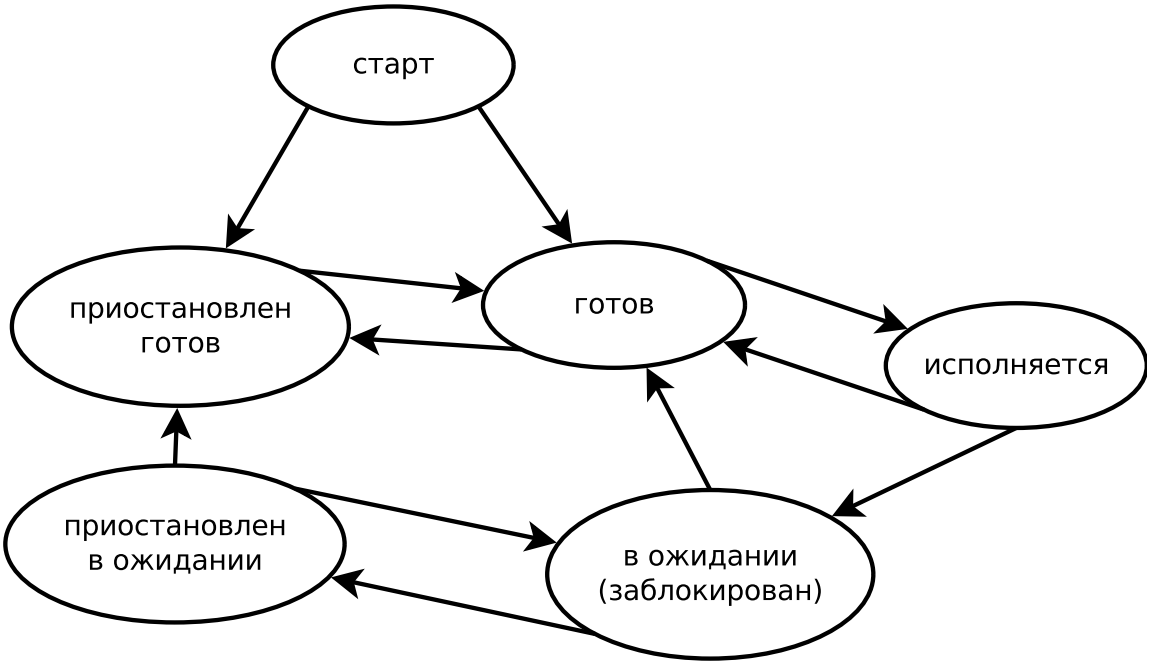
OS

Оглавление

Глава 1

Процесс (информатика)



Статусы процессов в современных ОС.

Проце́сс — команда, которая выполняется в текущий момент. Стандарт ISO 9000:2000 определяет процесс как совокупность взаимосвязанных и взаимодействующих действий, преобразующих входящие данные в исходящие.

Компьютерная программа сама по себе — это только пассивная совокупность инструкций, в то время как процесс — это непосредственное выполнение этих инструкций.

Часто процессом называют выполняющуюся программу и все её элементы: адресное пространство, глобальные переменные, регистры, стек, открытые файлы и т. д.

1.1. Создание процесса

Простейшей операционной системе не требуется создание новых процессов, поскольку внутри них работает одна-единственная программа, запускаемая во время включения устройства. В более сложных системах надо создавать новые процессы. Обычно они создаются:

1. При запуске ОС,

2. При появлении запроса на создание процесса — происходит в случае, если работающий процесс создает новый процесс.

