический счетчик команд сохраняется в логическом счетчике команд процесса в памяти. На Рис. 1(в) видно, что за достаточно большой промежуток времени изменилось состояние всех четырех процессов, но в каждый конкретный момент в действительности работает только один процесс.

Процесс – это род действия. У него есть программа, входные, выходные данные и состояние. Один процессор может совместно использоваться несколькими процессами в соответствии с неким алгоритмом планирования, который используется для определения, когда остановить один процесс и обслужить другой.

Создание процесса

ОС необходим способ, позволяющий удостовериться в наличии всех необходимых процессов. В простейших системах, а также системах, разработанных для выполнения одного-единственного приложения (например, контроллер микроволновой печи), можно реализовать такую ситуацию, в которой все процессы, которые когда-либо могут понадобиться, присутствуют в системе при ее загрузке. В универсальных системах необходим способ создания и прерывания процессов по мере необходимости. Рассмотрим некоторые из возможных способов решения этой задачи. Ниже перечислены четыре основных события, приводящие к созданию процессов.

1. Инициализация системы.

2. Выполнение системного запроса на создание процесса уже работающим процессом.

3. Запрос пользователя на создание процесса.

4. Инициирование пакетного задания.

Обычно при загрузке ОС создаются несколько процессов. Некоторые из них являются высокоприоритетными процессами, то есть обеспечивающими взаимодействие с пользователем и выполняющими заданную работу. Остальные процессы являются фоновыми, они не связаны с конкретными пользователями, но выполняют особые функции. Например, один фоновый процесс может быть предназначен для обработки приходящей на компьютер почты, активизируясь только по мере появления писем. Другой фоновый процесс может обрабатывать запросы к web-страницам, расположенным на компьютере, и активизироваться для обслуживания полученного запроса. Фоновые процессы, связанные с электронной почтой, web-страницами, новостями, выводом на печать и т. п., называются **демонами** (в UNIX системах) и сервисами (в Windows системах). В больших системах насчитываются десятки демонов. В UNIX для вывода списка запущенных процессов используется программа ps. В Windows можно воспользоваться Диспетчером задач.

Процессы могут создаваться не только в момент загрузки системы, но и позже. Например, новый процесс (или несколько) может быть создан по просьбе текущего процесса. Создание новых процессов особенно полезно в тех случаях, когда выполняемую задачу проще всего сформировать как набор связанных, тем не менее независимых взаимодействующих процессов. Если необходимо организовать выборку большого количества данных из сети для дальнейшей обработки, удобно создать один процесс для выборки данных и размещения их в совместно используемом буфере, в то время как второй процесс будет считывать данные из буфера и обрабатывать их. Эта схема даже ускорит обработку данных, если каждый процесс запустить на отдельном CPU в случае multi-CPU системы.

В интерактивных системах пользователь может запустить программу, набрав на клавиатуре команду или дважды щелкнув на значке программы. В обоих случаях результатом будет создание нового процесса и запуск в нем программы. Когда на UNIX работает X Windows, новый процесс получает то окно, в котором был запущен. В Microsoft Windows процесс не имеет собственного окна при запуске, но он может (и должен) создать одно или несколько окон. В обеих системах пользователь может одновременно открыть несколько окон, каждому из которых соответствует свой процесс. Пользователь может переключаться между окнами с помощью мыши и взаимодействовать с процессом, например, вводя данные по мере необходимости.

Последнее событие, приводящее к созданию нового процесса, связано с системами пакетной обработки на больших компьютерах. Пользователи посылают пакетное задание (возможно, с использованием удаленного доступа), а ОС создает новый процесс и запускает следующее задание из очереди в тот момент, 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. В роли текущего процесса может выступать процесс, запущенный пользователем, системный процесс, инициированный клавиатурой или мышью, а также процесс, управляющий пакетами. В любом случае этот процесс всего лишь выполняет системный запрос и создает новый процесс. Системный запрос заставляет ОС создать новый процесс, а также прямо или косвенно содержит информацию о программе, которую нужно запустить в этом процессе.

В UNIX существует только один системный запрос, направленный на создание процесса: `fork`. Этот запрос создает дубликат вызываемого процесса. После выполнения запроса `fork` двум процессам - родительскому и дочернему - соответствуют одинаковые образы памяти, строки окружения и открытые файлы. Обычно, дочерний процесс выполняет системный вызов `execve` для изменения образа памяти и запуска новой программы.

В Windows вызов всего одной функции `CreateProcess` управляет и созданием процесса и запуском нужной в ней программы. После создания нового процесса, родительский и дочерний процессы имеют собственные различные адресные пространства. В тоже время, созданный процесс может использовать одинаковые ресурсы с родительским процессом, например открытые файлы.

Завершение процесса

После того как процесс создан, он начинает выполнять свою работу. Но ничто не длится вечно, даже процесс — рано или поздно он завершится, чаще всего благодаря одному из следующих событий:

1. Обычный выход (преднамеренно).

2. Выход по ошибке (преднамеренно).

3. Выход по неисправимой ошибке (непреднамеренно).

4. Уничтожение другим процессом (непреднамеренно).

В основном процессы завершаются по мере выполнения своей работы. После окончания компиляции программы, компилятор выполняет системный запрос, чтобы сообщить ОС об окончании работы. В UNIX этот системный запрос — `exit`, а в Windows — `ExitProcess`. Программы, рассчитанные на работу с экраном, также поддерживают преднамеренное завершение. В текстовых редакторах, браузерах и других программах такого типа обычно есть кнопка или пункт меню, щелкнув на котором можно удалить все временные файлы, открытые процессом, и затем завершить процесс.

Второй причиной завершения процесса может стать неустранимая ошибка.

Интерактивные процессы, рассчитанные на работу с экраном, обычно не завершают работу при получении неверных параметров, вместо этого выводя на экран диалоговое окно и прося пользователя ввести правильные параметры.

Третьей причиной завершения процесса является ошибка, вызванная самим процессом, чаще всего связанная с ошибкой в программе. В качестве примера можно привести выполнение недопустимой команды, обращение к несуществующей области памяти и деление на ноль. В некоторых системах (например, в UNIX) процесс может информировать ОС о том, что он сам обработает некоторые ошибки, и в этом случае процессу посылается сигнал (процесс прерывается, а не завершается) при появлении ошибки.

Четвертой причиной завершения процесса может служить выполнение другим процессом системного запроса на уничтожение процесса. В UNIX такой системный запрос — `kill`, а соответствующая функция Win32 — `TerminateProcess`. В обоих случаях «киллер» должен обладать соответствующими полномочиями по отношению к «убиваемому» процессу. В некоторых системах при завершении процесса (преднамеренно или нет) все процессы, созданные процессом, также завершаются. Впрочем, это не относится ни к UNIX, ни к Windows.

Иерархия процессов

В некоторых системах родительский и дочерний процессы остаются связанными между собой определенным образом. Дочерний процесс также может, в свою очередь, создавать процессы, формируя иерархию процессов. Следует отметить, что в отличие

от животного мира у процесса может быть лишь один родитель и сколько угодно «детей».

В UNIX процесс, все его «дети» и дальнейшие потомки образуют группу процессов. Сигнал, посылаемый пользователем с клавиатуры, доставляется всем членам группы, взаимодействующим с клавиатурой в данный момент (обычно это все активные процессы, созданные в текущем окне). Каждый из процессов может перехватить сигнал, игнорировать его или выполнить другое действие, предусмотренное по умолчанию.

Рассмотрим в качестве еще одного примера иерархии процессов инициализацию UNIX при запуске. В образе загрузки присутствует специальный процесс `init`. При запуске этот процесс считывает файл, в котором находится информация о количестве терминалов. Затем процесс разветвляется таким образом, чтобы каждому терминалу соответствовал один процесс. Процессы ждут, пока какой-нибудь пользователь не войдет в систему. Если пароль правильный, процесс входа в систему запускает оболочку для обработки команд пользователя, которые, в свою очередь, могут запускать процессы. Таким образом, все процессы в системе принадлежат к единому дереву, начинающемуся с процесса `init`. Напротив, в Windows не существует понятия иерархии процессов, и все процессы равноправны. Единственное, в чем проявляется что-то вроде иерархии процессов — создание процесса, в котором родительский процесс получает специальный маркер (так называемый дескриптор), позволяющий контролировать дочерний процесс. Но маркер можно передать другому процессу, нарушая иерархию. В UNIX это невозможно.

Состояния процессов

Несмотря на то, что процесс является независимым объектом, со своим счетчиком команд и внутренним состоянием, существует необходимость взаимодействия с другими процессами. Например, выходные данные одного процесса могут служить входными данными для другого процесса. В команде оболочки

```
cat chapterl chapter2 chapter3 | grep tree
```

первый процесс, исполняющий файл `cat`, объединяет и выводит три файла. Второй процесс, исполняющий файл `grep`, отбирает все строки, содержащие слово «tree». В зависимости от относительных скоростей процессов (скорости зависят от относительной сложности программ и CPU времени, предоставляемого каждому процессу), может получиться, что `grep` уже готов к запуску, но входных данных для этого процесса еще нет. В этом случае процесс блокируется до поступления входных данных.

Процесс блокируется, поскольку с точки зрения логики он не может продолжать свою работу (обычно это связано с отсутствием входных данных, ожидаемых процессом). Также возможна ситуация, когда процесс, готовый и способный работать, останавливается, поскольку ОС решила предоставить на время CPU другому процессу. Эти ситуации являются принципиально разными. В первом случае приостановка выполнения является внутренней проблемой (поскольку невозможно обработать командную строку пользователя до того, как она была введена). Во втором случае проблема является технической (нехватка CPU для каждого процесса). На Рис. 2 представлена диаграмма состояний, показывающая три возможных состояния процесса:

1. Работающий (в этот конкретный момент использующий CPU).

2. Готовый к работе (процесс временно приостановлен, чтобы позволить выполняться другому процессу).

3. Заблокированный (процесс не может быть запущен прежде, чем произойдет некое внешнее событие).

Ошибка! Объект не может быть создан из кодов полей редактирования.

Рис. 2. Процесс может находиться в рабочем, готовом и заблокированном состоянии. Стрелками показаны возможные переходы между состояниями

С точки зрения логики первые два состояния одинаковы. В обоих случаях процесс может быть запущен, только во втором случае недоступен CPU. Третье состояние отличается тем, что запустить процесс невозможно, независимо от загруженности CPU.

Как показано на Рис. 2, между этими тремя состояниями возможны четыре перехода. Переход 1 происходит, когда процесс обнаруживает, что продолжение работы невозможно. В некоторых системах процесс должен выполнить системный запрос, например `block` или `pause`, чтобы оказаться в заблокированном состоянии. В других системах, как в UNIX, процесс автоматически блокируется, если при считывании из канала или специального файла (предположим, терминала) входные данные не были обнаружены.

Переходы 2 и 3 вызываются частью ОС, называемой **планировщиком процессов**, так что сами процессы даже не знают о существовании этих переходов. Переход 2 происходит, если планировщик решил, что пора предоставить CPU следующему процессу. Переход 3 происходит, когда все остальные процессы уже исчерпали свое CPU время, и CPU снова возвращается к первому процессу. Вопрос планирования (когда следует запустить очередной процесс и на какое время) сам по себе достаточно важен. Было разработано множество алгоритмов с целью сбалансировать требования эффективности для системы в целом и для каждого процесса в отдельности.

Переход 4 происходит с появлением внешнего события, ожидавшегося процессом (например, прибытие входных данных). Если в этот момент не запущен какой-либо другой процесс, то срабатывает переход 3, и процесс запускается. В противном случае процессу придется некоторое время находиться в состоянии готовности, пока не освободится CPU.

Модель процессов **упрощает** представление о внутреннем поведении системы. Некоторые процессы запускают программы, выполняющие команды, введенные с клавиатуры пользователем. Другие процессы являются частью системы и обрабатывают такие задачи, как выполнение запросов файловой службы, управление запуском диска или магнитного накопителя. В случае дискового прерывания система останавливает текущий процесс и запускает дисковый процесс, который был заблокирован в ожидании этого прерывания. Вместо прерываний мы можем представлять себе дисковые процессы, процессы пользователя, терминала и т. п., блокирующиеся на время ожидания событий. Когда событие произошло (информация прочитана с диска или клавиатуры), блокировка снимается, и процесс может быть запущен.

Рассмотренный подход описывается моделью, представленной на Рис. 3. Нижний уровень ОС — это планировщик, на верхних уровнях расположено множество процессов. Вся обработка прерываний и детали, связанные с остановкой и запуском процессов, спрятаны в том, что мы назвали планировщиком, являющимся, по сути, совсем небольшой программой. Вся остальная часть ОС удобно структурирована в виде набора процессов. Очень немногие существующие системы структурированы столь удобно.

