ТЕМА 2.2. ПОТОКИ ОПЕРАЦИОННЫХ СИСТЕМ

В данной теме рассматриваются следующие вопросы:

- Применение потоков.
- Классическая модель потоков.
- Параллельное исполнение потоков.
- Главный поток процесса.
- Диаграммы состояния потоков.
- Понятие контекста и переключения контекста.

Лекции – 2 часа, лабораторные занятия – 2 часа, самостоятельная работа – 2 часа.

Экзаменационные вопросы по теме:

- Концепция потока. Параллельное исполнение потоков. Главный поток процесса.
- Диаграммы состояния потоков. Понятие контекста и переключения контекста.

2.2.1. Применение потоков

Поток выполнения (англ. thread — нить) — наименьшая единица обработки, исполнение которой может быть назначено ядром операционной системы. Несколько потоков выполнения могут существовать в рамках одного и того же процесса и совместно использовать ресурсы, такие как память, тогда как процессы не разделяют этих ресурсов. В частности, потоки выполнения разделяют инструкции процесса (его код) и его контекст (значения переменных, которые они имеют в любой момент времени) [1].

В традиционных операционных системах у каждого процесса есть адресное пространство и единственный поток управления. Фактически это почти что определение процесса. Тем не менее нередко возникают ситуации, когда неплохо было бы иметь в одном и том же адресном пространстве несколько потоков управления, выполняемых квазипараллельно. Во многих приложениях одновременно происходит несколько действий, часть которых может периодически быть заблокированной. Модель программирования упрощается за счет разделения такого приложения на несколько последовательных потоков, выполняемых в квазипараллельном режиме, но разделяющих общее адресное пространство процесса (рис. 2.2.1).

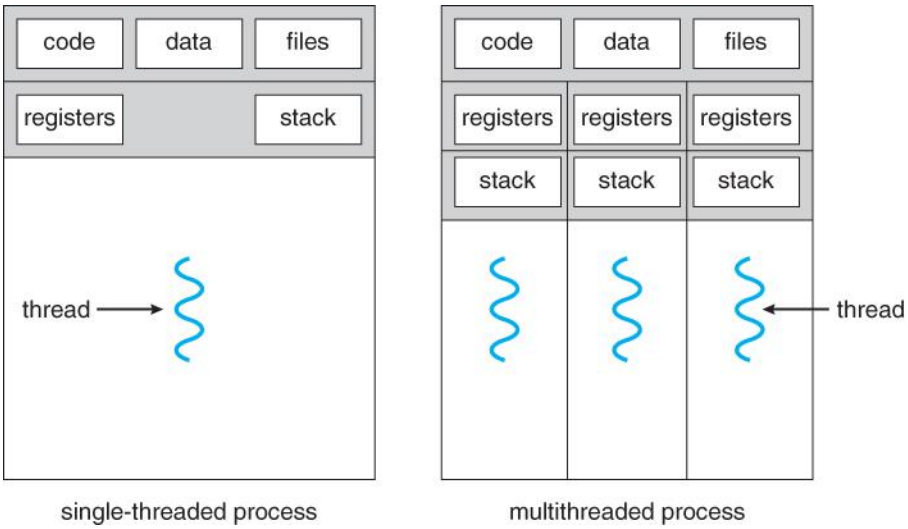

Рис. 2.2.1. Отношение между процессом и потоком

Также к преимуществам потоков относится легкость (то есть быстрота) их создания и ликвидации по сравнению с более «тяжеловесными» процессами. Во многих системах создание потоков осуществляется в 10–100 раз быстрее, чем создание процессов. Это свойство особенно важно, когда требуется быстро и динамично изменять количество потоков.

Третий аргумент в пользу потоков также касается производительности. Когда потоки работают в рамках одного центрального процессора, они не приносят никакого прироста производительности, но, когда выполняются значительные вычисления, а также значительная часть времени тратится на ожидание ввода-вывода, наличие потоков позволяет этим действиям перекрываться по времени, ускоряя работу приложения.

И наконец, потоки весьма полезны для систем, имеющих несколько центральных процессоров, где есть реальная возможность параллельных вычислений.

Наиболее яркие примеры использования потоков – ожидание ввода пользователя, ожидание данных из сети, фоновая проверка орфографии в набираемом документе, выделение синтаксических ошибок в IDE и т. п. Пока один поток отслеживает события клавиатуры и мыши, реагируя на ввод текста и нажатие кнопок, другие потоки с большой скоростью выполняют необходимые вычисления. Например, во время набора этого текста Microsoft Word использовал 32 потока (рис. 2.2.2).

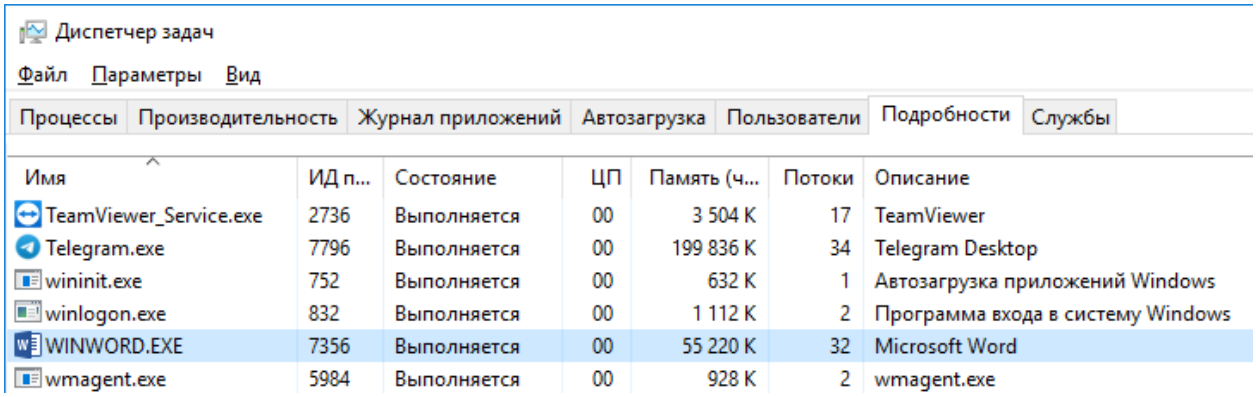

Рис. 2.2.2. Отображение информации о процессах и потоках

Третьим примером, подтверждающим пользу потоков, являются приложения, предназначенные для обработки очень большого объема данных. При обычном подходе блок данных считывается, после чего обрабатывается, а затем снова записывается. При этом процесс блокируется и при поступлении данных, и при их возвращении.