1.2. Завершение процесса

Минимум 2 этапа завершения:

1. Процесс удаляется из всех очередей планирования, т.е. ОС больше не планирует выделение каких-либо ресурсов процессу
2. Сбор статистики о потреблённых процессом ресурсах с последующим удалением его из памяти

Причины завершения процесса:

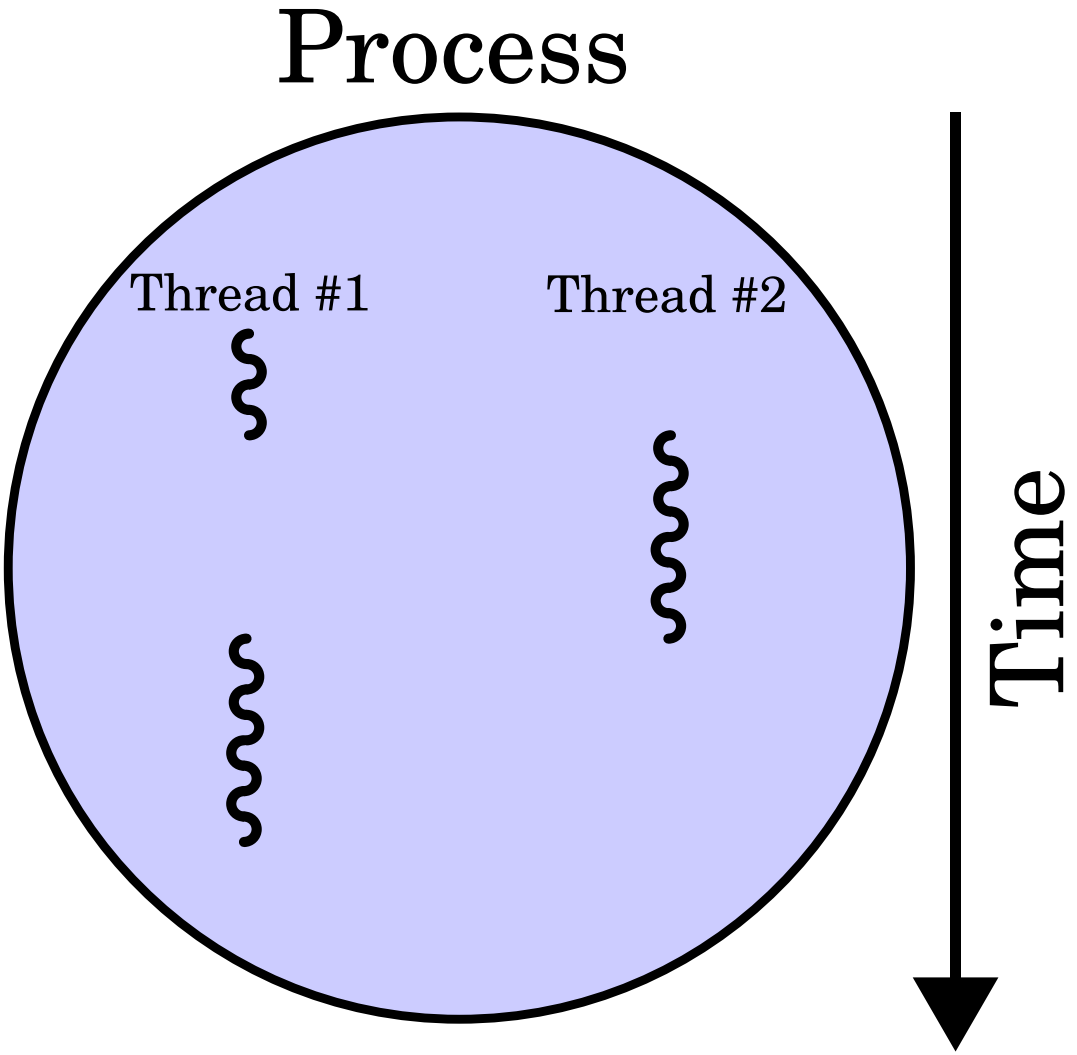
1. Обычный выход
2. Выход по исключению или ошибке
3. Недостаточный объем памяти
4. Превышение лимита отведённого программе времени
5. Выход за пределы отведённой области памяти
6. Неверная команда (данные интерпретируются как команды)
7. Ошибка защиты
8. Завершение родительского процесса
9. Ошибка ввода-вывода
10. Вмешательство оператора