Рис. 3. Нижний уровень ОС отвечает за прерывания и планирование. Выше - расположены последовательные процессы

Реализация процессов

Для реализации модели процессов ОС содержит таблицу (массив структур), называемую **таблицей процессов**, с одним элементом для каждого процесса. (Некоторые авторы называют эти элементы блоками управления процессом.) **Элемент таблицы** содержит информацию о состоянии процесса, счетчике команд, указателе стека, распределении памяти, состоянии открытых файлов, об использовании и распределении ресурсов, а также всю остальную информацию, которую необходимо сохранять при переключении в состояние готовности или блокировки для последующего запуска — как если бы процесс не останавливался.

В Таблица 1 представлены **некоторые наиболее важные поля** типичной системы. Поля в первой колонке относятся к управлению процессом. Остальные колонки описывают управление памятью и файлами. Необходимо отметить, что от конкретной системы очень сильно зависит, какие именно поля будут в таблице процессов, но Таблица 1 дает общее представление о необходимой информации.

Теперь, после знакомства с таблицей процессов, можно сказать еще несколько слов о том, как поддерживается **иллюзия нескольких последовательных процессов** на машине с одним CPU и несколькими устройствами ввода-вывода. С каждым классом устройств ввода-вывода (гибкий диск, жесткий диск, таймер, терминал) связана область памяти (обычно расположенная в нижних адресах), называемая вектором прерываний. Вектор прерываний содержит адрес процедуры обработки прерываний. Представьте, что в момент прерывания диска работал пользовательский процесс 3. Содержимое счетчика команд процесса, слово состояния программы и, возможно, один или несколько регистров записываются в (текущий) стек аппаратными средствами прерывания. Затем происходит переход по адресу, указанному в векторе прерывания диска. Вот и все, что делают аппаратные средства прерывания. С этого момента вся остальная обработка прерывания производится программным обеспечением, например процедурой обработки прерываний.

Все прерывания начинаются с сохранения регистров, часто в блоке управления текущим процессом в таблице процессов. Затем информация, помещенная в стек прерыванием, удаляется, и указатель стека переставляется на временный стек, используемый программой обработки процесса. Такие действия, как сохранение регистров и установка указателя стека, невозможно даже выразить на языке высокого уровня (например, на C). Поэтому они выполняются небольшой программой на ассемблере, обычно одинаковой для всех прерываний, поскольку процедура сохранения регистров не зависит от причины возникновения прерывания.

Таблица 1. Содержимое таблицы процессов (контекст процесса)

Управление процессом	Управление памятью	Управление вводом-выводом (файлами)
Регистры	Указатель на сегмент команд	Корневой каталог
Счётчик команд	Указатель на сегмент данных	Рабочий (текущий) каталог
Слово состояния программы	Указатель на сегмент стека	Дескрипторы файла
Указатель стека		Идентификатор пользователя
Состояние процесса		Идентификатор группы
Приоритет		
Параметры планирования		
Идентификатор процесса		
Родительский процесс		
Группа процесса		
Сигналы		
Время старта процесса		
Использованное время CPU		
Время CPU дочернего процесса		

Время следующего аварийного сигнала		

По завершении своей работы эта программа вызывает процедуру на языке С, которая выполняет все остальные действия, связанные с конкретным прерыванием. (Мы предполагаем, что ОС написана на С, что является стандартным решением для всех существующих ОС.) Когда процедура завершает свою работу (в результате чего, возможно, некоторые процессы переходят в состояние готовности), вызывается планировщик для выбора следующего процесса. После этого управление возвращается к программе на ассемблере, загружающей регистры и карту памяти для текущего процесса и запускающей его. Управление прерыванием и работа планировщика представлены в Таблица 2. Следует отметить, что отдельные детали могут несколько варьироваться от системы к системе.

Таблица 2. Схема обработки прерывания нижним уровнем ОС

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

Потоки

В обычных ОС каждому процессу соответствует адресное пространство и одиночный **управляющий поток**. Фактически это и определяет процесс. Тем не менее, часто встречаются ситуации, в которых предпочтительно иметь несколько квазипараллельных управляющих потоков в одном адресном пространстве, как если бы они были различными процессами (однако разделяющим одно адресное пространство).

Модель потока

Модель процесса, которую рассматривали ранее, базируется на двух независимых концепциях: группировании ресурсов и выполнении программы. Иногда полезно их разделять, и тут появляется понятие потока.

С одной стороны, процесс можно рассматривать как способ объединения родственных ресурсов в одну группу. У процесса есть адресное пространство, содержащее объектный код программы (машинные команды) и данные, а также другие ресурсы. **Ресурсами** являются открытые файлы, дочерние процессы, необработанные аварийные сообщения, обработчики сигналов, учетная информация и многое другое. Гораздо проще управлять ресурсами, объединив их в форме процесса.

С другой стороны, процесс можно рассматривать как **поток исполняемых команд** или просто поток. **У потока есть**:

- Значение счетчика команд, отслеживающий порядок выполнения действий.
- Значения всех регистров CPU.
- Свой стек, содержащий протокол выполнения процесса, где на каждую процедуру, вызванную, но еще не вернувшуюся, отведен отдельный фрейм.

Хотя поток должен исполняться внутри процесса, следует различать концепции потока и процесса. **Процессы** используются для группирования ресурсов, а **потоки** являются объектами, поочередно исполняющимися на CPU.

Концепция потоков добавляет к модели процесса возможность одновременного выполнения в одной и той же среде процесса нескольких программ, в достаточной степени независимых. Несколько потоков, работающих параллельно в одном процессе, аналогичны нескольким процессам, идущим параллельно на одном компьютере. В первом случае потоки разделяют адресное пространство, открытые файлы и другие

ресурсы. Во втором случае процессы совместно пользуются физической памятью, дисками, принтерами и другими ресурсами. Потоки обладают некоторыми свойствами процессов, поэтому их иногда называют **упрощенными процессами**. Термин **многопоточность** также используется для описания использования нескольких потоков в одном процессе.

На Рис. 4(а) представлены три обычных процесса, у каждого из которых есть собственное адресное пространство и одиночный поток управления. На Рис. 4(б) представлен один процесс с тремя потоками управления. В обоих случаях мы имеем три потока, но на Рис. 4(а) каждый из них имеет собственное адресное пространство, а на Рис. 4(б) потоки разделяют единое адресное пространство.

При запуске многопоточного процесса в системе с одним CPU потоки работают поочередно. Пример работы процессов в многозадачном режиме мы уже видели на Рис. 1. Иллюзия параллельной работы нескольких различных последовательных процессов создается путем постоянного переключения системы между процессами. Многопоточность реализуется примерно так же. CPU быстро переключается между потоками, создавая впечатление параллельной работы потоков, хотя и на не столь быстром CPU. В случае трех ограниченных производительностью CPU потоков в одном процессе все потоки будут работать параллельно, и каждому потоку будет соответствовать виртуальный CPU с быстродействием, равным одной трети быстродействия реального CPU.

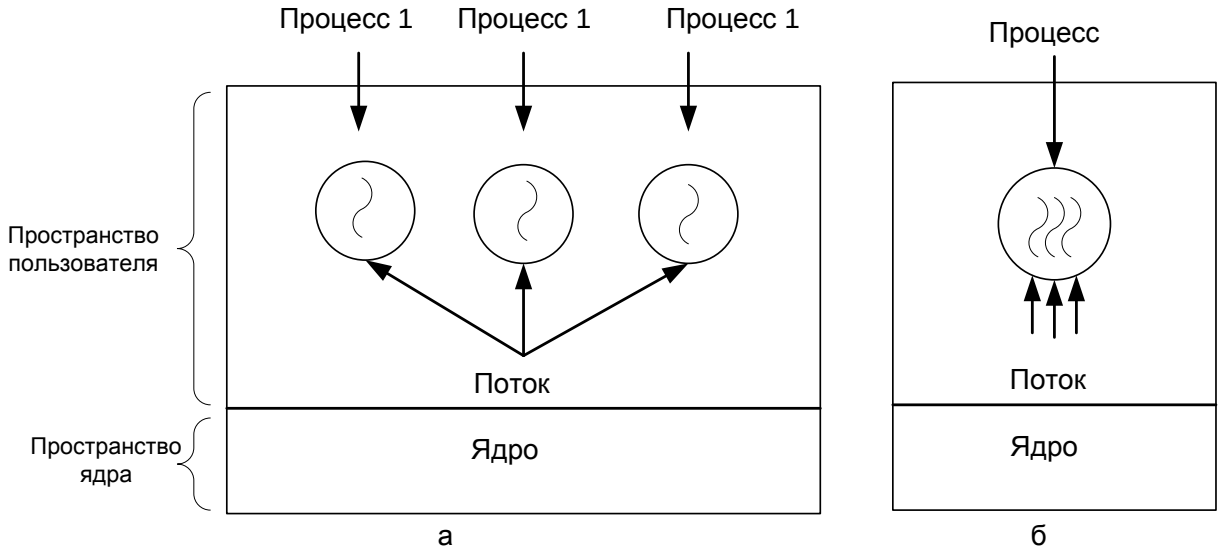

Рис. 4. Три процесса с одиночными потоками управления (а); один процесс с тремя потоками управления (б)

Различные потоки в одном процессе не так независимы, как различные процессы. У всех потоков одно и то же адресное пространство, что означает совместное использование глобальных переменных. Поскольку любой поток имеет доступ к любому адресу ячейки памяти в адресном пространстве процесса, один поток может считывать, записывать или даже стирать информацию из стека другого потока. Защиты не существует, поскольку:

1. Это невозможно.

2. Это ненужно.

В отличие от различных процессов, которые могут быть инициированы различными пользователями и преследовать несовместимые цели, один процесс всегда запущен одним пользователем, и потоки созданы таким образом, чтобы работать совместно, не мешая друг другу. Как показано в таблице 3, потоки разделяют не только адресное пространство, но и открытые файлы, дочерние процессы, сигналы и т. п. Таким образом, ситуацию на Рис. 4(а) следует использовать в случае абсолютно несвязанных процессов, тогда как схема на Рис. 4(б) будет уместна, когда потоки выполняют совместно одну работу.

Таблица 3. В первой колонке перечислены элементы, совместно используемые всеми потоками процесса, а во второй — элементы, индивидуальные для каждого потока

Элементы процесса	Элементы потока
Адресное пространство	Счетчик команд
Глобальные переменные	Регистры
Открытые файлы	Стек
Дочерние процессы	Состояние
Необработанные аварийные сигналы	TLS (Thread Local Storage) – локальное хранилище потока в ОС Windows
Сигналы и их обработчики	
Информация об использовании ресурсов	

Первая колонка содержит элементы, являющиеся свойствами процесса, а не потока. Например, если один поток открывает файл, этот файл тут же становится видимым для остальных потоков, и они могут считывать информацию и записывать ее в файл. Это логично, поскольку процесс, а не поток является единицей управления ресурсами. Если бы у каждого потока было собственное адресное пространство, открытые файлы, аварийные сигналы, требующие обработки и т. д., это были бы отдельные процессы. Концепция потоков состоит в возможности совместного использования набора ресурсов несколькими потоками для выполнения некой задачи в тесном взаимодействии.

Как и любой обычный процесс (то есть процесс с одним потоком), поток может находиться в одном из нескольких состояний:

- рабочем,
- заблокированном,
- готовности,
- завершенном.

Действующий поток взаимодействует с CPU. Блокированный поток ожидает некоторого события, которое его разблокирует. Например, при выполнении системного запроса чтения с клавиатуры поток блокируется, пока не поступит сигнал с клавиатуры. Поток может быть разблокирован каким-либо внешним событием или другим потоком. Поток в состоянии готовности будет запущен, как только до него дойдет очередь. Переходы между состояниями потоков такие же, как на Рис. 2.

Важно понимать, что у каждого потока свой собственный стек, как показано на Рис. 5. Стек каждого потока содержит по одному фрейму для каждой процедуры, вызванной, но еще не вернувшей управления. Во фрейме находятся локальные переменные процедуры и адрес возврата. Например, если процедура X вызывает процедуру Y и она, в свою очередь, вызывает процедуру Z, то во время работы процедуры Z в стеке будут находиться фреймы для всех трех процедур. Каждый поток может вызывать различные процедуры и, соответственно, иметь различный протокол выполнения процесса. Именно поэтому каждому потоку необходим собственный стек.

Ошибка! Объект не может быть создан из кодов полей редактирования.

