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

Потоки. Определение. Классическая модель потоков. Реализация потоков в пользовательском пространстве.

Поток (thread) — это, сущность операционной системы, процесс выполнения на процессоре набора инструкций, точнее говоря программного кода. Общее назначение потоков — параллельное выполнение на процессоре двух или более различных задач. Как можно догадаться, потоки были первым шагом на пути к многозадачным ОС. Планировщик ОС, руководствуясь приоритетом потока, распределяет кванты времени между разными потоками и ставит потоки на выполнение.

На ряду с потоком, существует также такая сущность, как процесс. **Процесс** (process) — не что более иное, как некая абстракция, которая инкапсулирует в себе все ресурсы процесса (открытые файлы, файлы отображенные в память...) и их дескрипторы, потоки и т.д. Каждый процесс имеет как минимум один поток. Также каждый процесс имеет свое собственное виртуальное адресное пространство и контекст выполнения, а потоки одного процесса разделяют адресное пространство процесса.
Каждый поток, как и каждый процесс, имеет свой контекст. Контекст — это структура, в которой сохраняются следующие элементы:

- Регистры процессора.
- Указатель на стек потока/процесса.

2.1 Классическая модель потоков

Процессы используются для группировки взаимосвязанных ресурсов в единое целое, а потоки являются «сущностью», распределяемой для выполнения на центральном процессоре.

Процесс содержит:

- адресное пространство (содержащее текст программы и данные);

- глобальные переменные;

- открытые файлы;

- дочерние процессы;

- необработанные аварийные сигналы;

- сигналы и обработчики сигналов;

- учетную информации.

Поток выполнения имеет:

- счетчик команд, отслеживающий, какую очередную инструкцию нужно выполнять;

- регистры, в которых содержатся текущие рабочие переменные;

- стек с протоколом выполнения, содержащим по одному фрейму для каждой вызванной, но еще не возвратившей управление процедуры

Потоки добавляют к модели процесса возможность реализации нескольких выполняемых задач в единой среде процесса. Потоки иногда называют облегченными (легковесными) процессами.

Многопоточный режим допускает работу нескольких потоков в одном и том же процессе.

Различные потоки в процессе не обладают той независимостью, которая есть у различных процессов. У потоков одно и то же адресное пространство, один поток может считывать данные из стека другого потока, записывать туда свои данные и даже стирать оттуда данные. Защита между потоками отсутствует, в ней нет необходимости.

Различные процессы могут сильно конкурировать друг с другом, а потоки внутри одного процесса служат для совместной работы, активно и тесно сотрудничают друг с другом.

Подобно процессу (с одним потоком), поток может быть в одном из следующих **состояний**:

- выполняемый,
- заблокированный,
- готовый,
- завершенный.

Каждый поток имеет **собственный стек**, который содержит по одному фрейму для каждой уже вызванной, но еще не возвратившей управление процедуры.

1. Применение потоков, моделирование режима многозадачности

Поток – это основной элемент системы, которому ОС выделяет машинное время для параллельного выполнения на процессоре двух и более различных задач.

С появлением понятия потоков в проектирование добавляется возможность использования в ходе параллельных действий единого адресного пространства и всех имеющихся данных процесса. Еще одной отличительной особенностью потоков является лёгкость создания и завершения по сравнению с более сложными процессами.

Потоки позволяют получить довольно большой прирост производительности приложений, выполняющих серьезные вычисления за счет параллельного выполнения различных задач и лёгкости создания. Особенно это актуально для систем, использующих несколько центральных процессоров, где появляется «реальная» возможность параллельных вычислений.

Потоки можно классифицировать следующим способом:

- По отображению в ядро: 1:1, N:M, N:1.
- По многозадачной модели: вытесняющая многозадачность, кооперативная многозадачность.
- По уровню реализации: режим ядра, режим пользователя, гибридная реализация.

Классификация по отображению в ядро.

Модель 1:1 – любой поток, созданный в любом процессе управляется напрямую планировщиков ядра ОС, то есть один пользовательский процесс на один поток ядра.