1.3. Литература

- *Э. Таненбаум, А. Вудхалл.* «Операционные системы: Разработка и реализация.» — СПб.: 2006. — ISBN 5-469-00148-2
- *Э. Таненбаум.* «Современные операционные системы. 2-е изд.» — СПб.: Питер, 2005. — 1038 с.: ил. ISBN 5-318-00299-4

Глава 2

Поток выполнения



Процесс с двумя потоками выполнения на одном процессоре

Пото́к выполне́ния (тред; от англ. *thread* — нить) — наименьшая единица обработки, исполнение которой может быть назначено ядром операционной системы. Реализация потоков выполнения и процессов в разных операционных системах отличается друг от друга, но в большинстве случаев поток выполнения находится внутри процесса. 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 част-

ности, потоки выполнения разделяют инструкции процесса (его код) и его контекст (значения переменных, которые они имеют в любой момент времени). В качестве аналогии потоки выполнения процесса можно уподобить нескольким вместе работающим поварам. Все они готовят одно блюдо, читают одну и ту же кулинарную книгу с одним и тем же рецептом и следуют его указаниям, причём не обязательно все они читают на одной и той же странице.

На одном процессоре *многопоточность* обычно происходит путём временного мультиплексирования (как и в случае многозадачности): процессор переключается между разными потоками выполнения. Это переключение контекста обычно происходит достаточно часто, чтобы пользователь воспринимал выполнение потоков или задач как одновременное. В многопроцессорных и многоядерных системах потоки или задачи могут реально выполняться одновременно, при этом каждый процессор или ядро обрабатывает отдельный поток или задачу.

Многие современные операционные системы поддерживают как временные нарезки от планировщика процессов, так и многопроцессорные потоки выполнения. Ядро операционной системы позволяет программистам управлять потоками выполнения через интерфейс системных вызовов. Некоторые реализации ядра называют *потоком ядра*, другие же — *лёгким процессом* (англ. *light-weight process*, LWP), представляющим собой особый тип потока выполнения ядра, который совместно использует одни и те же состояния и данные.

Программы могут иметь *пользовательское пространство потоков выполнения* при создании потоков с помощью таймеров, сигналов или другими методами, позволяющими прервать выполнение и создать временную нарезку для конкретной ситуации (Ad hoc).

2.1. Отличие от процессов

Потоки выполнения отличаются от традиционных процессов многозадачной операционной системы тем, что:

- процессы, как правило, независимы, тогда как потоки выполнения существуют как составные элементы процессов
- процессы несут значительно больше информации о состоянии, тогда как несколько потоков выполнения внутри процесса совместно используют информацию о состоянии, а также память и другие вычислительные ресурсы
- процессы имеют отдельные адресные пространства, тогда как потоки выполнения совместно используют их адресное пространство
- процессы взаимодействуют только через предоставляемые системой механизмы связей между процессами
- переключение контекста между потоками выполнения в одном процессе, как правило, быстрее, чем переключение контекста между процессами.

Такие системы, как Windows NT и OS/2, как говорят, имеют «дешёвые» потоки выполнения и «дорогие» процессы. В других операционных системах разница между потоками выполнения и процессами не так велика, за исключением расходов на переключение адресного пространства, которое подразумевает использование буфера ассоциативной трансляции.

2.2. Многопоточность

Многопоточность, как широко распространённая модель программирования и исполнения кода, позволяет нескольким потокам выполняться в рамках одного процесса. Эти потоки выполнения совместно используют ресурсы процесса, но могут работать и самостоятельно. Многопоточная модель программирования предоставляет разработчикам удобную абстракцию параллельного выполнения. Однако, пожалуй, наиболее интересное применение технологии имеется в том случае, когда она применяется к *одному* процессу, что позволяет его *параллельное выполнение* на *многопроцессорной* системе.