Рис. 5. Демонстрация персонально стека у каждого из потоков

В многопоточном режиме процессы, как правило, запускаются с одним потоком. Этот поток может создавать новые потоки, вызывая библиотечную процедуру, например `thread_create` в UNIX и `CreateThread` Windows. Параметром обычно является имя процедуры, которую необходимо запустить для создания нового потока. Указание какой-либо информации, касающейся адресного пространства нового потока, не является необходимым (или даже возможным), поскольку новый поток создается в адресном пространстве существующего потока. Иногда возникает иерархия потоков с отношениями типа «родительский—дочерний поток», но чаще всего иерархия отсутствует и все потоки считаются равнозначными. Независимо от иерархических отношений, создающему потоку чаще всего возвращается идентификатор потока, который дает имя новому потоку.

Выполнив задачу, поток может прекратить работу, вызвав библиотечную процедуру, скажем, `thread_exit`. После этого поток исчезает и уже не рассматривается планировщиком. В некоторых потоковых системах один поток может ждать прекращения работы другого (определенного) потока. Для этого вызывается

процедура, например `thread_wait`. Процедура блокирует вызывающий процедуру поток, пока другой поток (определенный) не прекратит работу. В этом отношении создание и завершение потоков очень похожи на создание и завершение процессов с практически такими же параметрами.

Еще одно распространенное обращение потока — `thread_yield` позволяет потоку добровольно «уступать свою очередь» другому потоку. Это важный момент, поскольку в случае потоков не существует прерывания по таймеру, позволяющего установить режим разделения времени, как это было в случае процессов. Потокам необходимо быть **вежливыми** и время от времени самим уступать CPU другим потокам. Существуют и процедуры, позволяющие одному потоку подождать, пока другой завершит какое-либо действие, оповестить о том, что он закончил какое-либо действие и т. п.

Несмотря на то, что потоки часто бывают полезными, они существенно усложняют программную модель. Представьте себе системный вызов `fork` в UNIX. Если у родительского процесса было много потоков, должно ли это свойство распространяться на дочерний процесс? Если нет, то процесс может неправильно функционировать, поскольку все потоки могут оказаться необходимы.

Но что произойдет, если поток **родительского процесса будет блокирован** вызовом `read` с клавиатуры, а у дочернего процесса столько же потоков, сколько у родительского? Будут ли теперь блокированы два потока — один из родительского процесса, другой из дочернего? И если с клавиатуры поступит строка, получат ли оба потока ее копию? Или только один — тогда какой? Эта же проблема возникает при работе с открытыми сетевыми соединениями.

Другой класс проблем связан с тем, что потоки совместно используют большое количество структур данных. Что произойдет, если один поток закроет файл в то время, когда другой считывает из него данные? Представьте себе, что одному потоку стало недостаточно памяти, и он просит выделить дополнительную память. На полпути происходит переключение потоков, и теперь новый поток также замечает, что ему не хватает памяти, и просит выделить дополнительную память. В этой ситуации память может быть выделена дважды. Все эти проблемы можно решить, но для создания корректно работающих многопоточных программ необходима тщательная и всесторонне обдуманная разработка.

Использование потоков

Почему же потоки так необходимы? Основной причиной является выполнение большинством приложений существенного числа действий, некоторые из них могут время от времени блокироваться. Схему программы можно существенно упростить, если разбить приложение на несколько последовательных потоков, запущенных в квазипараллельном режиме.

С этим рассуждением мы уже сталкивались — оно являлось аргументом в пользу существования процессов, не так ли? Мы можем рассуждать на языке параллельных процессов вместо прерываний, таймеров и переключателей контекста. В случае потоков придется добавить еще один элемент: **возможность совместного использования параллельными объектами адресного пространства и всех содержащихся в нем данных**. Для определенных приложений эта возможность является существенной, и в таких случаях схема параллельных процессов (с разными адресными пространствами) не подходит.

Еще одним аргументом в пользу потоков является легкость их создания и уничтожения (поскольку с потоком не связаны никакие ресурсы). В большинстве систем на создание потока уходит примерно в 100 раз меньше времени, чем на создание процесса. Это свойство особенно полезно, если необходимо динамическое и быстрое изменение числа потоков.

Третьим аргументом является производительность. Концепция потоков не дает увеличения производительности, если все они ограничены возможностями CPU. Но когда имеется одновременная потребность в выполнении большого объема вычислений и операций ввода-вывода, наличие потоков позволяет совмещать эти виды деятельности во времени, тем самым увеличивая общую скорость работы приложения.

И наконец, концепция потоков полезна в системах с несколькими процессорами, где возможен настоящий параллелизм.

Необходимость потоков проще продемонстрировать на конкретных примерах. Возьмем в качестве первого примера текстовый редактор. Большинство текстовых редакторов выводят текст на экран монитора в том виде, в котором он будет напечатан. В частности, разрывы строк и страниц находятся на своих местах, и пользователь может при необходимости их откорректировать (например, удалить неполные строки вверху и внизу страницы, неприемлемые с эстетической точки зрения).

Представьте себе, что пользователь пишет книгу. С точки зрения автора проще всего хранить книгу в одном файле, чтобы легче было искать отдельные разделы, выполнять глобальную замену и т. п. С другой стороны, можно хранить каждую главу в отдельном файле. Но было бы крайне неудобно хранить каждый раздел и параграф в своем файле — в случае глобальных изменений пришлось бы редактировать сотни файлов. Например, если предполагаемый стандарт xxx был утвержден только перед отправкой книги в печать, придется заменять «Черновой стандарт xxx» на «Стандарт xxx» в последнюю минуту. Эта операция делается одной командой в случае одного файла и, напротив, займет очень много времени, если придется редактировать каждый из 300 файлов, на которые разбита книга.

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

В этом случае помогут потоки. Пусть текстовый редактор написан в виде двухпоточной программы. Один поток взаимодействует с пользователем, а второй переформатирует документ в фоновом режиме. Как только предложение на первой странице было удалено, интерактивный поток дает команду фоновому потоку переформатировать весь документ. В то время как первый поток продолжает отслеживать и выполнять команды с клавиатуры или мыши — предположим, прокручивает первую страницу, — второй поток быстро переформатирует книгу. Немного везения — и форматирование будет закончено раньше, чем пользователь захочет перейти к 600 странице, и тогда команда будет выполнена мгновенно.

Раз уж мы об этом задумались, почему бы не добавить третий поток? Большинство текстовых редакторов автоматически сохраняет редактируемый текст раз в несколько минут, чтобы пользователь не лишился плодов работы целого дня в случае аварийного завершения программы, отказа системы или перебоев с питанием. Этим может заниматься третий поток, не отвлекая два оставшихся, Рис. 6.

Ошибка! Объект не может быть создан из кодов полей редактирования.

Рис. 6. Приложение с тремя потоками

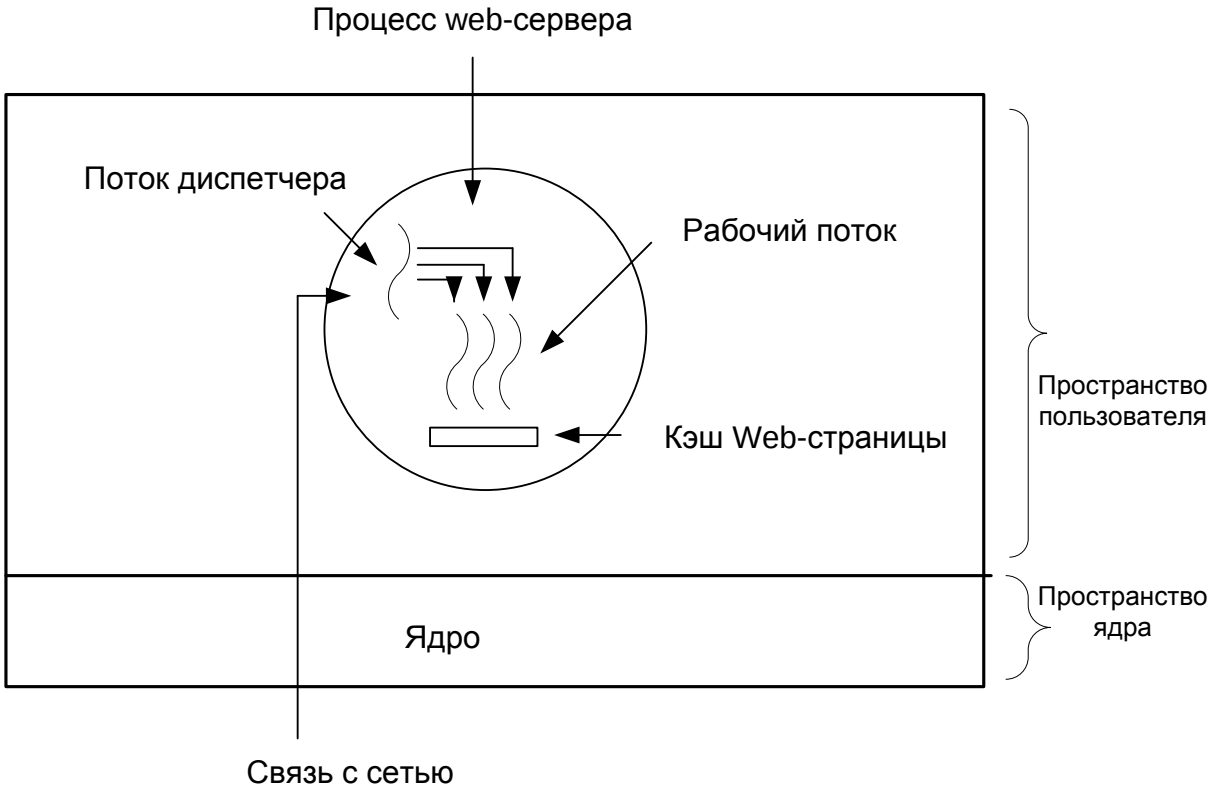

Рис. 7. Многопоточный web-сервер

Ошибка! Объект не может быть создан из кодов полей редактирования.

Рис. 8. Набросок программы для реализации многопоточного web-сервера: (а) – поток диспетчера запросов, (б) – рабочий поток обработки запросов

Теперь должно быть ясно, какие преимущества привносят потоки. **Они дают возможность сохранить модель последовательных Процессов, выполняющих блокирующие системные запросы (например, для ввода-вывода с диска), и тем не менее добиться параллелизма. Системные запросы с блокировкой упрощают программирование, а параллелизм увеличивает производительность.**

Однопоточный сервер сохраняет простоту программирования, связанную с наличием блокирующих системных запросов, но уступает в производительности. Модель конечного автомата существенно повышает производительность при помощи параллелизма, но использует системные запросы без блокировки, что усложняет программирование. Эти модели представлены в Таблица 4.

Таблица 4. Три способа конструирования сервера

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

Реализация потоков в пространстве пользователя

Есть два основных способа реализации пакета потоков: в пространстве пользователя и ядре. Выбор между ними остается спорным вопросом, и возможна смешанная реализация. Мы рассмотрим оба способа, а также их преимущества и недостатки.

Первый метод состоит в размещении пакета потоков целиком в пространстве пользователя. При этом ядро о потоках ничего не знает и управляет обычными, однопоточными процессами. Наиболее очевидное преимущество этой модели состоит в том, что пакет потоков на уровне пользователя можно реализовать даже в ОС, не поддерживающей потоки. Все ОС когда-то относились к этой категории, а некоторые относятся до сих пор.

Подобные реализации имеют в своей основе одинаковую общую схему, представленную на Рис. 9(а). Потоки работают поверх системы поддержки исполнения программ, которая является набором процедур, управляющих потоками. С четырьмя из них мы уже знакомы: `thread_create`, `thread_exit`, `thread_wait` и `thread_yield`, но обычно их больше.

Ошибка! Объект не может быть создан из кодов полей редактирования.

Рис. 9. Пакет потоков в пространстве пользователя (а); пакет потоков, управляемый ядром (б)

Если управление потоками происходит в пространстве пользователя, каждому процессу необходима собственная **таблица потоков** для отслеживания потоков в процессе. Эта таблица аналогична **таблице процессов**. Разницей является, что она отслеживает лишь характеристики потоков, такие как счетчик команд, указатель вершины стека, регистры, состояние и т. п. Когда поток переходит в состояние готовности или блокировки, вся информация, необходимая для повторного запуска, хранится в таблице потоков подобному тому, как в ядре хранится информация о процессах в таблице процессов.