Модель N:M описывает некоторое число потоков пользовательских процессов N на M потоков режим ядра. К примеру, имеется некая гибридная система, где часть потоков отдается на выполнение в планировщике, а большая их часть в планировщике потоков процесса или библиотеки. Данная модель труднореализуема, но обладает серьезной производительностью.

Модель N:1 – это множество потоков пользовательского процесса, которые отображается на один поток ядра ОС.

Классификация по многозадачной модели

Понятие кооперативной многозадачности обозначает то, что все потоки выполняют свою работу по очереди, с одинаковым отведенным на это временем.

Вытесняющая многозадачность.

Поток с большим приоритетом «вытесняет» поток с меньшим. Современные ОС используют данный подход для выполнения поставленных задач.

Классификация по уровню реализации

Реализацию потоков на уровне ядра можно сравнить с классической моделью 1:1.

Реализация потоков в пользовательском режиме. Такие тяжелые операции, как системный вызов и смена контекста требовали реализацию поддержку в режиме пользователя. Было сделано много попыток реализации, но данная методика не обрела популярности.

Гибридная реализация появилась при попытке совместить преимущества двух предшествующих реализаций, но задумка не удалась: получилось больше недостатков, чем достоинств.

Применение потоков

Допустим мы имеем веб-сайт, куда поступают запросы на определенные веб-страницы, которые затем отправляются обратно пользователям. Как правильно, на главную страницу сайта приходится большинство клиентских запросов. Данные «подборки» называются кэшем, который позволяет повысить производительность за счёт размещения часто используемых запросов на основной странице, то есть в основной памяти.

Один из потоков называется диспетчером, который «читает» входящие запросы от пользователей. Далее он анализирует запрос, подбирает «спящий» (то есть заблокированный) рабочий поток, которому и передает запрос. После этого диспетчер разблокирует данный поток и переводит его в состояние готовности к работе.

После перевода рабочий поток проверяет возможность получения запрашиваемых данных из кэша страниц (доступ к которому имеют все потоки). Если нужная информация не находится, то поток запускает операцию чтения с диска и переводится в «спящий» режим работы до завершения операции.

При блокировке одного потока из-за необходимости поиска данных с диска разблокируется другой рабочий поток и переходит в состояние готовности к работе.

Потоки позволяют использовать концепцию последовательных процессов, которые осуществляют блокирующие системные вызовы и в это же время позволяют «распараллелить» работу. Данные блокирующие системные вызовы позволяют упростить программирование, а работа в параллели даёт большую производительность.

Многозадачность

Многозадачность – это свойство ОС или среды программирования обеспечивать возможность параллельной обработки нескольких процессов. Примитивные многозадачные среды позволяют закрепить за каждой задачей определённый участок памяти и задача выполняется в строго определённые интервалы времени.

Развитые многозадачные системы проводят распределение ресурсов динамически, когда определенная задача занимает память или же освобождает её в зависимости от приоритета и стратегии системы. Такая многозадачная среда обладает данными особенностями:

- Каждая задача имеет свой приоритет, в соответствии с которым для её выполнения выделяется время и память.
- Очередь задач выстраивается по принципу стратегии системы и приоритетов, чтобы каждая задача получила требуемые ресурсы.
- Система обеспечивает защиту памяти от «несанкционированного» вмешательства других задач.
- Система обрабатывает запросы в режиме реального времени.

Режим многозадачности целесообразно применять для рационального использования ЦП.

Выполнение среднестатистического процесса занимает лишь около пятой части времени, пока он находится в памяти, следовательно, при одновременном нахождении пяти процессов в памяти ЦП будет полностью загружен. В данной модели предполагается, что ни один из пяти процессов не может быть остановлен одновременно с другим. Целесообразнее построить модель, которая использует вероятностный подход к занятости центрального процессора.