Проблема решается с помощью потоков. Структура процесса может включать входной поток, обрабатывающий поток и выходной поток. Входной поток считывает данные во входной буфер. Обрабатывающий поток извлекает данные из входного буфера, обрабатывает их и помещает результат в выходной буфер. Выходной буфер записывает эти результаты обратно на диск. Таким образом, ввод, вывод и обработка данных могут осуществляться одновременно. Разумеется, эта модель работает лишь при том условии, что системный вызов блокирует только вызывающий поток, а не весь процесс.

Преимущества потоков [1].

- Потоки довольно просто обмениваются данными по сравнению с процессами.
- Создавать потоки для ОС проще и быстрее (иногда на порядок), чем создавать процессы.

Недостатки потоков.

- При программировании приложения с множественными потоками необходимо постоянно думать о потокобезопасности (т. н. thread safety). Приложения, выполняющиеся через множество процессов, не имеют таких требований.
- Один неправильно работающий поток может повредить остальные, так как потоки делят общее адресное пространство. Процессы более изолированы друг от друга.
- Потоки конкурируют друг с другом в адресном пространстве. Стек и thread-local storage, занимая часть виртуального адресного пространства процесса, тем самым делают его недоступным для других потоков. Для встраиваемых (embedded) систем в условиях ограниченности ресурсов такое ограничение может иметь существенное значение.

POSIX Threads

Это стандарт POSIX-реализации потоков. Стандарт POSIX.1c, Threads extensions (IEEE Std 1003.1c-1995) определяет API для управления потоками, их синхронизации и планирования.

Реализации данного API существуют для большого числа UNIX-подобных ОС (GNU/Linux, Solaris, FreeBSD, OpenBSD, NetBSD, OS X), а также для Microsoft Windows и других ОС.

Библиотеки, реализующие этот стандарт (и функции этого стандарта), обычно называются pthreads (функции имеют приставку «pthread_»).

Для Linux известны две реализации Pthreads API:

- LinuxThreads — старая (появилась в 1996 году), оригинальная,

- NPTL (Native POSIX Threads Library) — современная (появилась в ядрах 2.6 в 2003 году), более чётко следует стандарту, сегодня является частью libc.

С точки зрения ядра Linux поток — особый тип процесса, который делит виртуальное адресное пространство и обработчики сигналов с другими процессами.

Для ядра каждый процесс идентифицируется по PID. Для так называемых потоков можно использовать термин TID, но для ядра это одно и то же. Можно понимать это так:

- если процесс однопоточный, то PID будет равен TID этого единственного потока;
- если в процессе работают несколько потоков, то у каждого потока свой TID, а PID идентифицирует группу потоков, которая разделяет адресное пространство, таблицу файловых дескрипторов...

Получается, что функции вида getpid, kill, ... на самом деле работают с идентификаторами групп потоков.

В табл. 2.2.1 перечислены некоторые из вызовов функций пакета Pthread.

Таблица 2.2.1. Ряд вызовов функций стандарта Pthreads

Вызовы, связанные с потоком	Описание
pthread_create	Создание нового потока
pthread_exit	Завершение работы вызвавшего потока
pthread_join	Ожидание выхода из указанного потока
pthread_yield	Освобождение центрального процессора, позволяющее выполняться другому потоку
pthread_attr_init	Создание и инициализация структуры атрибутов потока
pthread_attr_destroy	Удаление структуры атрибутов потока

Новый поток создается с помощью вызова функции **`pthread_create`**. В качестве значения функции возвращается идентификатор только что созданного потока. Этот вызов намеренно сделан очень похожим на системный вызов **`fork`** (за исключением параметров), а идентификатор потока играет роль PID, главным образом для идентификации ссылок на потоки в других вызовах.

Когда поток заканчивает возложенную на него работу, он может быть завершен путем вызова функции **`pthread_exit`**. Этот вызов останавливает поток и освобождает пространство, занятое его стеком.

Зачастую потоку необходимо перед продолжением выполнения ожидать окончания работы и выхода из другого потока. Ожидающий поток вызывает функцию **`pthread_join`**, чтобы ждать завершения другого указанного потока. В качестве параметра этой функции передается идентификатор потока, чьего завершения следует ожидать.

Иногда бывает так, что поток не является логически заблокированным, но считает, что проработал достаточно долго, и намеревается дать шанс на выполнение другому потоку. Этой цели он может добиться за счет вызова функции **`pthread_yield`**. Для процессов подобных вызовов функций не существует, поскольку предполагается, что процессы сильно конкурируют друг с другом и каждый из них требует как можно больше времени центрального процессора. Но поскольку потоки одного процесса, как правило, пишутся одним и тем же программистом, то он добивается от них, чтобы они давали друг другу шанс на выполнение.

Два следующих вызова функций, связанных с потоками, относятся к атрибутам. Функция **`pthread_attr_init`** создает структуру атрибутов, связанную с потоком, и инициализирует ее значениями по умолчанию. Эти значения (например, приоритет) могут быть изменены за счет работы с полями в структуре атрибутов.

И наконец, функция **pthread_attr_destroy** удаляет структуру атрибутов, принадлежащую потоку, освобождая память, которую она занимала. На поток, который использовал данную структуру, это не влияет, и он продолжает свое существование.

Потоки и процессы в Windows

В среде Microsoft Windows концепция иная, там процесс — это контейнер для потоков. Процесс-контейнер содержит как минимум один поток. Если потоков в процессе несколько, приложение (процесс) становится многопоточным.

Процесс — это исполнение программы. Операционная система использует процессы для разделения исполняемых приложений. Поток — это основная единица, которой операционная система выделяет время процессора. Каждый поток имеет приоритет планирования и набор структур, в которых система сохраняет контекст потока, когда выполнение потока приостановлено. Контекст потока содержит все сведения, позволяющие потоку безболезненно возобновить выполнение, в том числе набор регистров процессора и стек потока. Несколько потоков могут выполняться в контексте процесса. Все потоки процесса используют общий диапазон виртуальных адресов. Поток может исполнять любую часть программного кода, включая части, выполняемые в данный момент другим потоком [2].