Когда поток, ожидая окончания действия другого потока в том же процессе, делает нечто, что может привести к локальной блокировке, он вызывает процедуру системы поддержки исполнения программ. Процедура проверяет необходимость блокирования потока. В этом случае процедура сохраняет регистры потока в таблице потоков, ищет в таблице поток, готовый к запуску, и загружает его сохраненные значения в регистры машины. Как только указатель стека и счетчик команд переключены, работа нового потока возобновляется автоматически. Если у CPU есть команда, позволяющая за одну инструкцию сохранить все регистры, и еще одна, чтобы загрузить их все заново, переключение потоков может быть выполнено с помощью очень небольшого количества инструкций. Такое переключение потоков, по крайней мере, на порядок быстрее, чем переключения в режим ядра, и является серьезным аргументом в пользу управления потоками в пространстве пользователя.

Но существует одно серьезное отличие потоков от процессов. В тот момент, когда поток завершает на время свою работу, например, когда он вызывает процедуру `thread_yield`, программа `thread_yield` может сама сохранить информацию о потоке в таблице потоков. Более того, она может после этого вызвать планировщик потоков для выбора следующего потока. **Процедура, сохраняющая информацию о потоке, и планировщик являются локальными процедурами, и их вызов существенно более эффективен, чем вызов ядра.** Не требуются прерывание, переключение контекста, сохранение кэша и т. п., что существенно ускоряет переключение потоков.

Потоки, реализованные на уровне пользователя, имеют и другие преимущества. Они **позволяют каждому процессу иметь собственный алгоритм планирования**. Для некоторых приложений, например приложений с потоком «сборки мусора», оказывается удобным не задумываться о том, что поток может остановиться в неподходящий момент. Эти приложения также лучше масштабируются, поскольку потоки ядра неизменно занимают некоторое пространство в таблице и стековое пространство в ядре, что может стать проблемой в случае большого числа потоков.

Несмотря на более высокую производительность, с реализацией потоков на уровне пользователя связаны и некоторые **серьезные проблемы**:

- Первой из них является **проблема реализации блокирующих системных запросов**. Представьте, что поток начинает считывание с клавиатуры до того, как была нажата хотя бы одна клавиша. Было бы неприемлемо позволить потоку выполнить системный запрос, поскольку это остановило бы все потоки. Одной из основных целей использования потоков было предоставление возможности каждому потоку использовать блокирующие запросы, но так, чтобы один

блокированный поток не мешал остальным. Не очень понятно, как достичь этой цели, если использовать блокирующие системные запросы.

Можно сделать все системные запросы не блокирующими (как, например, запрос на чтение с клавиатуры `read`, который возвращал бы 0 байт в случае отсутствия данных), но это потребует неприемлемых изменений ОС. Ведь одним из основных аргументов в пользу потоков на уровне пользователя была возможность работы в существующих ОС. К тому же изменение семантики запроса `read` повлекло бы за собой изменения во многих пользовательских программах.

Оказалось, что существует альтернатива, если имеется возможность узнавать заранее, последует ли за запросом блокировка. В некоторых версиях UNIX есть системный запрос `select`, позволяющий узнать о наличии или отсутствии блокировки у последующего запроса `read`. При наличии такого системного запроса библиотечную процедуру `read` можно заменить новой, сначала выполняющей запрос `select`, а потом запрос `read`, если за ним не последует блокировки. Если блокировка должна произойти, то запрос не выполняется и запускается другой поток. В следующий момент, когда система поддержки исполнения программ получит управление, она может проверить еще раз, последует ли за запросом `read` блокировка. Такой подход требует переписывания части исходного кода библиотеки системных запросов, неэффективен и не отличается изяществом, но выбор не так уж велик. Код, который помещается вокруг системного запроса для проверки на блокировку, называется **чехлом** (jacket) или **упаковкой** (wrapper).

- **Проблема - ошибка из-за отсутствия страницы**. На данный момент нам важно, что компьютер можно настроить так, чтобы не вся программа находилась в основной памяти. Если программа производит вызов или переход к той инструкции, которой нет в памяти, возникает ошибка из-за отсутствия страницы, и ОС берет недостающую часть программы с диска. Именно эта ситуация называется ошибкой из-за отсутствия страницы. Процесс блокируется, пока необходимая инструкция не будет найдена и считана. Если поток приводит к ошибке из-за отсутствия страницы, ядро, не знающее о существовании потоков, блокирует весь процесс целиком, до завершения операции чтения с диска, несмотря на наличие остальных нормально функционирующих потоков.
- **Проблема потоков - при запуске одного потока ни один другой поток не будет запущен, пока первый поток добровольно не отдаст процессор**. Внутри одного процесса нет прерываний по таймеру, в результате чего невозможно создать планировщик для поочередного выполнения потоков. Планировщик ничего не сможет сделать, пока поток не окажется в системе поддержки исполнения программ по собственному желанию. **В Windows единицей переключения задач является поток, в связи с этим, в Windows, такая проблема отсутствует.**

Одним из решений этой проблемы может стать ежесекундное прерывание, передающее управление системе поддержки исполнения программ, но этот способ достаточно груб и неудобен. Периодические прерывания по таймеру с более высокой частотой не всегда возможны, а если и возможны, то издержки все равно будут существенными. Более того, возможно, что потоку необходимы свои прерывания по таймеру, которые будут конфликтовать с прерываниями системы поддержки исполнения программ.

- Еще один, и, возможно, наиболее серьезный аргумент против использования потоков на уровне пользователя состоит в том, что программисты хотят использовать потоки именно в тех приложениях, **в которых потоки часто блокируются**, например в многопоточном web-сервере. Эти потоки все время посылают системные запросы. И ядру, перехватившему управление, чтобы выполнить системный запрос, не составит труда заодно переключить потоки, если один из них блокирован. При этом исключается необходимость постоянных обращений к системе с запросом `select` для проверки наличия или отсутствия блокировки у последующего запроса `read`. А для приложений, которые полностью ограничены возможностями CPU и редко блокируются, потоки вообще не нужны. Вряд ли кто-либо станет всерьез применять потоки для вычисления первых n простых чисел или игры в шахматы.

Реализация потоков в ядре

Рассмотрим ситуацию, в которой ядро знает о существовании потоков и управляет ими. В этом случае система поддержки исполнения программ не нужна, как показано на Рис. 9(б). Нет необходимости и в наличии таблицы потоков. В каждом процессе, вместо

этого есть единая таблица потоков, отслеживающая все потоки системы. Если потоку необходимо создать новый поток или завершить имеющийся, он выполняет запрос ядра, который создает или завершает поток, внося изменения в таблицу потоков.

Таблица потоков, находящаяся в ядре, содержит регистры, состояние и другую информацию о каждом потоке. Информация та же, что и в случае управления потоками на уровне пользователя, только теперь она располагается не в пространстве пользователя (внутри системы поддержки исполнения программ), а в ядре. Эта информация является подмножеством информации, которую традиционное ядро хранит о каждом из своих однопоточных процессов (то есть подмножеством состояния процесса). Дополнительно ядро содержит обычную таблицу процессов, отслеживающую все процессы системы.

Все запросы, которые могут блокировать поток, реализуются как системные запросы, что **требует значительно больших временных затрат**, чем вызов процедуры системы поддержки исполнения программ. Когда поток блокируется, ядро по желанию запускает другой поток из этого же процесса (если есть поток в состоянии готовности) либо поток из другого процесса. При управлении потоками на уровне пользователя система поддержки исполнения программ запускает потоки из одного процесса, пока ядро не передает CPU другому процессу (или пока не кончаются потоки, находящиеся в состоянии готовности).

Поскольку создание и завершение потоков в ядре требует относительно больших расходов, некоторые системы используют повторное использование потоков. После завершения поток помечается как нефункционирующий, но в остальном его структура данных, хранящаяся в ядре, не затрагивается. Позже, когда нужно создать новый поток, ре-активируется остановленный (закешированный на уровне ядра) поток, что позволяет сэкономить на некоторых накладных расходах. При управлении потоками на уровне пользователя повторное использование потоков тоже возможно, но поскольку накладных расходов, связанных с управлением потоками, в этом случае существенно меньше, то и смысла в этом меньше.

Управление потоками в ядре не требует новых не блокирующих системных запросов. Более того, если один поток вызвал ошибку из-за отсутствия страницы, ядро легко может проверить, есть ли в этом процессе потоки в состоянии готовности, и запустить один из них, пока требуемая страница считывается с диска. Основным недостатком управления потоками в ядре является существенная цена системных запросов, поэтому постоянные операции с потоками (создание, завершение и т. п.) приведут к увеличению накладных расходов.

Смешанная реализация

С целью совмещения преимуществ реализации потоков на уровне ядра и на уровне пользователя были опробованы многие способы смешанной реализации. Один из методов заключается в использовании управления ядром и последующем мультиплексировании потоков на уровне пользователя, как показано на Рис. 10.

Ошибка! Объект не может быть создан из кодов полей редактирования.

Рис. 10. Мультиплексирование потоков пользователя в потоках ядра

В такой модели ядро знает только о потоках своего уровня и управляет ими. Некоторые из этих потоков могут содержать по нескольку потоков пользовательского уровня, мультиплексированных поверх них. Потоки пользовательского уровня создаются, завершаются и управляются так же, как потоки уровня пользователя в процессе, запущенном в не поддерживающей многопоточность системе. Предполагается, что у каждого потока ядра есть набор потоков на уровне пользователя, которые используют его по очереди.

Активация планировщика

Многие исследователи старались совместить преимущества реализации потоков на уровне ядра (**простота реализации**) и реализации потоков на уровне пользователя (**высокая производительность**). Ниже рассмотрим один из таких подходов, который называется **активацией планировщика**.

Целью активации планировщика является имитация функциональности потоков ядра, но с большей производительностью и гибкостью, свойственной потокам уровня

пользователя. В частности, пользовательские потоки не должны выполнять специальные системные запросы без блокировки или заранее должны проверять, не вызовет ли запрос блокировку. Тем не менее, когда поток блокируется системным запросом или ошибкой из-за отсутствия страницы, должна оставаться возможность запустить другой поток этого же процесса (если такой есть и находится в состоянии готовности).

Увеличение эффективности достигается **за счет уменьшения количества ненужных переходов между пространством пользователя и ядром**. Например, если поток блокирован в ожидании действий другого потока, совершенно не обязательно обращаться к ядру, что позволяет избежать накладных расходов по переходу «пользователь—ядро». Система поддержки исполнения программ, работающая в пространстве пользователя, может блокировать синхронизирующий поток и самостоятельно выбрать другой.

При использовании активации планировщика ядро назначает каждому процессу некоторое количество виртуальных CPU и позволяет системе поддержки исполнения программ (в пространстве пользователя) **распределять потоки по CPU**. Этот метод можно использовать и в multi-CPU системе, заменяя виртуальные CPU реальными. Исходное число виртуальных CPU, соответствующих одному процессу, равно единице, но процесс может запросить больше CPU и позже вернуть их. Ядро также может забрать виртуальный CPU у одного процесса и отдать другому, более нуждающемуся в нем в данный момент.

В основе механизма работы этой схемы лежит следующее утверждение. **Если ядро знает, что поток блокирован (например, если он выполнил блокирующий системный запрос или вызвал ошибку из-за отсутствия страницы), ядро оповещает об этом систему поддержки исполнения программ процесса, пересылая через стек в качестве параметров номер потока в запросе и описание случившегося.**

Недостатком метода активации планировщика является существенная зависимость от обратных вызовов, концепция, нарушающая свойственную любой многоуровневой системе структуру. Обычно уровень `n+1` может вызывать процедуры уровня `n`, но не наоборот. Обратные вызовы противоречат этому фундаментальному принципу.

Всплывающие потоки

Потоки часто используются в распределенных системах. Важным примером может служить обработка входящих сообщений, например запросов на обслуживание. Традиционный подход заключается в наличии процесса или потока, который блокируется по системному запросу `recieve`, ожидая входящего сообщения. Когда сообщение прибывает, оно принимается и обрабатывается.

Возможен и принципиально другой подход, при котором по прибытии сообщения система создает новый поток для его обработки. Такой поток называется **всплывающим**, его схема проиллюстрирована на Рис. 11. Основным преимуществом всплывающих потоков является **их «свежесть» — у такого потока нет истории**: регистров, стека и прочей информации, которую нужно восстанавливать. Всплывающие потоки абсолютно «стерильны» и идентичны, что позволяет создавать их быстро. Новый поток обрабатывает входящее сообщение. Использование всплывающих потоков позволяет **значительно сократить промежуток времени** между прибытием сообщения и началом его обработки.

