УВАЖАЕМЫЕ СТУДЕНТЫ! Изучите приведенную лекцию, законспектируйте основные понятия, дайте ответы на контрольные вопросы.

Ответы на вопросы, фотоотчет, предоставить преподавателю на e-mail: r.bigangel@gmail.com до 03.04.2023.

При возникновении вопросов по приведенному материалу обращаться по следующему номеру телефона: (072)111-37-59, (Viber, WhatsApp), vk.com: https://vk.com/daykini

ВНИМАНИЕ!!! При отправке работы, не забывайте указывать ФИО студента, наименование дисциплины, дата проведения занятия (по расписанию).

Лекция

Реализация потоков в ядре. Гибридная реализация. Всплывающие потоки

1. **Реализация потоков в ядре, гибридная реализация, активация планировщика**

Реализация потоков в ядре

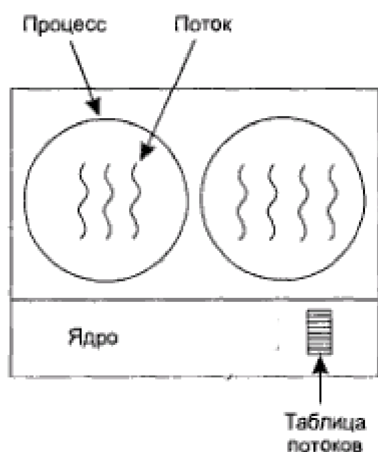

Рисунок 3. Пакет потоков, управляемый ядром.

При реализации потов в ядре не нужна система поддержки исполнения программ, здесь нет и таблицы процессов в каждом потоке. Вместо этого у ядра есть таблица потоков, в которой отслеживаются все потоки в системе. Когда потоку необходимо создать новый или уничтожить существующий поток, он обращается к ядру.

В таблице потоков, находящейся в ядре, содержатся регистры каждого потока, состояние и другая информация. Вся информация аналогична той, которая использовалась для потоков, создаваемых на пользовательском уровне, но теперь она содержится в ядре, а не в пространстве пользователя. Вдобавок к этому ядро также поддерживает традиционную таблицу процессов с целью их отслеживания.

Все вызовы, способные заблокировать поток, реализованы как системные вызовы, с более существенными затратами, чем вызов процедуры в системе поддержки исполнения программ. Когда поток блокируется, ядро по своему выбору может запустить либо другой поток из этого же самого процесса, либо поток из другого процесса. Когда потоки реализуются на пользовательском уровне, система поддержки исполнения программ работает с запущенными потоками своего собственного процесса до тех пор, пока ядро не заберет у нее центральный процессор.

Для потоков, реализованных на уровне ядра, не требуется никаких новых, неблокирующих системных вызовов. Главный недостаток этих потоков состоит в весьма существенных затратах времени на системный вызов, поэтому если операции над потоками (создание, удаление и т.п.) проводятся довольно часто, то это влечет за собой более существенные издержки.

Хотя потоки, создаваемые на уровне ядра, и позволяют решить ряд проблем, но справиться со всеми существующими проблемами они не в состоянии.

(Хотя потоки на уровне ядра по ряду ключевых позиций превосходят потоки на уровне пользователя, они, несомненно, более медлительны.)

Гибридная реализация

В попытках объединить преимущества создания потоков на уровне пользователя и на уровне ядра была исследована масса различных путей. Один из них на рис. 4, заключается в использовании потоков на уровне ядра, а затем нескольких потоков на уровне пользователя в рамках некоторых или всех потоков на уровне ядра. При использовании такого подхода программист может определить, сколько потоков использовать на уровне ядра и на сколько потоков разделить каждый из них на уровне пользователя. Эта модель обладает максимальной гибкостью.

Рис 4. Разделение на пользовательские потоки в рамках потока ядра

При таком подходе ядру известно только о потоках самого ядра, работу которых оно и планирует. У некоторых из этих потоков могут быть несколько потоков на пользовательском уровне, которые расходятся от их вершины. Создание, удаление и планирование выполнения этих потоков осуществляется точно так же, как и у пользовательских потоков, принадлежащих процессу, запущенному под управлением операционной системы, не способной на многопоточную работу. В этой модели каждый поток на уровне ядра обладает определенным набором потоков на уровне пользователя, которые используют его по очереди.