> *Примечание. Платформа .NET Framework предоставляет способ изоляции приложений в процессе с помощью доменов приложений. (Домены приложений недоступны в .NET Core.)*

По умолчанию программа .NET запускается с одним потоком, часто называемым основным потоком. Тем не менее она может создавать дополнительные потоки для выполнения кода параллельно или одновременно с основным потоком. Эти потоки часто называются рабочими потоками.

Многопоточность в .NET

Начиная с .NET Framework 4, для многопоточности рекомендуется использовать библиотеку параллельных задач (TPL) и Parallel LINQ (PLINQ) [2].

Библиотека параллельных задач и PLINQ полагаются на потоки **ThreadPool**. Класс **System.Threading.ThreadPool** предоставляет приложения .NET с пулом рабочих потоков. Также можно использовать потоки из пула потоков.

Наконец, можно использовать класс **System.Threading.Thread**, который представляет управляемый поток.

Несколько потоков могут требовать доступ к общему ресурсу. Чтобы сохранить ресурс в непроверенном состоянии и избежать условий гонки, необходимо синхронизировать доступ к потоку к нему. Вы также можете координировать взаимодействие нескольких потоков. Платформа .NET предоставляет ряд типов для синхронизации доступа к общему ресурсу или координации взаимодействия потоков.

Исключения следует обрабатывать в потоках. Необработанные исключения в потоках, как правило, приводят к завершению процесса.

2.2.2. Классическая модель потоков

Модель процесса основана на двух независимых понятиях: группировке ресурсов и выполнении [3]. Иногда их полезно отделить друг от друга, и тут на первый план выходят потоки. Сначала будет рассмотрена классическая модель потоков, затем изучена модель потоков, используемая в Linux, которая размывает грань между процессами и потоками.

Согласно одному из взглядов на процесс, он является способом группировки в единое целое взаимосвязанных ресурсов. У процесса есть адресное пространство, содержащее текст программы и данные, а также другие ресурсы. Эти ресурсы могут включать

открытые файлы, необработанные аварийные сигналы, обработчики сигналов, учетную информацию и т. д. Управление этими ресурсами можно значительно облегчить, если собрать их воедино в виде процесса.

Другое присущее процессу понятие — поток выполнения — обычно сокращается до слова поток. У потока есть счетчик команд, отслеживающий, какую очередную инструкцию нужно выполнять. У него есть регистры, в которых содержатся текущие рабочие переменные. У него есть стек с протоколом выполнения, содержащим по одному фрейму для каждой вызванной, но еще не возвратившей управление процедуры. Хотя поток может быть выполнен в рамках какого-нибудь процесса, сам поток и его процесс являются разными понятиями и должны рассматриваться по отдельности. Процессы используются для группировки ресурсов в единое образование, а потоки являются «сущностью», распределяемой для выполнения на центральном процессоре.

Потоки добавляют к модели процесса возможность реализации нескольких в значительной степени независимых друг от друга выполняемых задач в единой среде процесса. Наличие нескольких потоков, выполняемых параллельно в рамках одного процесса, является аналогией наличия нескольких процессов, выполняемых параллельно на одном компьютере. В первом случае потоки используют единое адресное пространство и другие ресурсы. А в последнем случае процессы используют общую физическую память, диски, принтеры и другие ресурсы. Поскольку потоки обладают некоторыми свойствами процессов, их иногда называют облегченными процессами. Термин «многопоточный режим» также используется для описания ситуации, при которой допускается работа нескольких потоков в одном и том же процессе.

На рис. 2.2.3.а показаны три традиционных процесса. У каждого из них имеется собственное адресное пространство и единственный поток управления. В отличие от этого, на рис. 2.2.3.б показан один процесс, имеющий три потока управления. Хотя в обоих случаях у нас имеется три потока, на рис. 2.2.3.а каждый из них работает в собственном адресном пространстве, а на рис. 2.2.3.б все три потока используют общее адресное пространство.

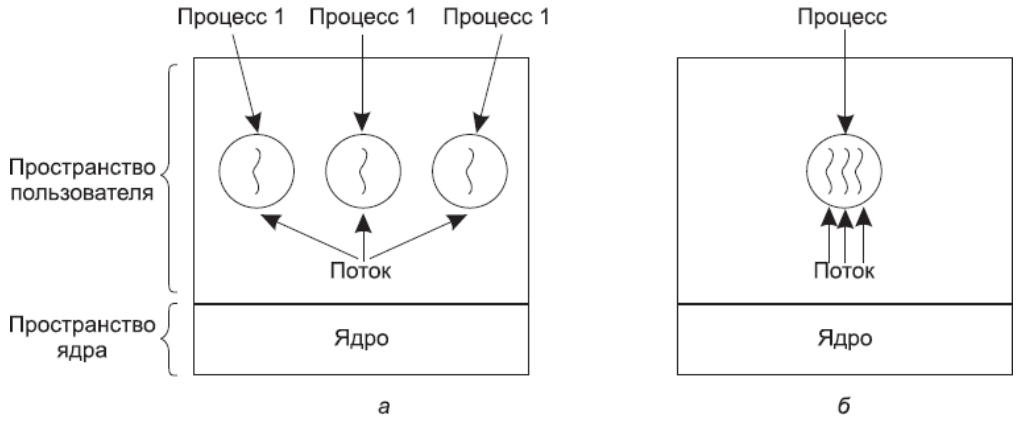

Рис. 2.2.3. а — три процесса, у каждого из которых по одному потоку; б — один процесс с тремя потоками

Когда многопоточный процесс выполняется на однопроцессорной системе, потоки выполняются, сменяя друг друга. За счет переключения между несколькими процессами система создает иллюзию параллельно работающих отдельных последовательных процессов. Многопоточный режим осуществляется аналогичным способом. Центральный процессор быстро переключается между потоками, создавая иллюзию, что потоки выполняются параллельно, пусть даже на более медленном центральном процессоре, чем реально используемый. При наличии в одном процессе трех потоков, ограниченных по скорости вычисления, будет казаться, что потоки выполняются параллельно и каждый

из них выполняется на центральном процессоре, имеющем скорость, которая составляет одну треть от скорости реального процессора.