При использовании всплывающих потоков необходимо предварительное планирование. Например, в каком процессе возникнет новый поток? Если система поддерживает потоки, работающие в контексте ядра, новый поток может возникнуть там (именно поэтому мы не показали ядро на Рис. 11). Создание всплывающих потоков в пространстве ядра всегда быстрее и проще, чем в пространстве пользователя. К тому же всплывающему потоку в пространстве ядра проще получить доступ ко всем таблицам ядра и устройств ввода-вывода, что может оказаться полезным при обработке прерываний. С другой стороны, наличие ошибок в потоке, расположенном в пространстве ядра, может нанести существенно больший ущерб. Например, если поток работает слишком долго и невозможно воспользоваться приоритетным прерыванием, это может привести к потере входных данных.

Рис. 11. Создание нового потока по прибытии сообщения: до прибытия сообщения (а); после прибытия сообщения (б)

Как сделать однопоточную программу многопоточной

Многие из существующих программ были написаны для однопоточных процессов. Сделать их многопоточными гораздо сложнее, чем это может показаться на первый взгляд. Ниже мы рассмотрим некоторые из возможных трудностей.

Прежде всего, программа потока обычно состоит из нескольких процедур, так же как и процесс. У этих процедур могут быть локальные переменные, глобальные переменные и параметры. Проблем с локальными переменными и параметрами не будет, зато проблемы будут с переменными, которые являются глобальными для потока, но не глобальными для всей программы. Эти переменные являются глобальными с точки зрения процедур одного потока (которые ими пользуются, как пользовались бы любыми другими глобальными переменными), но не имеют никакого отношения к другим потокам.

В качестве примера рассмотрим переменную `errno` в UNIX. Если процесс (или поток) выполняет неудачный системный запрос, код ошибки записывается в `errno`. На Рис. 12 поток 1 выполняет системный запрос `access`, чтобы узнать, имеет ли он разрешение на доступ к конкретному файлу. ОС возвращает ответ в глобальной переменной `errno`. После этого управление возвращается к потоку 1. Однако прежде, чем у него появляется возможность считать значение `errno`, планировщик решает, что поток 1 уже достаточно попользовался CPU и пора переключиться на поток 2. Поток 2 выполняет запрос `open`, завершающийся неудачей, в результате чего значение `errno` изменяется и предыдущее значение теряется. После того как поток 1 вновь получит управление, он прочитает неверное значение `errno`, и дальнейшие его действия будут неправильными.

Существует несколько различных решений проблемы. Одно из решений — **запретить глобальные переменные вообще**. Какой бы заманчивой ни была эта идея, она вступит в противоречие с большей частью существующего ПО. Другое решение — **предоставить каждому потоку собственные глобальные переменные**, как показано на Рис. 13. В этом случае конфликт исключается, поскольку у каждого потока будет своя копия `errno` и остальных глобальных переменных. Это решение фактически приводит к появлению новых уровней видимости переменных: переменные, доступные всем процедурам потока (в дополнение к уже имеющимся уровням видимости переменных, доступных только одной процедуре), и переменные, доступные всей программе.

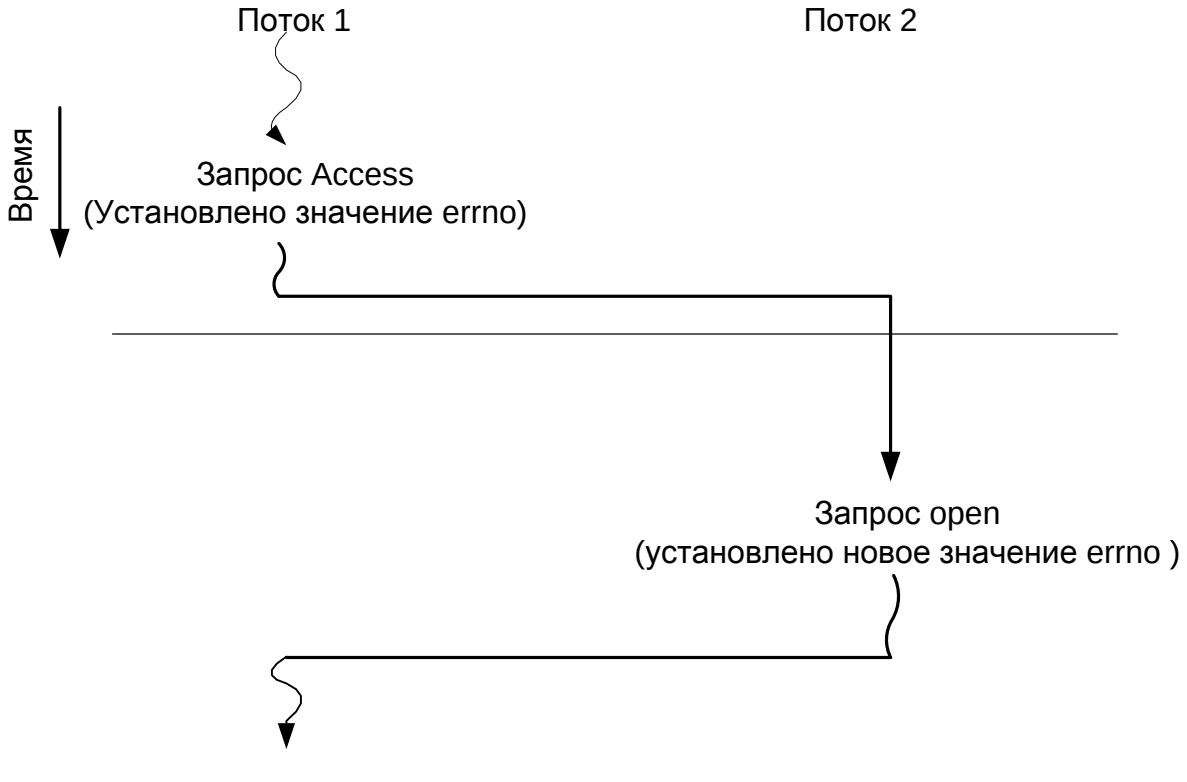

Рис. 12. Конфликт между потоками при использовании глобальной переменной

Ошибка! Объект не может быть создан из кодов полей редактирования.

Рис. 13. Возможность использования собственных глобальных переменных у потоков

Обеспечить доступ к собственным глобальным переменным не очень просто, поскольку в большинстве языков программирования есть способы описания локальных и глобальных переменных, но не промежуточных разновидностей. Можно отвести под глобальные переменные отдельный участок памяти и рассматривать их как дополнительные параметры процедур. Несмотря на некоторую неуклюжесть, этот метод работает.

В качестве альтернативы можно написать новые библиотечные процедуры, которые будут создавать, записывать и считывать переменные, глобальные для потока. Первый запрос будет выглядеть примерно так:

```
create_global("bufptr");
```

Этот запрос отводит участок памяти под указатель, называющийся `bufptr`, в динамической памяти или в отдельном участке памяти, зарезервированном Для вызывающего потока. Не имеет значения, где именно расположен этот участок памяти, важно, что лишь вызывающий поток имеет к нему доступ. Если другой поток создаст глобальную переменную с таким же именем, она будет размещаться в другом участке памяти и конфликта потоков не будет.

Для доступа к глобальной переменной нужно два запроса: один, чтобы записать ее значение, и другой — чтобы его считать. Для записи будет использоваться что-то вроде

```
set_global("bufptr",&buf);
```

Этот запрос сохраняет значение указателя в участке памяти, созданном запросом `create_global`. Запрос на чтение может выглядеть как:

```
bufptr=read_global("bufptr");
```

Запрос возвращает адрес для доступа к данным, хранящийся в глобальной переменной.

Другим препятствием может стать тот факт, что большинство библиотечных процедур не являются реентерабельными. Это означает, что при их написании не предполагалась ситуация, при которой процедуре будет необходимо ответить на второй запрос, не закончив ответа на первый. Например, пересылку сообщения по сети можно организовать следующим образом: сообщение помещается в буфер, затем эмулируется прерывание в ядро для его отсылки. Что произойдет, если один поток поместит сообщение в буфер, а затем прерывание по таймеру приведет к передаче управления второму потоку, который тут же поместит в этот буфер свое сообщение?

Подобная же проблема возникает с процедурами распределения памяти (`malloc` в UNIX), управляющими таблицами использования памяти (в виде связного списка доступных участков памяти). Пока процедура `malloc` занята переписыванием таблиц, таблицы могут временно находиться в несовместимом состоянии, с указателями, никуда не указывающими. Если в этот момент произойдет переключение потоков и от нового потока придет запрос, может быть использован неправильный указатель, что приведет к нарушению работы программы. Решение всех подобных проблем равнозначно полному переписыванию библиотеки.

Другим решением может быть снабжение каждой процедуры оберткой (wrapper), устанавливающим бит, означающий, что эта процедура используется. Любая попытка использования процедуры другим потоком до окончания выполнения предыдущего запроса блокируется. Этот метод можно использовать, но он практически исключает параллелизм.

Теперь рассмотрим сигналы. Одни из них связаны с потоками, тогда как другие — нет. Например, если поток выполняет запрос `alarm`, результирующий сигнал по логике должен вернуться к этому потоку. Однако если потоки реализованы в пространстве пользователя, ядро ничего не знает об их существовании и вряд ли направит сигнал к правильному потоку. Ситуация еще больше усложняется, если одновременно у процесса может быть только один необработанный аварийный сигнал, а несколько потоков выполняют запрос `alarm` независимо друг от друга.

Последняя проблема, связанная с потоками, — управление стеками. Во многих системах при переполнении стека процесса ядро автоматически увеличивает его. Если у процесса несколько потоков, стеков тоже должно быть несколько. Если ядро не знает о существовании этих стеков, оно не может их автоматически увеличивать при переполнении. Ядро может даже не связать ошибки памяти с переполнением стеков.

Литература

1. Э. Таненбаум. Современные операционные системы. 2-ое изд. –СПб.: Питер, 2002. – 1040 с.

2. А. Шоу. Логическое проектирование операционных систем. Пер. с англ. –М.: Мир, 1981. –360 с.

3. С. Кейслер. Проектирование операционных систем для малых ЭВМ: Пер. с англ. –М.: Мир, 1986. –680 с.

4. Э. Таненбаум, А. Вудхалл. Операционные системы: разработка и реализация. Классика CS. –СПб.: Питер, 2006. –576 с.

5. Microsoft Development Network. URL: http://msdn.com

Приложение А. Программная модель процессора i486. Аппаратная мультизадачность

CPU i486 обеспечивает аппаратную поддержку мультизадачности. Задачей называется программа, выполняемая в текущий момент, либо ожидающая выполнения во время работы другой программы. Задача запускается прерыванием, исключением, переходом или вызовом. Когда одна из этих форм передачи управления используется с назначением, заданным элементом одной из дескрипторных таблиц, этот дескриптор может иметь тип, вызывающий начало выполнение новой задачи после сохранения состояния текущей задачи. Существует два типа задаче-ориентированных дескрипторов, которые могут находиться в таблице дескрипторов:

- дескрипторы сегмента состояния задачи;
- шлюзы задачи.

Когда управление передается любому из таких дескрипторов, происходит переключение задачи.

Переключение задачи похоже на вызов процедуры, но оно выполняет сохранение большего количества информации о состоянии CPU. Вызов процедуры сохраняет только содержимое регистров общего назначение, а в некоторых случаях содержимое только одного регистра (`EIP`). При вызове процедуры содержимое сохраняемых регистров помещается в стек, чтобы процедура имела возможность вызвать сама себя. Когда процедура вызывает сама себя, она называется реентерабельной.

Переключение задачи передает выполнение в полностью иную среду, среду задачи. Для этого требуется сохранить содержимое практически всех регистров CPU, таких как регистр `EFLAGS`. В отличие от процедур, задачи не реентерабельны. Переключение задачи ничего не помещает в стек. Информация о состоянии CPU сохраняется в структуре данных в памяти, которая называется сегмент состояния задачи.

В число регистров и структур данных, поддерживающих мультизадачность, входят:

- Сегмент состояния задачи.
- Дескриптор сегмента состояния задачи.
- Регистр задачи
- Дескриптор шлюза задачи.

Используя эти структуры, CPU i486 может переключать выполнение с одной задачи на другую, сохраняя контекст текущей задачи, допуская тем самым рестарт другой задачи. Помимо простого переключения задач, CPU i486 предлагает еще два средства организации мультизадачности:

1. Переключение задачи может выполняться вследствие прерываний и исключений (если это требуется конструкции системы). CPU не только выполняет переключение задачи для обработки прерывания или исключения, но и автоматическое переключение назад, на прерванную задачу, после возврата из прерывания или исключения. Прерывания могут происходить и во время задач обработки прерывания.