Это преимущество многопоточной программы позволяет ей работать быстрее на компьютерных системах, которые имеют несколько процессоров, процессор с несколькими ядрами или на кластере машин — из-за

того, что потоки выполнения программ естественным образом поддаются действительно параллельному выполнению процессов. В этом случае программисту нужно быть очень осторожным, чтобы избежать состояния гонки, и другого неинтуитивного поведения. Для того, чтобы правильно манипулировать данными, потоки выполнения должны часто проходить через процедуру рандеву, чтобы обрабатывать данные в правильном порядке. Потокам выполнения могут также потребоваться мьютексы (которые часто реализуются с использованием семафоров), чтобы предотвратить одновременное изменение общих данных или их чтение во время процесса изменения. Неосторожное использование таких примитивов может привести к тупиковой ситуации.

Другим использованием многопоточности, применяемым даже для однопроцессорных систем, является возможность для приложения реагирования на ввод данных. В однопоточных программах, если основной поток выполнения заблокирован выполнением длительной задачи, всё приложение может оказаться в замороженном состоянии. Перемещая такие длительные задачи в *рабочий поток*, который выполняется параллельно с основным потоком, становится возможным для приложений продолжать реагировать на действия пользователя во время выполнения задач в фоновом режиме. С другой стороны, в большинстве случаев многопоточность — не единственный способ сохранить чувствительность программы. То же самое может быть достигнуто через асинхронный ввод-вывод или сигналы в UNIX.[1]

Операционные системы планируют выполнение потоков одним из двух способов:

1. *Приоритетная многопоточность*, вообще говоря, считается более совершенным подходом, так как она позволяет операционной системе определить, когда должно происходить переключение контекста. Недостаток приоритетной многопоточности состоит в том, что система может сделать переключение контекста в неподходящее время, что приводит к инверсии приоритета и другим негативным эффектам, которых можно избежать, применяя кооперативную многопоточность.
2. *Кооперативная многопоточность* полагается на сами потоки и отказывается от управления, если потоки выполнения находятся в точках остановки. Это может создать проблемы, если поток выполнения ожидает ресурс, пока он не станет доступным.

До конца 1990-х процессоры в настольных компьютерах не имели поддержки многопоточности, так как переключение между потоками, как правило, происходило быстрее, чем полное переключение контекста процесса. Процессоры во встраиваемых системах, которые имеют более высокие требования к поведению в реальном времени, могут поддерживать многопоточность за счёт уменьшения времени на переключение между потоками, возможно, путём распределения выделенных регистровых файлов для каждого потока выполнения, вместо сохранения/восстановления общего регистрового файла. В конце 1990-х идея выполнения инструкций нескольких потоков одновременно, известная как одновременная многопоточность, под названием Hyper-Threading, достигла настольных компьютеров с процессором Intel Pentium 4. Потом она была исключена из процессоров архитектуры Intel Core и Core 2, но позже восстановлена в архитектуре Core i7.

Критики многопоточности утверждают, что увеличение использования потоков имеет существенные недостатки:

> Хотя кажется, что потоки выполнения — это небольшой шаг от последовательных вычислений, по сути они представляют собой огромный скачок. Они отказываются от наиболее важных и привлекательных свойств последовательных вычислений: понятности, предсказуемости и детерминизма. Потоки выполнения, как модель вычислений, являются потрясающе недетерминированными, и работа программиста становится одним из обрезков этого недетерминизма.[2]

2.3. Процессы, потоки выполнения ядра, пользовательские потоки и файберы

Процесс является «самой тяжёлой» единицей планирования ядра. Собственные ресурсы для процесса выделяются операционной системой. Ресурсы включают память, дескрипторы файлов, разъёмы, дескрипторы устройств и окна. Процессы используют адресное пространство и файлы ресурсов в режиме разделения времени только через явные методы, такие как наследование дескрипторов файлов и сегментов разделяемой памяти. Процессы, как правило, предварительно преобразованы к многозадачному способу выполнения.