Различные потоки в процессе не обладают той независимостью, которая есть у различных процессов. У всех потоков абсолютно одно и то же адресное пространство, а значит, они так же совместно используют одни и те же глобальные переменные. Поскольку каждый поток может иметь доступ к любому адресу памяти в пределах адресного пространства процесса, один поток может считывать данные из стека другого потока, записывать туда свои данные и даже стирать оттуда данные. Защита между потоками отсутствует, потому что ее невозможно осуществить и в ней нет необходимости. В отличие от различных процессов, которые могут принадлежать различным пользователям и которые могут враждовать друг с другом, один процесс всегда принадлежит одному и тому же пользователю, который, по-видимому, и создал несколько потоков для их совместной работы, а не для вражды. В дополнение к использованию общего адресного пространства все потоки, как показано в табл. 2.2.2, могут совместно использовать одни и те же открытые файлы, дочерние процессы, ожидаемые и обычные сигналы и т. п. Поэтому структура, показанная на рис. 2.2.3.а, может использоваться, когда все три процесса фактически не зависят друг от друга, а структура, показанная на рис. 2.2.3.б, может применяться, когда три потока фактически являются частью одного и того же задания и активно и тесно сотрудничают друг с другом.

Таблица 2.2.2. Использование объектов потоками

Элементы, присущие каждому процессу	Элементы, присущие каждому потоку
Адресное пространство	Счетчик команд
Глобальные переменные	Регистры
Открытые файлы	Стек
Дочерние процессы	Состояние
Необработанные аварийные сигналы	
Сигналы и обработчики сигналов	
Учетная информация	

Элементы в первом столбце относятся к свойствам процесса, а не потоков. Например, если один из потоков открывает файл, этот файл становится видимым в других потоках, принадлежащих процессу, и они могут производить с этим файлом операции чтения-записи. Это вполне логично, поскольку именно процесс, а не поток является элементом управления ресурсами. Если бы у каждого потока были собственные адресное пространство, открытые файлы, необработанные аварийные сигналы и т. д., то он был бы отдельным процессом. С помощью потоков мы пытаемся достичь возможности выполнения нескольких потоков, использующих набор общих ресурсов с целью тесного сотрудничества при реализации какой-нибудь задачи.

Подобно традиционному процессу (то есть процессу только с одним потоком), поток должен быть в одном из следующих состояний: выполняемый, заблокированный, готовый или завершенный. Выполняемый поток занимает центральный процессор и является активным в данный момент. В отличие от этого, заблокированный поток ожидает события, которое его разблокирует. Например, когда поток выполняет системный вызов для чтения с клавиатуры, он блокируется до тех пор, пока на ней не будет что-нибудь набрано. Поток может быть заблокирован в ожидании какого-то внешнего события или его разблокировки другим потоком. Готовый поток планируется к выполнению и будет выполнен, как только подойдет его очередь. Переходы между состояниями потока аналогичны переходам между состояниями процесса.

Следует учесть, что каждый поток имеет собственный стек (рис. 2.2.4). Стек каждого потока содержит по одному фрейму для каждой уже вызванной, но еще не возвратившей управление процедуры. Такой фрейм содержит локальные переменные процедуры и адрес возврата управления по завершении ее вызова. Например, если процедура X вызывает процедуру Y, а Y вызывает процедуру Z, то при выполнении Z в стеке будут фреймы для X, Y и Z. Каждый поток будет, как правило, вызывать различные процедуры и, следовательно, иметь среду выполнения, отличающуюся от среды выполнения других потоков. Поэтому каждому потоку нужен собственный стек.

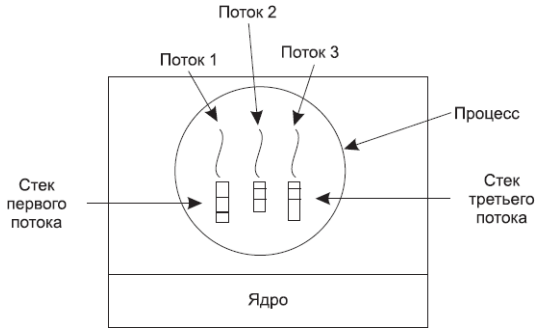

Рис. 2.2.4. У каждого потока имеется собственный стек

Когда используется многопоточность, процесс обычно начинается с использования одного потока. Этот поток может создавать новые потоки, вызвав библиотечную процедуру, к примеру, **`thread_create`**. В параметре **`thread_create`** обычно указывается имя процедуры, запускаемой в новом потоке. Нет необходимости (или даже возможности) указывать для нового потока какое-нибудь адресное пространство, поскольку он автоматически запускается в адресном пространстве создающего потока. Иногда потоки имеют иерархическую структуру, при которой у них устанавливаются взаимоотношения между родительскими и дочерними потоками, но чаще всего такие взаимоотношения отсутствуют и все потоки считаются равнозначными. Независимо от наличия или отсутствия иерархических взаимоотношений создающий поток обычно возвращает идентификатор потока, который дает имя новому потоку.

Когда поток завершает свою работу, выход из него может быть осуществлен за счет вызова библиотечной процедуры, к примеру, **`thread_exit`**. После этого он прекращает свое существование и больше не фигурирует в работе планировщика. В некоторых использующих потоки системах какой-нибудь поток для выполнения выхода может ожидать выхода из какого-нибудь другого (указанного) потока после вызова процедуры, к примеру, **`thread_join`**. Эта процедура блокирует вызывающий поток до тех пор, пока не будет осуществлен выход из другого (указанного) потока. В этом отношении создание и завершение работы потока очень похожи на создание и завершение работы процесса при использовании примерно одних и тех же параметров.