2. При каждом переключении на другую задачу CPU i486 может также выполнять переключение на другую LDT. Это может использоваться для того, чтобы дать каждой задаче собственное отображение логических адресов в физические. Тем самым обеспечивается дополнительное средство защиты, поскольку задачи могут быть таким образом изолированы, и их взаимное влияние друг на друга исключено. Регистр PDBR также перезагружается. Это позволяет использовать механизм подкачки страниц для обеспечения изолированности задач.

Использование механизма мультизадачности является необязательным. Для некоторых прикладных программ этот способ организации выполнения программ не является лучшим.

A.1 Сегмент состояния задачи

Информация о состоянии CPU, необходимая для восстановления контекста задачи, хранится в типе сегмента, называемом сегментом состояния задачи, или TSS. На Рис. A.1 показан формат TSS для задачи, выполняемой центральным CPU i486 (совместимость с задачами 80286 обеспечивается другим типом TSS). Поля TSS делятся на две основные категории:

1. Динамические поля, обновляемые CPU при каждом переключении задачи. В число этих полей входят:

- Регистры общего назначения (EAX, ECX, EDX, EBX, ESP,EBP, ESI и EDI).
- Сегментные регистры (ES, CS, SS, DS, FS и GS).
- Регистр флагов (EFLAGS).
- Указатель команд (EIP),
- Селектор для TSS предыдущей задачи (обновляется только когда ожидается возврат).

2. Статические поля, которые CPU считывает, но не изменяет. Эти поля устанавливаются при создании задачи. Эти поля:

- Селектор для LDT задачи.
- Логический адрес для стеков привилегированных уровней 0, 1 и 2.
- Бит T (бит отладочной ловушки), который, будучи установленным, заставляет CPU устанавливать при переключении задачи отладочное исключение.
- Базовый адрес битового массива разрешения ввода/вывода. При наличии, данный массив всегда хранится в TSS по старшим адресам. Базовый адрес указывает на начало массива.

При использовании механизма подкачки страниц важно избегать помещения границы страницы в пределах части TSS, считываемой CPU при переключении задачи (первые 108 байтов). Если граница страницы находится в пределах этой части TSS, то страницы по обеим сторонам границы должны присутствовать в памяти одновременно. При отсутствии страницы или генерации исключения общей защиты после того, как CPU начал чтение TSS, возникает состояние невосстановимой ошибки.

A.2. Дескриптор TSS

Сегмент состояния задачи, как и все прочие сегменты, определяется дескриптором. Формат дескриптора TSS показан на Рис. A. 1.

31	15	0

Базовый адрес массива ввода/вывода	000000000000000	T
0000000000000000	Селектор для LDT задачи	
0000000000000000	GS	
0000000000000000	FS	
0000000000000000	DS	
0000000000000000	SS	

0000000000000000	CS
0000000000000000	ES
EDI	
ESI	
EBP	
ESP	
EBX	
EDX	
ECX	
EAX	
EFLAGS	
EIP	
RESERVES	
0000000000000000	SS2
ESP2	
0000000000000000	SS2
ESP1	
0000000000000000	SS1
ESP0	
0000000000000000	Компоновка задач (старый селектор TSS)

Рис. А. 1. Сегмент состояния задачи

Адреса показаны в шестнадцатеричном формате
ПРИМЕЧАНИЕ: Биты, помеченные как 0, резервируются. Не используйте их.

Бит "Занят" в поле Типа указывает на то, что задача занята. Занятой задачей называется текущая выполняемая или ожидающая выполнения задача.

Поле Типа со значением 9 указывает на неактивную задачу; значение 11 (десятичное) указывает на занятую задачу. Задачи нереентерабельны. CPU i486 использует бит занятости, для обнаружения попыток вызова задачи, выполнение которой прервано.

Рис. А. 2. Дескриптор TSS

Дадим пояснения к рисунку:

- AVL -Доступно для использования системным ПО.

- B - Бит "Занятости".
- BASE - Базовый адрес сегмента.
- DPL - Уровень привилегированности дескриптора.
- G – Грануляция.
- LIMIT - Граница сегмента.
- P - Присутствие сегмента.
- TYPE - Тип сегмента.

Поля Базового адреса сегмента, Границы и DPL, а также биты Грануляции и Присутствия выполняют функции, аналогичные тем, что были у них в дескрипторах сегментов данных. Поле Границы должно иметь значение, равное или больше чем 67H, на один байт меньше минимального размера сегмента состояния задачи. Попытка выполнить переключение на задачу, дескриптор TSS которой имеет границу меньше чем 67H, генерирует исключение. При использовании битового массива разрешения ввода/вывода требуется большее значение Границы. Большее значение Границы может также понадобиться для самой операционной системы, если система хранит в TSS дополнительные данные.

Процедура с доступом к дескриптору TSS может вызвать переключение задачи. В большинстве систем поля DPL дескрипторов TSS должны быть очищены, чтобы только привилегированное программное обеспечение могло выполнить переключение задачи.

Доступ к дескриптору TSS не дает процедуре возможности читать или модифицировать дескриптор. Чтение и модификация его возможны только путем отображения в тот же адрес памяти дескриптора данных. Загрузка дескриптора TSS в сегментный регистр вызывает исключение. Дескрипторы TSS могут находиться только в таблице GDT. Попытка доступа к TSS при помощи селектора с установленным битом TI (который обозначает текущую LDT) генерирует исключение.

A.3. Регистр задачи

Регистр задачи (TR) используется для поиска текущего TSS. На Рис. A. 3 показан путь, по которому CPU выполняет доступ к TSS.

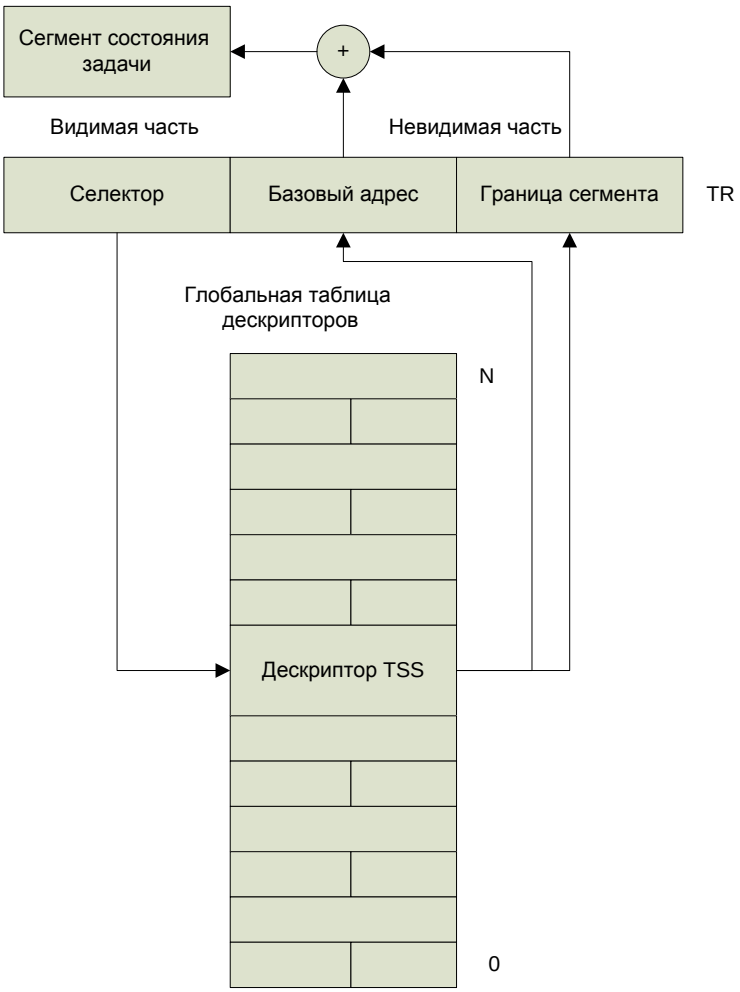

Рис. A. 3. Регистр TR

Регистр задачи имеет «видимую» часть (т.е. часть, которую может считывать и изменять программное обеспечение) и «невидимую» часть (т.е. часть, обслуживаемую CPU и недоступную программному обеспечению). Селектор, находящийся в видимой части, индексирует дескриптор TSS в GDT. CPUиспользует невидимую часть регистра TR для приема туда значений базы и границы из дескриптора TSS. Хранение в регистре этих значений делает выполнение задачи более эффективным, поскольку для ссылки к TSS текущей задачи CPU не требуется извлекать эти значения из памяти.

Команды LTR и STR используются для модификации и чтения видимой части регистра задачи. Обе эти команды принимают один операнд, а именно 16-разрядный селектор сегмента, расположенный в памяти или в регистре общего назначения.

LTR (Загрузить регистр задачи) загружает в видимую часть регистра задачи операнд, который должен индексировать позицию дескриптора TSS в GDT. Команда LDT также загружает в невидимую часть информацию из дескриптора TSS. Команда LTR является привилегированной командой: она может быть выполнена только при CPL равном 0. Команда LTR обычно используется во время инициализации системы для помещения в регистр задачи исходного значения: далее содержимое регистра TR изменяется событиями, вызывающими переключение задачи.

STR (Сохранить регистр задачи) записывает видимую часть регистра задачи в регистр общего назначения или в память. Команда STR не является привилегированной командой.

A.4. Дескриптор шлюза задачи

Дескриптор шлюза задачи обеспечивает косвенные, защищенные ссылки к задаче. Формат шлюза задачи показан на Рис. A. 4.

Поле Селектор шлюза задачи индексирует дескриптор TSS. RPL в данном селекторе не используется.

DPL шлюза задачи управляет доступом к дескриптору для переключения задачи. Процедура не может выбрать дескриптор шлюза задачи до тех пор, пока RPL селектора и CPL процедуры не будут численно меньше или равны DPL дескриптора. Тем самым предотвращается переключение задачи менее привилегированными, чем она сама, процедурами. (Отметим, что при использовании шлюза задачи DPL дескриптора TSS назначения не используется).

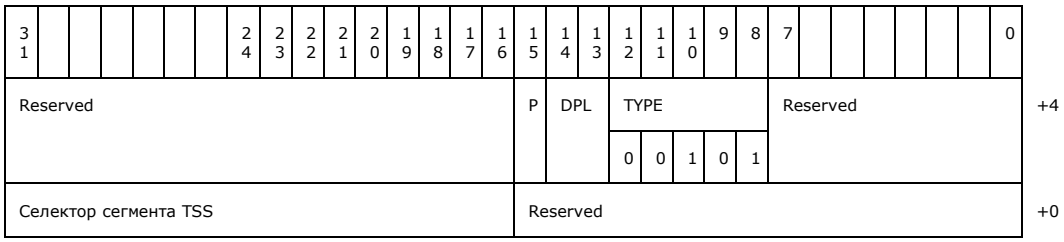

Рис. A. 4. Дескриптор шлюза задачи

DPL - Уровень привилегированности дескриптора.

P - Присутствие сегмента.

Процедура с доступом к шлюзу задачи может вызвать переключение задачи, как и процедура с доступом к дескриптору TSS. Как шлюзы задачи, так и дескрипторы TSS предназначены для решения следующих вопросов:

1. Необходимость иметь для задач только один бит Занятости. Поскольку этот бит хранится в дескрипторе TSS, каждая задача должна иметь только один такой дескриптор. Однако, может существовать несколько шлюзов задачи, выбирающих один дескриптор TSS.

2. Необходимость обеспечить селективный доступ к задачам. Шлюзы задачи позволяют решить эту проблему, поскольку они могут размещаться в LDT и иметь DPL, отличный от DPL дескриптора TSS. Процедура, недостаточно привилегированная для использования дескриптора TSS в GDT (который обычно имеет DPL, равный 0), может тем не менее вызвать другую задачу, если она имеет доступ к шлюзу задачи в LDT. Благодаря шлюзам задачи операционная система может ограничить переключение задач конкретными задачами.

3. Необходимость выполнения переключения задачи в случае прерывания или особой ситуации. Если прерывание или исключение передает в шлюз задачи вектор, CPU i486 выполняет переключение на указанную задачу. На Рис. А. 5 показано, как шлюз задачи в LDT и шлюз задачи в IDT могут идентифицировать одну и ту же задачу.

A.5. Переключение задачи

CPU i486 передает управление другой задаче в одном из следующих четырех случаев:

1. Текущая задача выполняет команду JMP или CALL для дескриптора TSS.

2. Текущая задача выполняет команду JMP или CALL для шлюза задачи.

3. Прерывание или исключение индексирует шлюз задачи в IDT.

4. Текущая задача выполняет команду IRET при установленном флаге NT.