Потоки выполнения ядра относятся к «лёгким» единицам планирования ядра. Внутри каждого процесса существует по крайней мере один поток выполнения ядра. Если в рамках процесса могут существовать несколь-

ко потоков выполнения ядра, то они совместно используют общую память и файл ресурсов. Если процесс выполнения планировщика операционной системы является приоритетным, то потоки выполнения ядра тоже являются приоритетно многозадачными. Потоки выполнения ядра не имеют собственных ресурсов, за исключением стека вызовов, копии регистров процессора, включая счётчик команд, и локальную память потока выполнения (если она есть). Ядро может назначить по одному потоку выполнения для каждого логического ядра системы (поскольку каждый процессор разделяет сам себя на несколько логических ядер, если он поддерживает многопоточность, либо поддерживает только одно логическое ядро на каждое физическое ядро, если не поддерживает многопоточность), а может выполнять свопинг заблокированных потоков выполнения. Однако потоки выполнения ядра требует гораздо больше времени, чем требуется на свопинг пользовательских потоков выполнения.

Потоки выполнения иногда реализуются в пользовательском пространстве библиотек, в этом случае они называются *пользовательскими потоками выполнения*. Ядро не знает о них, так что они управляются и планируются в пользовательском пространстве. В некоторых реализациях *пользовательские потоки выполнения* основываются на нескольких верхних *потоках выполнения ядра*, чтобы использовать преимущества многопроцессорных машин (модели M:N). В данной статье под термином «поток выполнения» по умолчанию (без квалификатора «ядра» или «пользовательский») имеется в виду «поток выполнения ядра». Пользовательские потоки выполнения, реализованные с помощью виртуальных машин, называют также «зелёными потоками выполнения». Пользовательские потоки выполнения, как правило, можно быстро создавать, и ими легко управлять, но они не могут использовать преимущества многопоточности и многопроцессорности. Они могут блокироваться, если все связанные с ним потоки выполнения ядра заняты, даже если некоторые пользовательские потоки готовы к запуску.

Файберы являются ещё более «лёгкими» блоками планирования, относящимися к кооперативной многозадачности: выполняющийся файбер должен явно «уступить» право другим файберам на выполнение, что делает их реализацию гораздо легче, чем реализацию потоков выполнения ядра или пользовательских потоков выполнения. Файберы могут быть запланированы для запуска в любом потоке выполнения внутри того же процесса. Это позволяет приложениям получить повышение производительности за счет управления планированием самого себя, вместо того чтобы полагаться на планировщик ядра (который может быть не настроен на такое применение). Параллельные среды программирования, такие как OpenMP, обычно реализуют свои задачи посредством файберов.

2.3.1. Сущность потока выполнения и файбера

Одновременность и структуры данных

Потоки выполнения одного и того же процесса совместно используют одно и то же адресное пространство. Это позволяет одновременно выполняющимся кодам быть плотно связанными и удобно обмениваться данными без накладных расходов и сложности межпроцессного взаимодействия. При распределении даже простых структур данных между потоками возникает опасность возникновения состояния гонки в том случае, если для обновления требуется более одной инструкции процессора: два потока выполнения могут в конечном итоге попытаться одновременно обновить структуры данных и найти неожиданное доступное решение. Ошибки, вызванные состоянием гонки, бывает очень трудно воспроизвести и выделить.

Чтобы избежать этого, прикладные программные интерфейсы (API) потоков выполнения предлагают примитивы синхронизации, такие как мьютексы, для блокировки структур данных от одновременного доступа. На однопроцессорных системах поток выполнения, обратившийся к заблокированному мьютексу, должен остановить работу и, следовательно, инициировать переключение контекста. На многопроцессорных системах поток выполнения может вместо опроса мьютекса произвести захват спинлока. Оба этих способа могут снижать производительность и вынуждать процессор в SMP-системах конкурировать за шину памяти, особенно если уровень модульности блокировок слишком высокий.

Ввод-вывод и планирование

Реализация пользовательских потоков выполнения и файберов, как правило, производится полностью в пользовательском пространстве. В результате переключение контекста между пользовательскими потоками выполнения и файберами в одном и том же процессе очень эффективно, поскольку вообще не требует никакого взаимодействия с ядром. Переключение контекста производится локально путём сохранения регистров процессора, используемых работающим пользовательским потоком выполнения или файбером, и затем загруз-

кой регистров, требуемых для нового выполнения. Поскольку планирование происходит в пользовательском пространстве, политика планирования может быть легко адаптирована к требованиям конкретной программы.