Другой распространенной процедурой, вызываемой потоком, является **`thread_yield`**. Она позволяет потоку добровольно уступить центральный процессор для выполнения другого потока. Важность вызова такой процедуры обусловливается отсутствием прерывания по таймеру, которое есть у процессов и благодаря которому фактически задается режим многозадачности. Для потоков важно проявлять вежливость и время от времени добровольно уступать центральный процессор, чтобы дать возможность выполнения другим потокам. Другие вызываемые процедуры позволяют одному потоку ожидать, пока другой поток не завершит какую-нибудь работу, а этому потоку — оповестить о том, что он завершил определенную работу, и т. д.

Потоки могут быть реализованы на уровне пользовательского пространства и на уровне ядра. При первом подходе (на уровне пользовательского пространства) процесс ведет собственную таблицу потоков, а ядро считает этот процесс однопоточным. При этом

переключение потоков требует меньше затрат процессорного времени. Но возникают проблемы планирования (если поток добровольно не возвращает управление, другие не получат шанса на выполнение), системные вызовы или отсутствие страницы в памяти могут заблокировать весь процесс. Для потоков, реализованных на уровне ядра, не требуется никаких новых, неблокирующих системных вызовов. Более того, если один из выполняемых потоков столкнется с ошибкой обращения к отсутствующей странице, ядро может с легкостью проверить наличие у процесса любых других готовых к выполнению потоков и при наличии таковых запустить один из них на выполнение, пока будет длиться ожидание извлечения запрошенной страницы с диска. Главный недостаток этих потоков состоит в весьма существенных затратах времени на системный вызов, поэтому, если операции над потоками (создание, удаление и т. п.) выполняются довольно часто, это влечет за собой более существенные издержки. Но и в этом случае могут быть проблемы, например, неопределенность при разветвлении многопоточного процесса и неопределенность при обработке сигналов.

2.2.3. Параллельное исполнение потоков

Под словами "параллельный поток" или просто "поток" мы будем понимать объект, выполняемый параллельно с основным потоком приложения и с другими параллельными потоками [4]. Термин "параллельность" рассматривается в контексте вытесняющей многозадачности операционной системы - то есть, операционная система выделяет потоку некоторый квант времени, а затем переключается на другой поток. Параллельный поток может добровольно отдавать часть своего кванта времени, если он переходит в состоянии ожидания некоторого события. Выполнение потока может быть прервано более приоритетным потоком. Момент переключения параллельных потоков и адрес процессорной инструкции в момент переключения будем считать полностью недетеминированными. Реализация параллельных потоков опирается на возможность, предоставляемую операционной системой — "thread". Для именования параллельных потоков в литературе встречаются и другие термины: активные объекты, нити, параллельные процессы.

Модель конкурентного выполнения сопрограмм (concurrency) - это немного более широкий термин, чем параллелизм (parallelism). Данная модель предполагает, что несколько задач могут выполняться одновременно, но не говорит, как это должно быть достигнуто. В англоязычном мире есть поговорка, “Concurrency does not imply parallelism”. Асинхронный ввод-вывод, не являющийся ни многопроцессным, ни многопоточным, тем не менее также подпадает под формулировку конкурентного выполнения кода. Резюмируя вышесказанное, многопроцессность это форма параллелизма (которого также можно добиться многопоточностью), а параллелизм в свою очередь, это подмножество конкурентного выполнения сопрограмм.

Среди множества различных видов взаимодействия параллельных потоков обзорно рассмотрим только несколько наиболее важных видов: совместное использование разделяемых данных, асинхронное взаимодействие, синхронное взаимодействие. Более подробное рассмотрение будет приведено при изучении конкретных механизмов [4].

Разделяемые данные

Задача, которую нужно решить при совместном использовании данных заключается в обеспечении атомарности (неделимости) изменения данных, что иногда обозначается термином "транзакция". Иными словами, необходим взаимно-исключающий доступ к данным - недопустимо, чтобы один поток читал данные, в то время как другой поток их изменял. Наиболее простое и эффективное средство - это использование критической секции.

Блок **try-finally** позволяет разблокировать критическую секцию даже при возникновении исключительной ситуации. Опускать **try-finally** допустимо только при очень простом коде, в котором ошибка исключена. У критической секции есть одно очень важное достоинство - высокая эффективность, поскольку критическая секция - это не объект ядра операционной системы, а запись (record) в адресном пространстве приложения, именно поэтому критическая секция решает задачу синхронизации потоков только в рамках одного приложения. Альтернативное и более универсальное средство - мьютекс, более чем на порядок уступает критической секции по эффективности. В качестве недостатков критической секции можно отметить следующие:

- при программировании требуется очень внимательно относиться к обязательной парности операций Lock-Unlock, ошибка может привести к зависанию приложения;
- действия в критической должны быть максимально короткими, у приостановленных потоков нет возможности завершиться до тех пор, пока поток, захвативший критическую секцию, будет оставаться в ней;
- критическая секция не делает различия между потоками, изменяющими данные и потоками, которые их только читают - более эффективно было бы разрешить многим потокам одновременно читать данные, но блокировать данные при их изменении.

Последний недостаток можно исправить, если применить более сложную технику, основанную не на критических секциях, а на событиях.

Задача реализации взаимоисключающего доступа всегда сталкивается с принципиальной проблемой взаимной блокировки при одновременном захвате более чем одного ресурса. Предположим, что существуют два потока A и B, а также два совместно используемых ресурса R1 и R2. Далее предположим такой сценарий: поток A захватывает ресурс R1, а затем ресурс R2 (не освобождая при этом ресурса R1). Поток B захватывает ресурс R2, а затем ресурс R1. Если после того как поток A захватил R1, поток B захватит ресурс R2, то оба потока попадут в состояние дедлока (взаимоблокировки) из которого никогда не выйдут. Обнаружение и предотвращение дедлока в общем случае весьма сложная задача, поэтому требуется очень аккуратно программировать ситуацию захвата нескольких ресурсов: либо отказаться от множественного захвата, введя поток-посредник, либо захватывать ресурсы всегда только в одном и том же порядке.

Несколько уменьшить трудности, связанные с критическими секциями, можно, если инкапсулировать разделяемые данные в специально разработанных классах.