Команды JMP, CALL и RET, равно как прерывания и исключения, представляют собой обычные механизмы CPU i486, которые могут быть использованы и при обстоятельствах, не приводящих к переключению задачи. Тип дескриптора (при вызове задачи) или флаг NT (при возврате из задачи) определяют разницу между стандартным механизмом и его формой, вызывающей переключение задачи. Для того, чтобы произошло переключение задачи, команда JMP или CALL может передать управление либо дескриптору TSS, либо шлюзу задачи. Эффект в обоих случаях одинаковый: CPU i486 передает управление требуемой задаче. Исключение или прерывание вызывают переключение задачи, индексируя шлюз задачи в IDT. Если они индексируют в IDT шлюз прерывания или шлюз ловушки, то переключения задачи не происходит.

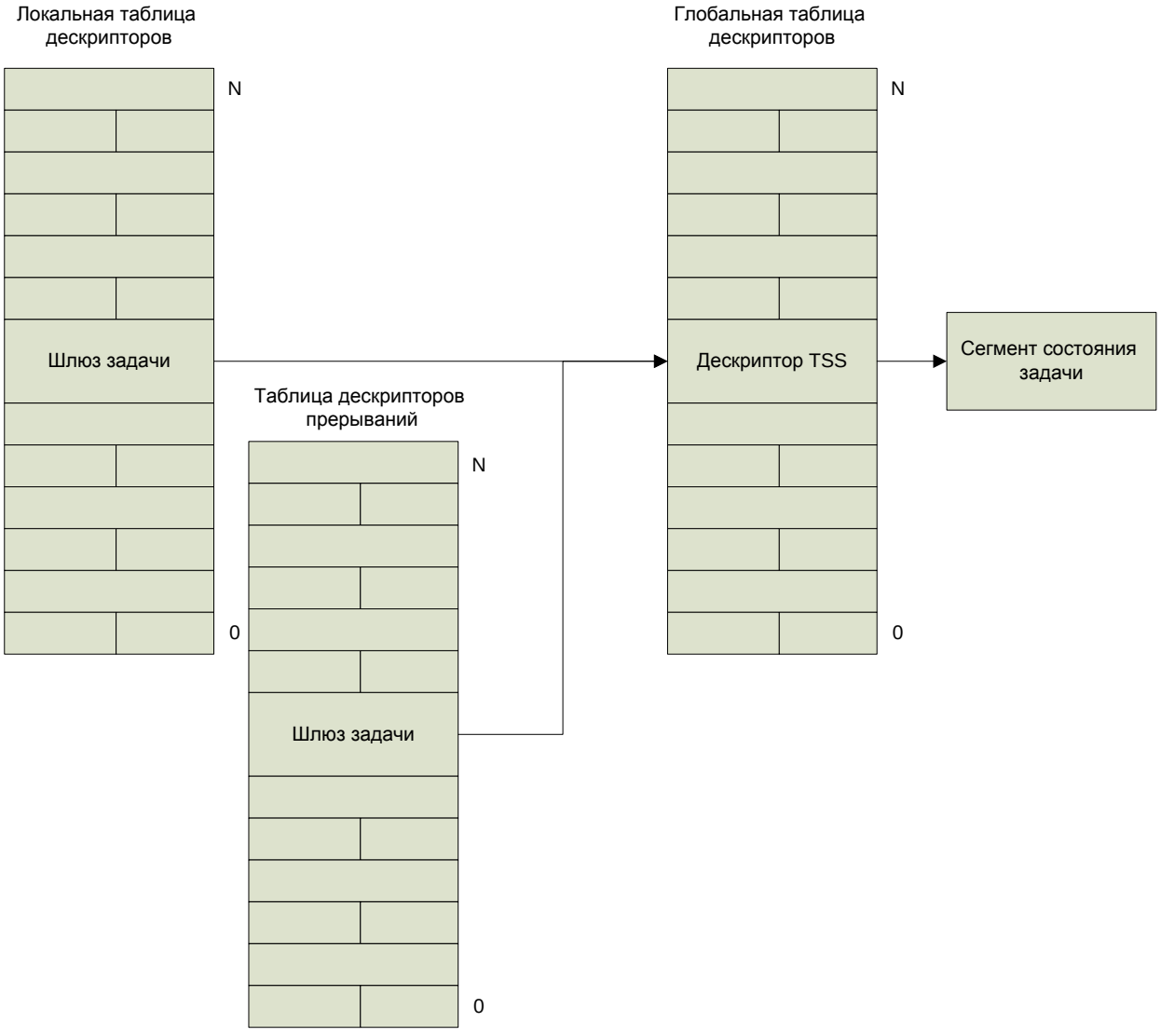

Рис. А. 5. Задачи со ссылками на шлюзы задачи

Подпрограмма обслуживания прерывания всегда возвращает выполнение в прерванную процедуру, которая может находиться в другой задаче. Если флаг NT очищен,

происходит нормальный возврат. Если флаг NT установлен, происходит переключение задачи. Задача, принимающая переключение, задается селектором TSS в TSS подпрограммы обслуживания прерывания.

Переключение задачи имеет следующие этапы:

1. Проверка того, что текущей задаче разрешено выполнить переключение на другую задачу. К командам JMP и CALL применимы правила привилегированности доступа к данным. DPL дескриптора TSS и шлюза задачи должен быть больше чем или равен одновременно CPL и RPL селектора шлюза. исключения, прерывания и команды IRET имеют право переключать задачу независимо от DPL шлюза задачи или дескриптора TSS назначения.

2. Проверка того, что дескриптор TSS новой задачи помечен как присутствующий и имеет допустимую границу (превышающую или равную 67H). Любые случившиеся до этой точки ошибки принадлежат контексту текущей задачи. При попытке выполнить приводящую к ошибке команду эти ошибки восстанавливают любые изменения состояния CPU. Благодаря этому адрес возврата для обработчика прерываний указывает на команду, вызвавшую ошибку, а не на команду, следующую за ней. Обработчик исключений может зафиксировать условие, вызвавшее ошибку, и выполнить рестарт задачи. Вмешательство обработчика исключений может быть полностью прозрачно для прикладной программы.

3. Сохранение состояния текущей задачи. CPU находит базовый адрес текущего TSS в регистре задачи. Регистры CPU копируются в текущий TSS (регистры EAX, ECX, EDX, EBX, ESP, EBP, ESI, EDI, ES, CS, SS, DS, FS, GS и EFLAGS).

4. Загрузка в регистр TR селектора для дескриптора TSS новой задачи, установка бита Занятости новой задачи и установка бита TS в регистре CR0. Селектор либо является операндом команды JMP или CALL, либо берется из шлюза задачи.

5. Загрузка состояния новой задачи из ее TSS и продолжение ее выполнения. При этом загружаются регистры LDTR, EFLAGS, регистры общего назначения EIP, EAX, ECX, EDX, EBX, ESP, EBP, ESI, EDI, а также сегментные регистры Es, CS, SS, DS, FS и GS. Любые ошибки, обнаруживаемые на этом шаге, принадлежат к контексту новой задачи. С точки зрения обработчика исключений первая команда новой задачи не является выполненной.

Отметим, что состояние старой задачи при переключении задачи всегда сохраняется. При возобновлении этой задачи выполнение ее продолжается с той команды, которая была бы выполнена следующей при обычной работе. Регистры восстанавливаются в те значения, которые они имели к моменту останова задачи для переключения.

Каждое переключение задачи устанавливает бит TS (Задача Переключена) Apar регистра CR0. Бит TS полезен системным программам для координации работы

целочисленного блока и блока операций с плавающей точкой или сопроцессора. Бит TS указывает на то, что содержимое блока операций с плавающей точкой или сопроцессора может отличиться от соответствующего содержимого для текущей задачи.

Подпрограммы обслуживания исключений, вызванных переключением задачи (исключения вследствие шагов 5 - 17 в Таблица A. 1) могут начать вызываться рекурсивно в случае попытки перезагрузить селектор сегмента, сгенерировавшего данное исключение. Причина исключения (или одна из нескольких причин) до повторения загрузки сегмента должна быть зафиксирована.

Уровень привилегированности, с которым выполнялась старая задача, не имеет отношения к уровню привилегированности новой задачи. Поскольку задачи изолированы друг от друга благодаря отдельным адресным пространствам и сегментам состояния задачи, и поскольку доступ к TSS выполняется по правилам привилегированности, переключение задачи не требует никаких проверок привилегированности. Новая задача начинает выполняться с уровнем привилегированности, указанным в RPL нового содержимого регистра CS, загружаемого из TSS.

Таблица A. 1. Проверки, выполняемые при переключении задачи

Шаг	Проверяемое условие	Особая ситуация	Ссылка на код ошибки
1	Дескриптор TSS присутствует в памяти	NP	TSS новой задачи

2	Дескриптор TSS не Занят	GP	TSS новой задачи
3	Граница сегмента TSS больше чем или равна 103	TS	TSS новой задачи
4	Загрузка регистров из значений, хранимых в TSS		
5	Допустимость селектора LDT новой задачи	TS	TSS новой задачи
6	DPL кодового сегмента соответствует RPL селектора	TS	Новый сегмент кода
7	Допустимость селектора SS	GP	Новый сегмент кода
8	Сегмент стека присутствует в памяти	SF	Новый сегмент стека
9	DPL кодового сегмента соответствует CPL селектора	SF	Новый сегмент стека
10	LDT новой задачи присутствует в памяти	TS	TSS новой задачи
11	Допустимость селектора CS	TS	Новый сегмент кода
12	Сегмент кода присутствует в памяти	NP	Новый сегмент кода
13	DPL сегмента стека соответствует RPL селектора	GP	Новый сегмент стека
14	Допустимость селекторов DS, ES, FS и GS	GP	Новый сегмент данных
15	Сегменты DS, ES, FS и GS доступны для чтения	GP	Новый сегмент данных
16	Сегменты DS, ES, FS и GS присутствуют в памяти	NP	Новый сегмент данных
17	DPL сегментов DS, ES, FS и GS больше или равен CPL (если эти сегменты не являются конформными)	GP	Новый сегмент данных

Примечание: Следующие CPU Intel могут использовать другой порядок проверок.

1. NP - исключение "Сегмент не присутствует"; GP - исключение общей защиты; TS - исключение "Неверный TSS"; SF -исключение "Сбой в стеке".

2. Селектор является допустимым, если он находится в таблице совместимого типа (например, селектор LDT не может находиться ни в одной таблице, кроме GDT), занимает адрес в пределах границы табличного сегмента и ссылается на совместимый тип дескриптора (например, селектор в регистре CS является допустимым только в том случае, если он индексирует дескриптор кодового сегмента; тип дескриптора задается в его поле Типа).

A.6 Компоновка задач

Для возврата выполнения на предыдущую задачу используются поле Компоновки в TSS и флаг NT. Флаг NT указывает на то, является ли текущая выполняемая задача вложенной в выполнение другой задачи, а поле Компоновки в TSS текущей задачи содержит селектор TSS для задачи более старшего уровня, если таковая имеется (см. Рис. A.6).

Когда прерывание, исключение, переход или вызов вызывают переключение задачи, CPU i486 копирует селектор сегмента состояния текущей задачи в TSS для новой задачи и устанавливает флаг NT. Флаг NT указывает на то, что поле Компоновки TSS было загружено селектором сохраненного TSS. Новая задача возвращает управление командой IRET. При выполнении команды IRET происходит проверка флага NT. Если он

установлен, то CPU выполняет переключение на предыдущую задачу. В Таблица А. 2 показано использование полей TSS, на которые воздействует переключение задачи.

Отметим, что флаг NT может быть модифицирован программным обеспечением, выполняемым на любом уровне привилегированности. Программа может установить свой бит NT и выполнить команду IRET, что будет иметь эффект запуска задачи, заданной в поле Компоновки TSS текущей задачи. Для предотвращения непредусмотренных переключений задачи операционная система должна инициализировать поле Компоновки каждого создаваемого ей TSS.