Активация планировщика

Цель работы по активации планировщика заключается в имитации функциональных возможностей потоков на уровне ядра, но при лучшей производительности и более высокой гибкости, свойственной пакетам потоков, реализуемых в пользовательском пространстве. В частности, пользовательские потоки не должны осуществлять специальные неблокирующие системные вызовы или заранее проверять, будет ли безопасным осуществление конкретного системного вызова. Тем не менее когда поток блокируется на системном вызове или на ошибке обращения к отсутствующей странице, должна оставаться возможность выполнения другого потока в рамках того же процесса, если есть хоть один готовый к выполнению поток.

Эффективность достигается путем уклонения от ненужных переходов между пространствами пользователя и ядра.

При использовании активации планировщика ядро назначает каждому процессу определенное количество виртуальных процессоров, а системе поддержки исполняемых программ (в пользовательском пространстве) разрешается распределять потоки по процессорам. Этот механизм также может быть использован на мультипроцессорной системе, где виртуальные процессоры могут быть представлены настоящими центральными процессорами. Изначально процессу назначается только один виртуальный

процессор, но процесс может запросить дополнительное количество процессоров, а также вернуть уже неиспользуемые процессоры. Ядро может также забрать назад уже распределенные виртуальные процессоры с целью переназначения их более нуждающимся процессам.

Работоспособность этой схемы определяется следующей основной идеей: когда ядро знает, что поток заблокирован, оно уведомляет принадлежащую процессу систему поддержки исполнения программ, передавая через стек в качестве параметров номер данного потока и описание произошедшего события. Уведомление осуществляется за счет того, что ядро активирует систему поддержки исполнения программ с заранее известного стартового адреса, — примерно так же, как действуют сигналы в UNIX. Этот механизм называется upcall (вызовом наверх).

Активированная таким образом система поддержки исполнения программ может перепланировать работу своих потоков, как правило, за счет перевода текущего потока в заблокированное состояние и выбирая другой поток из списка готовых к выполнению, устанавливая значения его регистров и возобновляя его выполнение. Чуть позже, когда ядро узнает, что исходный поток может возобновить свою работу (например, заполнился канал, из которого он пытался считать данные, или была извлечена из диска ранее не существующая страница), оно осуществляет еще один вызов наверх (upcall) в адрес системы поддержки исполнения программ, чтобы уведомить ее об этом событии. Система поддержки исполнения программ по собственному усмотрению может либо немедленно возобновить выполнение заблокированного потока, либо поместить его в список ожидающих потоков для последующего выполнения.

Недостатком активаций планировщика является полная зависимость этой технологии от вызовов наверх (upcall) — концепции, нарушающей структуру, свойственную любой многоуровневой системе.

2. Всплывающие потоки, превращение однопоточного кода в многопоточный

Всплывающие потоки

Рассмотрим суть на примере обработки входящих сообщений: в традиционном варианте имеется поток, который блокируется, ожидая входящего сообщения, когда мы имеем дело со ВСПЛЫВАЮЩИМ ПОТОКОМ, по прибытии сообщения система создает новый поток для его обработки

Использование всплывающих потоков позволяет значительно сократить промежуток времени между прибытием сообщения и началом его обработки.