Асинхронное взаимодействие

Это очень универсальный и весьма эффективный способ взаимодействия параллельных потоков. Суть его состоит во введении посредника-буфера между параллельными потоками. Иногда этот способ называют взаимодействием с помощью обмена сообщениями. Поток-писатель, желающий послать другому потоку некоторые данные, записывает их в очередь. Поток-читатель ожидает появления данных в очереди, получает их, обрабатывает и, если нужно, посылает результат в очередь потока-писателя. Принципиальный недостаток асинхронного взаимодействия - необходимость буфера неопределенного размера. Этот недостаток можно сгладить, если ввести ограничение на размер очереди - в этом случае поток-писатель должен приостанавливаться, если очередь достигла своего максимального размера.

Важная проблема эффективности многопоточной обработки возникает в том случае, если поток должен обрабатывать несколько очередей входящих сообщений или несколько событий. Единственный способ достигнуть этого - ожидать именно первого из многих возможных событий.

Синхронное взаимодействие

Синхронное взаимодействие будем рассматривать только применительно к двум параллельным потокам. В конце раздела будет особо отмечен случай множественной синхронизации. Ограничимся только простым случаем парного синхронного взаимодействия - асимметричное рандеву. Суть этого способа состоит в том, что оба взаимодействующих потока подходят к точке синхронизации, обмениваются данными и затем продолжают работать параллельно и независимо. Если один из потоков подошел к точке синхронизации раньше, то он дожидается партнера. Асимметричность выражается в том, что потоки играют при встрече разные роли. Поток-отправитель несет основную, активную, нагрузку, а поток-получатель более пассивен. Рандеву фактически означает совмещение синхронизации и обмена данными. Предлагаемая реализация рандеву представляет собой объект-канал, по которому происходит взаимодействие. Обмен данными в этой реализации состоит в том, что поток-отправитель вызывает у потока-получателя некоторый метод, передавая данные как аргумент и принимая реакцию как результат.

Обычно канал принадлежит потоку-получателю; поток-отправитель начинает взаимодействие, вызывая для канала метод Send и передавая себя как аргумент. Кроме того, функции Send передается еще один аргумент - адрес метода потока-отправителя, который будет вызван при успешном рандеву. При готовности к рандеву поток-получатель вызывает метод Receive, передавая себя как аргумент. Каждый из взаимодействующих потоков может ограничить время своего ожидания, указав таймаут. Обсудим возможные варианты взаимного поведения потоков:

Отправитель вызвал функцию Send раньше, чем получатель вызвал Receive - в этом случае отправитель приостановится до момента, когда получатель вызовет Receive. В функции Receive получатель сохраняет в канале ссылку на себя, сбрасывает свое ожидающее сообщения, устанавливает ожидающее сообщение отправителя и переходит в состояние ожидания. Отправитель выходит из состояния ожидания и вызывает свой метод-обработчик, передавая ему ссылку на получателя. В этом методе происходит собственно обмен данными. После этого отправитель устанавливает флаг успешного рандеву и активизирует событие получателя. Получатель приводит события к начальному состоянию и завершает взаимодействие. Обе функции вернут True при успешном рандеву и False при неуспешном. Все действия по изменению состояния канала защищены критической секцией.

Противоположная ситуация: получатель вызвал функцию Receive раньше, чем отправитель вызвал Send — в этом случае получатель подготавливает все события и переходит в состояние ожидания. Когда отправитель вызывает свой метод Send, то ожидающее событие уже установлено и отправитель без реального ожидания сразу начинает обмен данными. Далее все действия происходят, как и в первом варианте.

Отправитель начинает взаимодействие первым, но завершается по таймауту или завершающему событию потока. В этом случае состояние канала не изменяется.

Получатель начинает взаимодействие первым, но завершается по таймауту или завершающему событию потока. В этом случае ожидающее событие отправителя сначала устанавливается, а затем сбрасывается, то есть, состояние канала остается неизменным.

Ситуация аналогичная предыдущей, но отправитель начинает взаимодействие после того, как получатель вышел из состояния ожидания с ошибкой таймаута или завершения, но еще не начал завершающих действий. В этом случае рандеву будет успешным, так как отправитель сумеет выполнить всю обычную работу по взаимодействию. Результат функции ожидания получателя не учитывается.

В заключение отметим случай множественной синхронизации параллельных потоков. Иногда множественная синхронизация обозначается термином "барьер". Суть

синхронизации состоит в том, что параллельные потоки приходят к точке синхронизации в разное время, а выходят из нее в одно и тоже время. Реализуется множественная синхронизация Windows-функцией **`WaitForMultipleObjects`**. Третий аргумент этой функции в данном случае должен быть True, то есть, функция будет ожидать прихода всех указанных событий. Обычно множественная синхронизация не связана с взаимодействием параллельных потоков, а используется для их привязки по времени.

2.2.4. Главный поток процесса

По умолчанию процесс создается с одним потоком, называемым главным или основным потоком. Потоки в Unix по существу являются дешевой копией процессов и по аналогии с процессами предоставляют механизм для одновременного выполнения нескольких параллельных задач в рамках одного приложения [5].

Техника программирования, позволяющая коду выполняться внутри единого процесса с помощью запуска нескольких потоков называется многопоточностью. Потоки могут выполняться как одновременно, так и нет. Одновременное выполнение потоков одного процесса называется параллелизмом (parallelism). Параллельное выполнение потоков в рамках одного процесса возможно только в многоядерных системах и не является обязательным поведением. В одноядерных системах многопоточность может быть только последовательной. Переводя на программистский язык: многопоточный код не обязан быть по определению быстрым или параллельным, но быть таковым он может.

Все потоки, выполняясь внутри своего процесса, разделяют общую глобальную память — данные и сегменты кучи. Это упрощает обмен данным между потоками процесса. Для этого всего лишь нужно скопировать данные в общие переменные (глобальные или в куче). Если поток изменяет ресурс процесса, это изменение сразу же становится видно другим потокам этого процесса. Тем не менее, каждый поток обладает локальным стеком для хранение локальных данных, которые доступны в рамках только текущего потока.

В Linux потоки реализованы с помощью системного вызова **`clone`**(), который как минимум в 10 раз меньше занимает времени для создания еще одного потока, чем создание еще одного процесса при помощи **`fork`**(). Такая скорость достигается за счет того, что многие атрибуты процесса разделяются между потоками.

