**Тезаурусный словарь**

Пользовательское пространство — адресное пространство виртуальной памяти операционной системы, отводимое для пользовательских программ, в отличие от пространства ядра, которое резервируется для работы ядра операционной системы, его расширений и, возможно, некоторых драйверов устройств. В англоязычной компьютерной литературе термин «Userland» часто используется для обозначения совокупности приложений, которые выполняются в пространстве пользователя.

Режим супервизора, привилегированный режим, режим ядра (англ. kernel mode) — привилегированный режим работы процессора, как правило, используемый для выполнения ядра операционной системы.

В данном режиме работы процессора доступны привилегированные операции, такие, как операции ввода-вывода к периферийным устройствам, изменение параметров защиты памяти, настроек виртуальной памяти, системных параметров и прочих параметров конфигурации. Как правило, в режиме супервизора или вообще не действуют ограничения защиты памяти, или же они могут быть произвольным образом изменены, поэтому код, работающий в данном режиме, как правило, имеет полный доступ ко всем системным ресурсам (адресное пространство, регистры конфигурации процессора и так далее). Во многих типах процессоров это наиболее привилегированный режим из всех доступных режимов.

**Ментальная карта**

**Реализация потоков в пользовательском пространстве**

Реализация в пользовательском ядре

Реализация в пользовательском пространстве

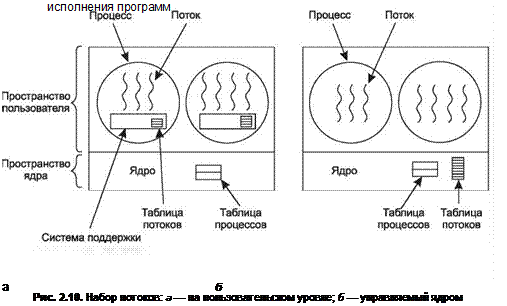
# Реализация потоков в пользовательском пространстве

Есть два основных места реализации набора потоков: в пользовательском пространстве и в ядре. Это утверждение носит несколько спорный характер, поскольку возможна еще и гибридная реализация. А теперь мы опишем эти способы со всеми их достоинствами и недостатками.

Первый способ — это поместить весь набор потоков в пользовательском пространстве. И об этом наборе ядру ничего не известно. Что касается ядра, оно управляет обычными, однопотоковыми процессами. Первое и самое очевидное преимущество состоит в том, что набор потоков на пользовательском уровне может быть реализован в операционной системе, которая не поддерживает потоки. Под эту категорию подпадают все операционные системы, даже те, которые еще находятся в разработке. При этом подходе потоки реализованы с помощью библиотеки.

У всех этих реализаций одна и та же общая структура (рис. 2.10, а). Потоки запускаются поверх системы поддержки исполнения программ (run-time system), которая представляет собой набор процедур, управляющих потоками. Четыре из них: pthread\_create, pthread\_exit, pthread\_join и pthread\_yield — мы уже рассмотрели, но обычно в наборе есть и другие процедуры.

Когда потоки управляются в пользовательском пространстве, каждому процессу необходимо иметь собственную таблицу потоков, чтобы отслеживать потоки, имеющиеся в этом процессе. Эта таблица является аналогом таблицы процессов, имеющейся в ядре,



за исключением того, что в ней содержатся лишь свойства, принадлежащие каждому потоку, такие как счетчик команд потока, указатель стека, регистры, состояние и т. д. Таблица потоков управляется системой поддержки исполнения программ. Когда поток переводится в состояние готовности или блокируется, информация, необходимая для возобновления его выполнения, сохраняется в таблице потоков, точно так же, как ядро хранит информацию о процессах в таблице процессов.

Когда поток совершает какие-то действия, которые могут вызвать его локальную блокировку, например ожидание, пока другой поток его процесса не завершит какую- нибудь работу, он вызывает процедуру системы поддержки исполнения программ. Эта процедура проверяет, может ли поток быть переведен в состояние блокировки. Если может, она сохраняет регистры потока (то есть собственные регистры) в таблице потоков, находит в таблице поток, готовый к выполнению, и перезагружает регистры машины сохраненными значениями нового потока. Как только будут переключены указатель стека и счетчик команд, автоматически возобновится выполнение нового потока. Если машине дается инструкция сохранить все регистры и следующая инструкция — загрузить все регистры, то полное переключение потока может быть осуществлено за счет всего лишь нескольких инструкций. Переключение потоков, осуществленное таким образом, по крайней мере на порядок, а может быть, и больше, быстрее, чем перехват управления ядром, что является веским аргументом в пользу набора потоков, реализуемого на пользовательском уровне.

Но у потоков есть одно основное отличие от процессов. Когда поток на время останавливает свое выполнение, например когда он вызывает thread\_yield, код процедуры thread\_yield может самостоятельно сохранять информацию о потоке в таблице потоков.

Более того, он может затем вызвать планировщик потоков, чтобы тот выбрал для выполнения другой поток. Процедура, которая сохраняет состояние потока, и планировщик — это всего лишь локальные процедуры, поэтому их вызов намного более эффективен, чем вызов ядра. Помимо всего прочего, не требуется перехват управления

ядром, осуществляемый инструкцией trap, не требуется переключение контекста, кэш в памяти не нужно сбрасывать на диск и т. д. Благодаря этому планировщик потоков работает очень быстро.