Допустим, некий процесс проводит часть времени p, которое ему выделил ЦП, в режиме ожидания завершения некоторых операций ввода-вывода. В памяти также присутствуют одновременно n процессов, то вероятность того, что все процессы находятся в режиме ожидания равна p^n.

$$\text{Время использования ЦП} = 1 - p^n$$

На рис. 1 представлена функция, называемая степенью многозадачности, которая представляет из себя зависимость времени использования ЦП от количества процессов, находящихся одновременно в памяти.

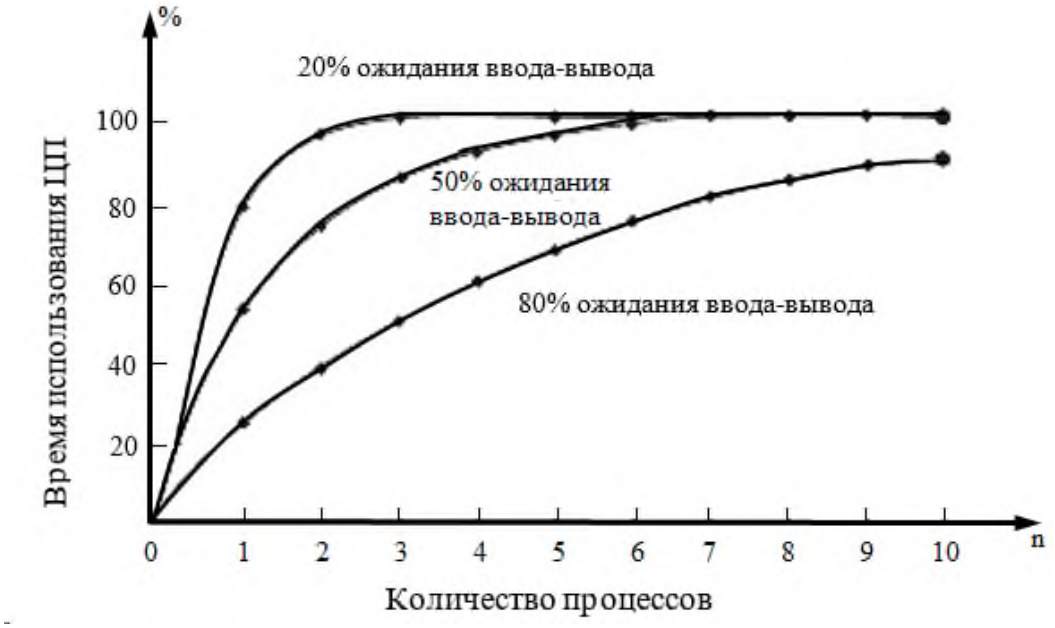

Рис. 1. Зависимость времени использования ЦП от количества процессов, находящихся одновременно в памяти.

Если процесс находится в режиме ожидания 80% своего времени, то для снижения бездействия ЦП до 10% в памяти должны находиться одновременно 10 процессов. В данном случае под режимом ожидания подразумевается не только ожидание ввода-вывода, а также ещё ожидание пользовательского ввода.

Например, память компьютера имеет объем 8 Гбайт, ОС и её таблицы используют около 2 Гбайт памяти, а каждая пользовательская программа также занимает около 2 Гбайт. Значит в памяти компьютера можно одновременно разместить всего три программы пользователя. В данном случае ЦП будет загружен на $1 - 0.8^3 = 49\%$ (учитывая среднее ожидание ввода-вывода равное 80% времени). Если увеличить объем памяти компьютера до 16 Гбайт, то это позволит системе перейти от трёхкратной многозадачности к семикратной, что повысит загрузку ЦП до 79%, то есть дополнительные 8 Гбайт памяти увеличат производительность процессора на 30%.

Первое увеличение объема памяти привносит большой рост производительности процессора для повседневных задач рядового пользователя. Увеличение объема до 24 Гбайт поднимет уровень производительности с 79% до 91% (прирост только 12%).

Прям совсем коротко по теме

Поток – это основной элемент системы, которому ОС выделяет машинное время для параллельного выполнения на процессоре двух и более различных задач, то есть он может выполнять какую-то часть общего кода процесса, в том числе и ту часть, которая в это время уже выполняется другим потоком.