В Windows процесс может породить практически неограниченное количество потоков. Процесс будет активен, пока активен хотя бы один поток. Но при этом следует различать фоновые и обычные, не фоновые потоки. В управляемых потоках (класс Thread в .NET) для этого служит свойство **`IsBackground`**. В чистом Win32 API все потоки создаются как обычные, не фоновые. Чтобы стать фоновым, поток должен выполнить вызов функции **`SetThreadPriority`** с аргументом `THREAD_MODE_BACKGROUND_BEGIN`. Соответственно, для выхода из фонового режима поток должен выполнить вызов функции **`SetThreadPriority`** с аргументом `THREAD_MODE_BACKGROUND_END`.

2.2.5. Диаграммы состояния потоков

Для каждого созданного потока в системе предусматриваются три возможных его состояния:

- состояние **выполнения**, когда код потока выполняется процессором; на однопроцессорных платформах в этом состоянии в каждый момент времени может находиться только один поток;
- состояние **готовности к выполнению**, когда поток готов продолжать свою работу и ждет освобождения ЦП;
- состояние **ожидания наступления некоторого события**; в этом случае поток не претендует на время ЦП, пока не наступит определенное событие (завершение

операции ввода/вывода, освобождение необходимого потоку занятого ресурса, сигнала от другого потока); часто такие потоки называют блокированными.

Изменение состояния потока происходит в результате соответствующих действий. Удобно для этих целей использовать следующую диаграмму состояний и переходов (рис. 2.2.5).

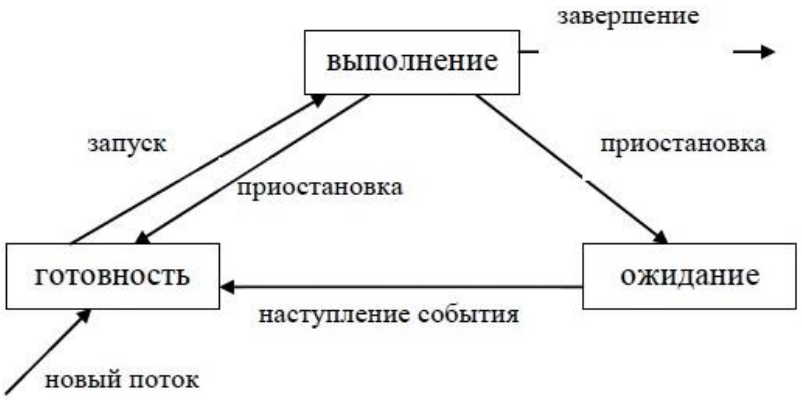

Рис. 2.2.5. Диаграмма состояний потока

Переходы между состояниями можно описать следующим образом:

- «**готовность**» → «**выполнение**»: система в соответствии с алгоритмом планирования выбирает для выполнения текущий поток, предоставляя ему ЦП
- «**выполнение**» → «**готовность**»: поток готов продолжать свою работу, но система принимает решение прервать его выполнение; чаще всего это происходит по следующим двум причинам:
    - завершается выделенное потоку время владения процессором;
    - в числе готовых к выполнению появляется более приоритетный поток по сравнению с текущим;
- «**выполнение**» → «**ожидание**»: дальнейшее исполнение кода текущего активного потока невозможно без наступления некоторого события, и поэтому активный поток прерывает свое выполнение и переводится системой в состояние ожидания (блокируется);
- «**ожидание**» → «**готовность**»: в системе происходит некоторое событие, наступление которого ожидает один из блокированных потоков, и поэтому система переводит этот поток в состояние готовности (разблокирует), после чего он будет учитываться системой при планировании порядка предоставления ЦП;
- наконец, поток может нормально или аварийно завершить свое выполнение, после чего система удаляет его дескриптор из своей внутренней структуры, и тем самым поток перестает существовать.

В состояниях готовности и ожидания может находиться несколько потоков, поэтому система создает для хранения их дескрипторов отдельные списковые структуры. Организация этих списков зависит от тех принципов, которые положены в основу планирования потоков для данной ОС.

2.2.6. Понятие контекста и переключения контекста

Переключение контекста (context switch) — в многозадачных ОС и средах — процесс прекращения выполнения процессором одной задачи (процесса, потока, нити) с сохранением всей необходимой информации и состояния, необходимых для последующего продолжения с прерванного места, и восстановления и загрузки состояния задачи, к выполнению которой переходит процессор.

В процедуру переключения контекста входит так называемое планирование задачи — процесс принятия решения, какой задаче передать управление.

При переключении контекста происходит сохранение и восстановление следующей информации:

- Регистровый контекст регистров общего назначения (в том числе флаговый регистр)
- Контекст состояния сопроцессора с плавающей точкой / регистров MMX (x86)
- Состояние регистров SSE, AVX (x86)
- Состояние сегментных регистров (x86)
- Состояние некоторых управляющих регистров (например, регистр CR3, отвечающий за страничное отображение памяти процесса) (x86)

В ядре ОС с каждым потоком связаны следующие структуры:

- Общая информация pid, tid, uid, gid, euid, egid и т. д.
- Состояние процесса/потока
- Права доступа
- Используемые потоком ресурсы и блокировки
- Счетчики использования ресурсов (например, таймеры использованного процессорного времени)
- Регионы памяти, выделенные процессу

Кроме того, что очень важно, при переключении контекста происходят следующие аппаратные действия, влияющие на производительность:

- Происходит очистка конвейера команд и данных процессора
- Очищается TLB, отвечающий за страничное отображение линейных адресов на физические.

Кроме того, следует учесть следующие факты, влияющие на состояние системы:

- Содержимое кэша (особенно это касается кэша первого уровня), накопленное и «оптимизированное» под выполнение одного потока, оказывается совершенно неприменимым к новому потоку, на который происходит переключение.
- При переключении контекста на процесс, который до этого долгое время не использовался, многие страницы могут физически отсутствовать в оперативной памяти, что порождает подкачку вытесненных страниц из вторичной памяти.

С точки зрения прикладного уровня переключение контекста можно разделить на добровольное (voluntary) и принудительное (non-voluntary): выполняющийся процесс/поток может сам передать управление другому потоку либо ядро может насильно отобрать у него управление.