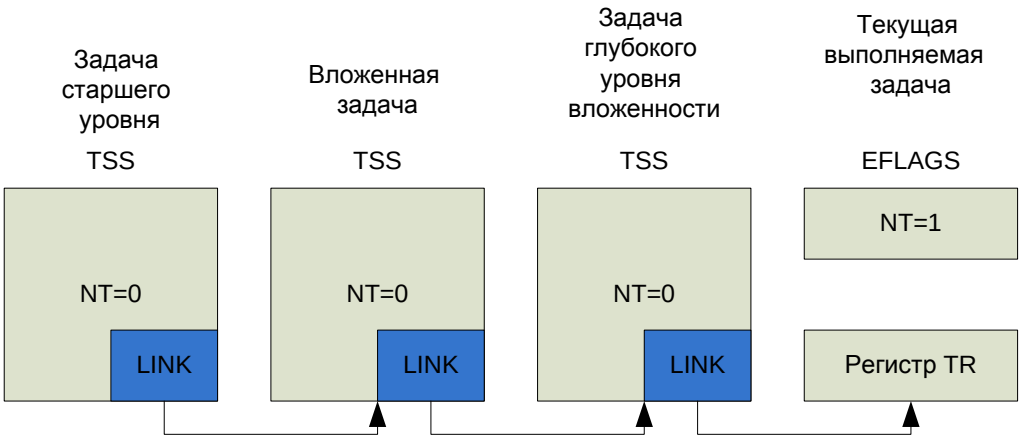

Рис. А. 6. Вложенные задачи

LINK - Поле Компоновки задач

Таблица А. 2. Воздействие переключения задачи на поля Занятости, NT и Компоновки

Поле	Воздействие JUMP	Воздействие команды CALL или прерывания	Воздействие команды IRET
Бит Занятости новой задачи	Бит установлен. Перед этим должен быть очищен	Воздействие команды CALL или прерывания	Воздействие команды IRET
Бит Занятости новой задачи	Бит установлен. Перед этим должен быть очищен	Бит установлен. Перед этим должен быть очищен	Изменений нет. Должен быть установлен
Бит Занятости старой задачи	Бит очищен	Изменений нет. В текущий момент бит установлен	Бит очищен
Флаг NT новой задачи	Флаг очищен	Флаг установлен	Изменений нет
Флаг NT старой задачи	Изменений нет	Изменений нет	Флаг очищен
Поле Компоновки новой задачи	Изменений нет	Загружено селектором для TSS старой задачи	Изменений нет
Поле Компоновки старой задачи	Изменений нет	Изменений нет	Изменений нет

А.6.1 Бит Занятости предотвращает зацикливание

Бит Занятости дескриптора TSS предотвращает реентерабельные переключения задач. Существует лишь один сохраненный контекст каждой задачи, а именно контекст,

сохраненный в TSS, следовательно, задача до своего завершения может быть вызвана только один раз. Цепочка отложенных задач может вырасти до любой длины вследствие множественных прерываний, исключений, переходови вызовов. Бит Занятости предотвращает вызов задачи, поставленной в такую цепочку. Реентерабельное же переключение задачи затрет старый TSS задачи, что приведет к разрушению всей цепочки.

CPU организует бит Занятости следующим образом:

1. При переключении задачи CPU устанавливает бит Занятости новой задачи.

2. При обратном переключении из задачи CPU очищает бит занятости старой задачи, если эта задача не должна быть поставлена в цепочку (т.е. команда, вызвавшая переключение задачи, это команда JMP или IRET). Если задача поставлена в цепочку, то ее бит Занятости остается установленным.

3. При переключении на задачу CPU генерирует исключение общей защиты, если бит Занятости новой задачи оказывается уже установленным.

Таким образом, CPU предотвращает переключение задачи самой на себя, либо на любую задачу в цепочке задач, что исключает реентерабельное переключение задачи.

Бит Занятости может использоваться в multi-CPU конфигурации системы, поскольку при установке или очистке бита Занятости CPU захватывает шину. Это исключает одновременный запуск одной и той же задачи двумя CPU.

A.6.2 Модификация компоновки задач

Для возобновления выполнения прерванной задачи до выполнения прервавшей ее задачи может понадобиться модификация цепочки отложенных задач. Надежный способ состоит в следующем:

1. Запретить прерывания.

2. Сначала изменить поле Компоновки TSS задачи прерывания, а затем очистить бит Занятости в дескрипторе TSS задачи, удаляемой из цепочки.

3. Снова разрешить прерывания.

A.7 Адресное пространство задачи

Для того, чтобы каждая задача имела собственную LDT и собственные таблицы страниц, могут быть использованы селектор LDT и поле PDBR (в CR3) TSS. Поскольку дескрипторы сегментов в LDT представляют собой связки между задачами и сегментами, для установки индивидуального управления этими связками для каждой задачи могут использоваться отдельные LDT. Доступ к любому конкретному сегменту может быть передан любой конкретной задаче путем помещения дескриптора этого сегмента в LDT задачи. При разрешенном механизме подкачки каждая задача может иметь свой собственный набор страничных таблиц для отображения линейных адресов в физические.

Задачи могут также иметь общую LDT. Это простой и эффективный по затратам памяти способ организации коммуникации между задачами или управления одних задач другими, не снимая защитных барьеров в системе в целом.

Поскольку все задачи имеют доступ к GDT, также возможно создание разделяемых сегментов, доступ к которым будет происходить через сегментные дескрипторы из данной таблицы.

A.7.1 Отображение линейного адресного пространства задачи в физическое

Способы организации отображения линейного адресного пространства задачи в физическое делятся на два общих класса:

1. Единое отображение линейного адресного пространства в физическое, разделяемое всеми задачами. Если механизм подкачки страниц запрещен, то этот способ единственный. Без подкачки страниц все линейные адреса отображаются в те же физические адреса. При разрешенной подкачке страниц данная форма отображения достигается путем использования одного каталога страниц для всех задач. Линейное адресное пространство может превышать физическое пространство, если поддерживается виртуальная память с подкачкой по обращению.

2. Независимое отображение линейного адресного пространства в физическое. Эта форма отображения существует при использовании собственного каталога страниц для каждой задачи. Поскольку PDBR (базовый регистр каталога страниц) загружается из TSS при каждом переключении задачи, каждая задача может иметь собственный страничный каталог.

Линейные адресные пространства различных задач могут отображаться в полностью раздельные физические адреса. Если элементы разных страничных каталогов указывают на разные страничные таблицы, а эти таблицы указывают на разные страницы физической памяти, то такие задачи не разделяют никаких физических адресов памяти.

Сегменты состояния задачи должны находиться в пространстве, доступном всем задачам, таким образом чтобы отображение адресов TSS не изменялось во время чтения или обновления CPU TSS при переключении задачи. Линейное пространство, в которое отображается GDT, также должно быть разделяемым физическим пространством; в противном случае теряется смысл GDT. На Рис. A. 7 показано, как линейные пространства двух задач могут перекрываться в физическом адресном пространстве при разделении ими страничных таблиц.

A.7.2 Логическое адресное пространство задачи

Само по себе отображение линейного адресного пространства в физическое с перекрытием не позволяет разделение данных задачами. Для разделения данных задачи должны также иметь общее отображение логического адресного пространства в линейное, т.е. они также должны иметь доступ к дескрипторам, указывающим на разделяемое линейное адресное пространство. Существует три способа создания разделяемого отображения логического адресного пространства в физическое:

1. При помощи сегментных дескрипторов в GDT. Все задачи имеют доступ к дескрипторам в GDT. Если эти дескрипторы указывают на линейное адресное пространство, отображаемое в общее для всех задач физическое адресное пространство, то задачи могут разделять данные и команды.

2. При помощи разделяемых LDT. Две или более задачи могут использовать одну и ту же LDT, если селекторы LDT в их TSS для использовании при трансляции адресов выбирают одну и ту же LDT. Сегментные дескрипторы в LDT, адресующие линейные пространства, отображаемые в перекрывающиеся физическое адресное пространство, обеспечивают разделяемую физическую память. Этот метод разделения более селективен, чем метод организации разделения посредством GDT, поскольку он позволяет ограничить разделение конкретными задачами. Прочие задачи в системе могут иметь другие LDT, не дающие им доступ к разделяемым областям памяти.

3. При помощи сегментных дескрипторов в LDT, отображающихся в одно и то же линейное адресное пространство. Если линейное адресное пространство отображается в то же самое физическое адресное пространство за счет постраничного отображения участвующих задач, эти дескрипторы позволяют задачам разделение адресного пространства. Такие дескрипторы обычно называют "алиасами". Такой метод разделения даже более селективен, чем предыдущий: прочие дескрипторы в LDT могут указывать на независимые линейные адреса, не являющиеся разделяемыми.

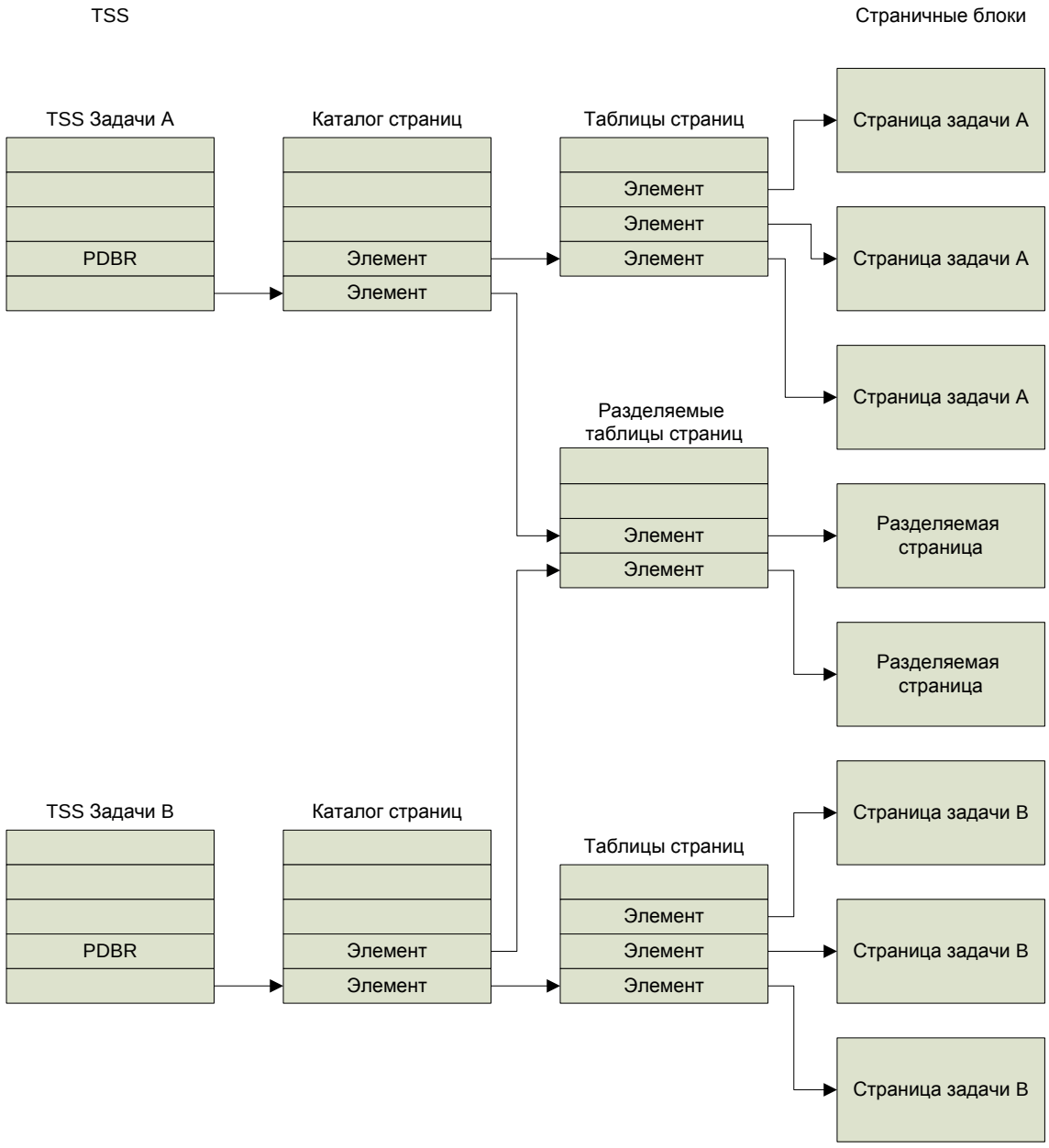

Рис. А. 7. Отображение линейного адресного пространства в физическое, с перекрытием

Литература

1. Э. Таненбаум. Современные операционные системы. 2-ое изд. –СПб.: Питер, 2002. – 1040 с.

2. А. Шоу. Логическое проектирование операционных систем. Пер. с англ. –М.: Мир, 1981. –360 с.

3. С. Кейслер. Проектирование операционных систем для малых ЭВМ: Пер. с англ. –М.: Мир, 1986. –680 с.

4. Э. Таненбаум, А. Вудхалл. Операционные системы: разработка и реализация. Классика CS. –СПб.: Питер, 2006. –576 с.

5. Microsoft Development Network. URL: http://msdn.com