Однако использование блокировок системных вызовов для пользовательских потоков выполнения (в отличие от потоков выполнения ядра) и файберов имеет свои проблемы. Если пользовательский поток выполнения или файбер выполняет системный вызов, другие потоки выполнения и файберы процесса не могут работать до завершения этой обработки. Типичный пример такой проблемы связан с выполнением операций ввода-вывода. Большинство программ рассчитаны на синхронное выполнение ввода-вывода. При инициации ввода-вывода делается системный вызов, и он не возвращается до его завершения. В промежутке весь процесс блокируется ядром и не может исполняться, лишая возможности работы другие пользовательские потоки и файберы этого процесса.

Общим решением этой проблемы является обеспечение отдельного API для ввода-вывода, который реализует синхронный интерфейс с использованием внутреннего неблокирующего ввода-вывода, и запуск другого пользовательского потока выполнения или файбера на время обработки ввода-вывода. Подобные решения могут быть предусмотрены для блокирующих системных вызовов. Кроме того, программа может быть написана так, чтобы избежать использования синхронного ввода-вывода или других блокирующих системных вызовов.

В SunOS 4.x реализованы так называемые «лёгкие процессы» или LWP. В NetBSD 2.x + и DragonFly BSD реализованы LWP как потоки выполнения ядра (модель 1:1). В SunOS 5.2 и до SunOS 5.8, а также в NetBSD 2 и до NetBSD 4 реализована двухуровневая модель, использующая один или несколько пользовательских потоков выполнения на каждый поток выполнения ядра (модель M:N). В SunOS 5.9 и последующих версиях, а также в NetBSD 5 ликвидирована поддержка пользовательских потоков выполнения, произошёл возврат к модели 1:1.[3] В FreeBSD 5 реализована модель M:N. В FreeBSD 6 поддерживаются обе модели: 1:1 и M:N, и пользователь может выбрать, какую из них он будет использовать в данной программе, используя /etc/libmap.conf. В FreeBSD начиная с версии 7 модель 1:1 стала использоваться по умолчанию, а в FreeBSD 8 и последующих версиях модель M:N не поддерживается совсем.

Использование потоков выполнения ядра упрощает код пользователя, перемещая некоторые из наиболее сложных аспектов многопотоковости в ядро. От программы не требуется планирование потоков выполнения и явных захватов процессора. Пользовательский код может быть записан в привычном процедурном стиле, включая вызовы блокирующего API без лишения доступа к процессору других потоков выполнения. Тем не менее, потоки выполнения ядра может вызвать переключение контекста между потоками выполнения в любое время и тем самым подвергнуть опасности появления ошибок гонки и одновременности, которые могли бы не возникать. На SMP-системах это ещё более усугубляется, потому как потоки выполнения ядра могут в прямом смысле выполняться одновременно на разных процессорах.

2.4. Модели

2.4.1. 1:1 (потоки выполнения на уровне ядра)

Потоки выполнения, созданные пользователем в модели 1-1, соответствуют диспетчируемым сущностям ядра. Это простейший возможный вариант реализации потоковости. В Windows API этот подход использовался с самого начала. В Linux обычная библиотека C реализует этот подход (через библиотеку потоков POSIX, а в более старших версиях через LinuxThreads). Такой же подход используется ОС Solaris, NetBSD и FreeBSD.

2.4.2. N:1 (потоки выполнения уровня пользователя)

В модели N:1 предполагается, что все потоки выполнения уровня пользователя отображаются на единую планируемую сущность уровня ядра, и ядро ничего не знает о составе прикладных потоков выполнения. При таком подходе переключение контекста может быть сделано очень быстро, и, кроме того, он может быть реализован даже на простых ядрах, которые не поддерживают многопоточность. Однако, одним из главных недостатков его является то, что в нём нельзя извлечь никакой выгоды из аппаратного ускорения на многопоточных процессорах или многопроцессорных компьютерах, потому что только один поток выполнения может быть запланирован на любой момент времени. Эта модель используется в GNU Portable Threads.