Потоки позволяют использовать концепцию последовательных процессов, которые осуществляют блокирующие системные вызовы и в это же время позволяют «распараллелить» работу.

Многозадачность – это свойство ОС или среды программирования обеспечивать возможность параллельной обработки нескольких процессов. Примитивные многозадачные среды позволяют закрепить за каждой задачей определённый участок памяти и задача выполняется в строго определённые интервалы времени.

2. Потоки в POSIX, реализация потоков в пользовательском пространстве

Потоки POSIX (Pthreads)

В качестве конкретной модели многопоточности рассмотрим потоки POSIX (напомним, что данная аббревиатура расшифровывается как Portable Operating Systems Interface of uniX kind – стандарты для переносимых ОС типа UNIX). Многопоточность в POSIX специфицирована стандартом IEEE 1003.1c, который описывает API для создания и синхронизации потоков. Отметим, что POSIX-стандарт API определяет лишь требуемое поведение библиотеки потоков. Реализация потоков оставляется на усмотрение авторов конкретной POSIX-совместимой библиотеки. POSIX-потоки распространены в ОС типа UNIX, а также поддержаны, с целью совместимости программ, во многих других ОС, например, Solaris и Windows NT.

Стандарт POSIX определяет два основных типа данных для потоков: pthread_t – дескриптор потока ; pthread_attr_t – набор атрибутов потока.

Стандарт POSIX специфицирует следующий набор функций для управления потоками:

- pthread_create(): создание потока
- pthread_exit():завершение потока (должна вызываться функцией потока при завершении)
- pthread_cancel():отмена потока
- pthread_join():заблокировать выполнение потока до прекращения другого потока, указанного в вызове функции
- pthread_detach():освободить ресурсы занимаемые потоком (если поток выполняется, то освобождение ресурсов произойдёт после его завершения)
- pthread_attr_init():инициализировать структуру атрибутов потока
- pthread_attr_setdetachstate():указать системе, что после завершения потока она может автоматически освободить ресурсы, занимаемые потоком
- pthread_attr_destroy():освободить память от структуры атрибутов потока (уничтожить дескриптор).

Имеются следующие примитивы синхронизации POSIX-потоков с помощью мьютексов (mutexes) – аналогов семафоров – и условных переменных (conditional variables) – оба эти типа объектов для синхронизации подробно рассмотрены позже в данном курсе:

- - pthread_mutex_init() – создание мьютекса;
- - pthread_mutex_destroy() – уничтожение мьютекса;
- - pthread_mutex_lock() – закрытие мьютекса;
- - pthread_mutex_trylock() – пробное закрытие мьютекса (если он уже закрыт, вызов игнорируется, и поток не блокируется);
- - pthread_mutex_unlock() – открытие мьютекса;
- - pthread_cond_init() – создание условной переменной;
- - pthread_cond_signal() – разблокировка условной переменной;
- - pthread_cond_wait() – ожидание по условной переменной.

Реализация потоков в пространстве пользователя. Основные положения.

Метод основан на полном размещении всего пакета потоков внутри процесса, при этом ядро не знает о существовании многопоточности. Таким образом, реализовать многопоточность можно даже в однопоточных системах. Потоки должны работать поверх системы поддержки исполнения программ, которые являются набором структур данных и набором процедур, управляющих потоками, например функции creat, exit.

Каждому процессу необходима своя собственная таблица потоков, которая аналогична таблице процессов, но отслеживает только программный счетчик, регистры, состояния и другие характеристики потока. За счет того, что система поддержки исполнения программ реализована внутри процесса, создание потоков и переключение между ними выполняются с помощью небольшого количества команд без переключения в режим ядра. Если поток завершает либо приостанавливает свою работу, функции wait или yield могут сами сохранить информацию о потоке в таблице потоков, а так же вызвать внутренний планировщик для запуска следующего потока. У каждого процесса может быть свой собственный алгоритм планирования.