- **Ядро ОС может отобрать управление у выполняющегося процесса/потока при истечении кванта времени, выделенного на выполнение**. С точки зрения программиста это означает, что управление могло уйти от потока в «самый неподходящий» момент времени, когда структуры данных могут находиться в противоречивом состоянии из-за того, что их изменение не было завершено.
- **Выполнение блокирующего системного вызова**. Когда приложение производит ввод-вывод, ядро может решить, что можно отдать управление другому потоку/процессу в ожидании, пока запрошенный данным потоком дисковый либо сетевой ввод-вывод будет выполнен. Данный вариант является самым производительным.
- **Синхронизирующие примитивы ядра**. Мьютексы, семафоры и т. д. Это и есть основной источник проблем с производительностью. Недостаточно продуманная работа с синхронизирующими примитивами может приводить к десяткам тысяч, а в особо запущенных случаях — и к сотням тысяч переключений контекста в секунду.
- **Системный вызов, явно ожидающий наступления события (select, poll, epoll, pause, wait, …) либо момента времени (sleep, nanosleep, …)**. Данный

вариант является относительно производительным, так как ядро ОС имеет информацию об ожидающих процессах.

Разницу между операционными системами реального времени и разделения времени особенно хорошо видно в различии логики планирования при переключении контекста: Планировщик систем разделения времени старается максимизировать производительность всей системы в целом, возможно, в ущерб производительности отдельных процессов. Задача планировщика систем реального времени — обеспечить приоритетное выполнение отдельных критических процессов, причем не важно, насколько жесткими накладными расходами для всей остальной системы в целом это обойдется.

Как видно из вышесказанного, переключение контекста является очень ресурсоёмкой операцией, причем, чем более функциональным является процессор, тем более ресурсоёмкой эта операция становится. Исходя из этого, ядро использует ряд стратегий, чтобы, во-первых, сократить количество переключений контекста, а во-вторых, сделать переключение контекста менее ресурсоёмким.

Методы уменьшения количества переключений контекста:

- Существует возможность конфигурирования выделяемого потоку кванта процессорного времени. При сборке ядра Linux возможно указать Server/Desktop/Low-Latency Desktop. Для серверных конфигураций этот квант больше.

Методы снижения ресурсоемкости переключения контекста:

- При переключении контекста между потоками, разделяющими одно адресное пространство в пределах одного процесса, ядро не трогает регистр CR3, тем самым сохраняя TLB
- Во многих случаях ядро располагается в том же адресном пространстве, что и пользовательский процесс. При переключении контекста между user-space и kernel-space (и обратно), что, например, происходит при выполнении системных вызовов, ядро не трогает регистр CR3, тем самым сохраняя TLB
- Производя планирование, ядро старается минимизировать перемещение процесса между вычислительными ядрами в SMP-системе, тем самым улучшая эффективность работы кэша второго уровня.
- Реальное сохранение/восстановление контекста регистров сопроцессора плавающей точки и MMX/SSE-контекста происходит при первом обращении нового потока, что оптимизировано под случай, когда большинство потоков производит только операции с регистрами общего назначения.

Вышеприведенные примеры относятся к ядру Linux, однако прочие операционные системы так же применяют сходные методы, хотя в случае проприетарных ОС доказать/опровергнуть использование этого является проблематичным.

Дополнительная информация

Вывод списка потоков

Утилита **ps** по умолчанию для многопоточного процесса выводит одну строку. Чтобы выводить несколько, можно выполнить

```
$ ps -L
```

Утилита **htop** выводит список потоков. Чтобы выводить только процессы, есть опция «Hide userland threads».

Потоки и nice

Напомним, что число **nice** задаёт приоритет выполнения процесса: от −20 (наивысший приоритет) до +19 (низший приоритет).

Хоть это и противоречит стандарту, но потоки не разделяют nice-значение.

Если вы применяете утилиту **renice**, то нужно применить её к каждому потоку.

Потоки и сигналы

Сигналы были придуманы задолго до появления **Pthreads**.

Комбинирование потоков и сигналов — сложное дело, стоит его избегать почти всегда.

- Действие сигнала распространяется на весь процесс.
- Настройка обработчиков также общая на весь процесс.
- Если для сигнала назначен обработчик, то он может оказаться вызванным в произвольном потоке данного процесса (часто это главный поток, но не всегда). Получается, что один поток приостанавливается, но при этом в момент выполнения обработчика другие потоки продолжают выполнять свою работу.
- Однако для сигналов, вызванных машинными исключениями, таких как **SIGSEGV** и **SIGFPE**, обработчик будет запущен в том потоке, который этот сигнал породил.
- Сигнальная маска у каждого потока своя. Изменением маски (с помощью функции **pthread_sigmask**) можно настроить, разрешается ли в данном потоке выполнять обработчик того или иного сигнала.
- Сигнал может быть направлен конкретному потоку посредством специальной функции **pthread_kill**.

Потоки и fork — exec

Если поток вызывает **exec**(), остальные потоки тут же уничтожаются, никакие деструкторы и функции очистки не вызываются.

Если поток вызывает **fork**(), то только этот поток будет продолжать работать в новом дочернем процессе.

Список использованных источников

1. Unix2019b/Потоки

https://acm.bsu.by/wiki/Unix2019b/%D0%9F%D0%BE%D1%82%D0%BE%D0%BA%D0%B8

2. Потоки и работа с ними

https://learn.microsoft.com/ru-ru/dotnet/standard/threading/threads-and-threading

3. Таненбаум, Э. Современные операционные системы. / Э. Таненбаум, Х. Бос. — 4-е изд. — СПб.: Питер, 2015. — 1120 с.

4. Параллельные потоки

http://gurin.tomsknet.ru/delphithreads.html

5. Процессы, потоки и конкурентное выполнение программ

https://serghei.blog/posts/2021/10/08/processy-potoki-i-konkurentnoe-vypolnenie-programm.html

[up] Состояния потоков и планирование их выполнения
https://upread.ru/blog/articles-it/sostoyaniya-potokov

Системные вызовы:
https://acm.bsu.by/wiki/Unix2019b/%D0%A1%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%BD%D1%8B%D0%B5_%D0%B2%D1%8B%D0%B7%D0%BE%D0%B2%D1%8B