У потоков, реализованных на пользовательском уровне, есть и другие преимущества. Они позволяют каждому процессу иметь собственные настройки алгоритма планирования. Например, для некоторых приложений, которые имеют поток сборщика мусора, есть еще один плюс — им не следует беспокоиться о потоках, остановленных в неподходящий момент. Эти потоки также лучше масштабируются, поскольку потоки в памяти ядра безусловно требуют в ядре пространства для таблицы и стека, что при очень большом количестве потоков может вызвать затруднения.

Но несмотря на лучшую производительность, у потоков, реализованных на пользовательском уровне, есть ряд существенных проблем. Первая из них — как реализовать блокирующие системные вызовы. Представьте, что поток считывает информацию с клавиатуры перед нажатием какой-нибудь клавиши. Мы не можем разрешить потоку осуществить настоящий системный вызов, поскольку это остановит выполнение всех потоков. Одна из главных целей организации потоков в первую очередь состояла в том, чтобы позволить каждому потоку использовать блокирующие вызовы, но при этом предотвратить влияние одного заблокированного потока на выполнение других потоков. Работая с блокирующими системными вызовами, довольно трудно понять, как можно достичь этой цели без особого труда.

Все системные вызовы могут быть изменены и превращены в неблокирующие (например, считывание с клавиатуры будет просто возвращать нуль байтов, если в буфере на данный момент отсутствуют символы), но изменения, которые для этого необходимо внести в операционную систему, не вызывают энтузиазма. Кроме того, одним из аргументов за использование потоков, реализованных на пользовательском уровне, было именно то, что они могут выполняться под управлением существующих операционных систем. Вдобавок ко всему изменение семантики системного вызова read потребует изменения множества пользовательских программ.

В том случае, если есть возможность заранее сообщить, будет ли вызов блокирующим, существует и другая альтернатива. В большинстве версий UNIX существует системный вызов select, позволяющий сообщить вызывающей программе, будет ли предполагаемый системный вызов read блокирующим. Если такой вызов имеется, библиотечная процедура read может быть заменена новой процедурой, которая сначала осуществляет вызов процедуры select и только потом — вызов read, если он безопасен (то есть не будет выполнять блокировку). Если вызов read будет блокирующим, он не осуществляется. Вместо этого запускается выполнение другого потока. В следующий раз, когда система поддержки исполнения программ получает управление, она может опять проверить, будет ли на этот раз вызов read безопасен. Для реализации такого подхода требуется переписать некоторые части библиотеки системных вызовов, что нельзя рассматривать в качестве эффективного и элегантного решения, но все же это тоже один из вариантов. Код, который помещается вокруг системного вызова с целью проверки, называется конвертом (jacket), или оболочкой, или оберткой (wrapper).

С проблемой блокирующих системных вызовов несколько перекликается проблема ошибки отсутствия страницы. Мы изучим эту проблему в главе 3. А сейчас достаточно сказать, что компьютеры могут иметь такую настройку, что в одно и то же время в оперативной памяти находятся не все программы. Если программа вызывает инструкции (или переходит к инструкциям), отсутствующие в памяти, возникает ошибка обращения к отсутствующей странице и операционная система обращается к диску и получает отсутствующие инструкции (и их соседей). Это называется ошибкой вызова отсутствующей страницы. Процесс блокируется до тех пор, пока не будет найдена и считана необходимая инструкция. Если ошибка обращения к отсутствующей странице возникает при выполнении потока, ядро, которое даже не знает о существовании потоков, как и следовало ожидать, блокирует весь процесс до тех пор, пока не завершится дисковая операция ввода-вывода, даже если другие потоки будут готовы к выполнению.

Использование набора потоков, реализованного на пользовательском уровне, связано еще с одной проблемой: если начинается выполнение одного из потоков, то никакой другой поток, принадлежащий этому процессу, не сможет выполняться до тех пор, пока первый поток добровольно не уступит центральный процессор. В рамках единого процесса нет прерываний по таймеру, позволяющих планировать работу процессов по круговому циклу (поочередно). Если поток не войдет в систему поддержки выполнения программ по доброй воле, у планировщика не будет никаких шансов на работу.

Проблему бесконечного выполнения потоков можно решить также путем передачи управления системе поддержки выполнения программ за счет запроса сигнала (прерывания) по таймеру с периодичностью один раз в секунду, но для программы это далеко не самое лучшее решение. Возможность периодических и довольно частых прерываний по таймеру предоставляется не всегда, но даже если она и предоставляется, общие издержки могут быть весьма существенными. Более того, поток может также нуждаться в прерываниях по таймеру, мешая использовать таймер системе поддержки выполнения программ.

Другой наиболее сильный аргумент против потоков, реализованных на пользовательском уровне, состоит в том, что программистам потоки обычно требуются именно в тех приложениях, где они часто блокируются, как, к примеру, в многопоточном веб-сервере. Эти потоки часто совершают системные вызовы. Как только для выполнения системного вызова ядро осуществит перехват управления, ему не составит особого труда заняться переключением потоков, если прежний поток заблокирован, а когда ядро займется решением этой задачи, отпадет необходимость постоянного обращения к системному вызову select, чтобы определить безопасность системного вызова read. Зачем вообще использовать потоки в тех приложениях, которые, по существу, полностью завязаны на скорость работы центрального процессора и редко используют блокировку? Никто не станет всерьез предлагать использование потоков при вычислении первых n простых чисел или при игре в шахматы, поскольку в данных случаях от них будет мало проку.