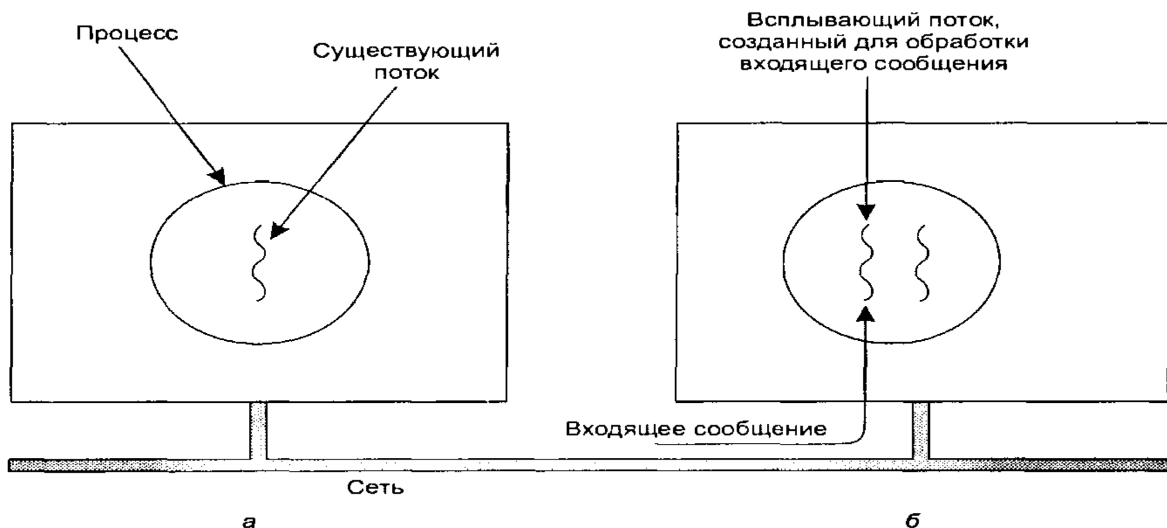

Рис. 1 – Создание нового потока прибытия сообщения: до прибытия(а); после прибытия(б)

При использовании всплывающих потоков необходимо предварительное планирование. Создание всплывающих потоков в ядре всегда проще и быстрее, чем в пространстве пользователя. Если система поддерживает потоки, работающие в контексте ядра, новый поток может возникнуть там. 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.

2.2. Превращение однопоточного кода в многопоточный

Многие из существующих программ были написаны для однопоточных процессов. Сделать их многопоточными гораздо сложнее, чем это может показаться на первый взгляд.

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.

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 программного обеспечения. Другое решение — предоставить каждому потоку собственные глобальные переменные. 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 .

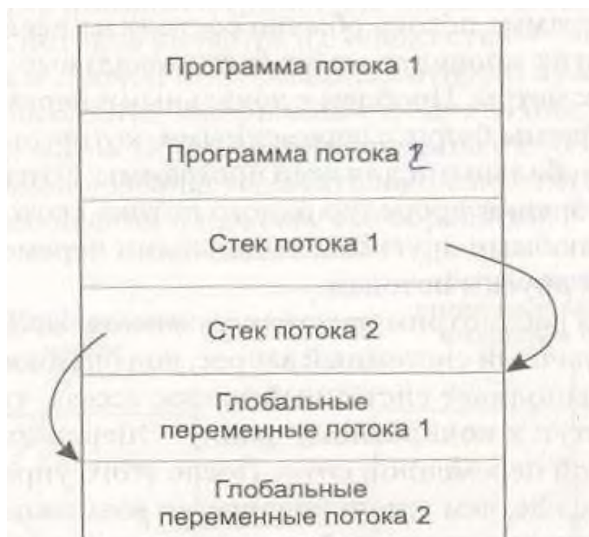

Рис. 3 – Собственные переменные у потоков.

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.

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.

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.

3.

Межпроцессное взаимодействие.

Процессам часто бывает необходимо взаимодействовать между собой. Поэтому необходимо правильно организованное взаимодействие между процессами, по возможности не использующее прерываний. **Проблема разбивается на три пункта. Первый: передача информации от одного процесса другому.**

Второй связан с контролем над деятельностью процессов: как гарантировать, что два процесса не пересекутся в критических ситуациях.

Третий касается согласования действий процессов: если процесс А должен поставлять данные, а процесс В выводить их на печать, то процесс В должен подождать и не начинать печатать, пока не поступят данные от процесса А

Состояние состязания.

Состояние состязания - **ситуация, когда несколько процессов считывают или записывают данные (в память или файл) одновременно.**

Критическая область.

Критическая область - часть программы, в которой есть обращение к совместно используемым данным.

Условия избегания состязания и эффективной работы процессов:

1. Два процесса не должны одновременно находиться в критических областях.
2. Процесс, находящийся вне критической области, не может блокировать другие процессы.
3. Невозможна ситуация, когда процесс вечно ждет (зависает) попадания в критическую область.