3. Реализация потоков в ядре, гибридная реализация, активация планировщика

Реализация потоков в ядре

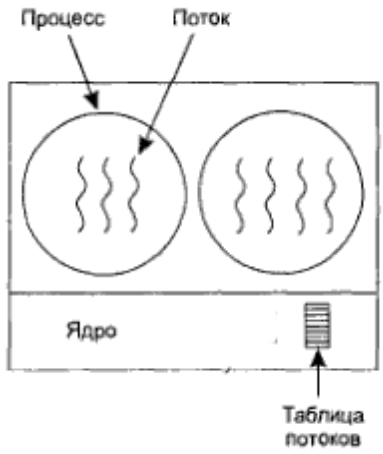

Рисунок 3. Пакет потоков, управляемый ядром.

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.

4. 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.

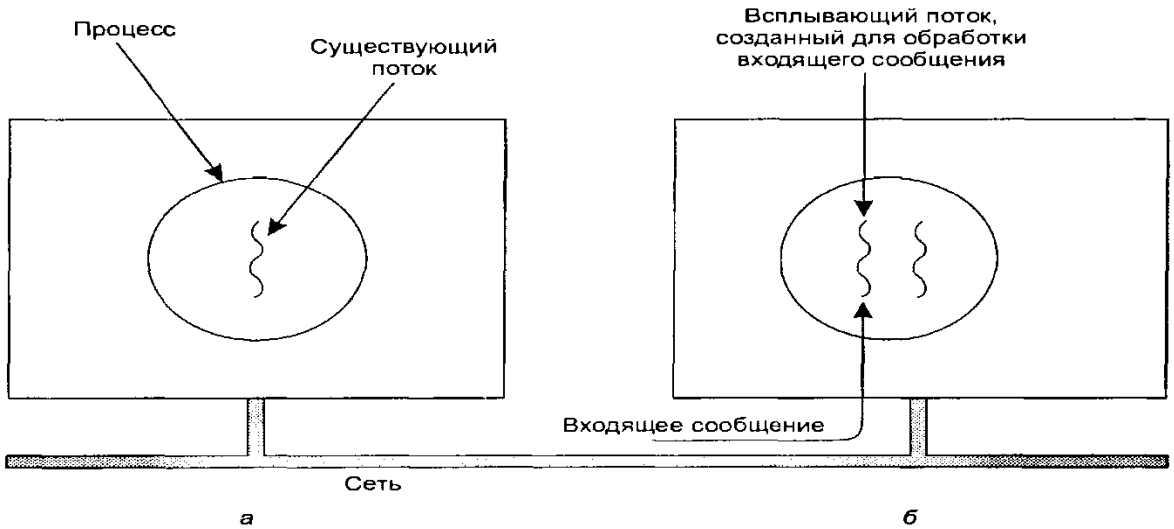

Рис. 1 – Создание нового потока прибытия сообщения: до прибытия(а); после прибытия(б)

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 .

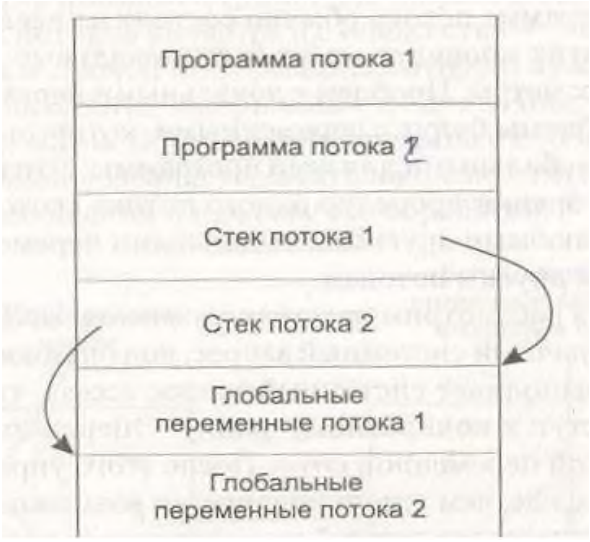

Рис. 3 – Собственные переменные у потоков.

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.

5. Взаимодействие процессов: состязательная ситуация, критические области

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.