2.4.3. M:N (смешанная потоковость)

В модели M:N некоторое число N прикладных потоков выполнения отображаются на некоторое число M сущностей ядра или «виртуальных процессоров». Модель является компромиссной между моделью уровня ядра («1:1») и моделью уровня пользователя («N:1»). Вообще говоря, «M:N» потоковость системы являются более сложной для реализации, чем ядро или пользовательские потоки выполнения, поскольку изменение кода как для ядра, так и для пользовательского пространства не требуется. В M:N реализации библиотека потоков отвечает за планирование пользовательских потоков выполнения на имеющихся планируемых сущностях. При этом переключение контекста потоков делается очень быстро, поскольку модель позволяет избежать системных вызовов. Тем не менее, увеличивается сложность и вероятность инверсии приоритетов, а также неоптимальность планирования без обширной (и дорогой) координации между пользовательским планировщиком и планировщиком ядра.

2.5. Реализации

Есть много различных, несовместимых друг с другом реализаций потоков. К ним относятся как реализации на уровне ядра, так и реализации на пользовательском уровне. Чаще всего они придерживаются более или менее близко стандарта интерфейса POSIX Threads.

2.5.1. Примеры реализаций потоков на уровне ядра

- Light Weight Kernel Threads (LWKT) в различных версиях BSD
- Потоковость MxN
- Библиотека потоков POSIX (NPTL) для Linux, реализация стандарта POSIX Threads
- Apple Multiprocessing Services, версия 2.0 и последующие, использует встроенное микроядро в Mac OS 8.6, в более поздних версиях сделана модификация с целью последующего сопровождения.
- Windows начиная с Windows 95, Windows NT и после них.

2.5.2. Примеры реализаций потоков на уровне пользователя

- GNU Portable Threads
- FSU Pthreads
- Thread Manager компании Apple
- REALbasic (включая API для совместного использования потоков)
- Netscape Portable Runtime (включая реализацию файберов в пользовательском пространстве)

2.5.3. Примеры реализаций смешанных потоков

- «Scheduler activations» используется в собственной библиотеке приложений потоков POSIX для NetBSD (модель M:N в противоположность модели 1:1 ядра или модели приложений пользовательского пространства)
- Marcel из проекта PM2
- ОС для суперкомпьютера Tera/Cray MTA
- Windows 7

2.5.4. Примеры реализаций файберов

Файберы могут быть реализованы без поддержки операционной системы, хотя некоторые операционные системы и библиотеки предоставляют явную поддержку для них.

- Библиотека Win32 содержит API для файберов[4] (Windows NT 3.51 SP3 и выше)
- Ruby как реализация «зелёных потоков»

2.6. Поддержка языков программирования

Многие языки программирования поддерживают потоки иначе. Большинство реализаций C и C++ сами по себе не обеспечивают прямой поддержки потоков, но обеспечивают доступ к потокам, предоставляемым операционной системой, через API. Некоторые языки программирования более высокого уровня (как правило, кросс-платформенные), такие как Java, Python, и .NET, предоставляют потоковость разработчику в виде абстрактной специфической платформы, отличающейся от реализации потоков в среде выполнения разработчика. Ряд других языков программирования также пытаются полностью абстрагировать концепцию параллелизма и потоковости от разработчика (Cilk, OpenMP, MPI…). Некоторые языки разрабатываются специально для параллелизма (Ateji PX, CUDA).

Некоторые интерпретирующие языки программирования, такие как Руби и CPython (реализация Python) поддерживают потоки, но имеют ограничение, которое известно как глобальная блокировка интерпретатора (GIL). GIL является взаимной блокировкой исключений, выполняемых интерпретатором, которая может уберечь интерпретатор от одновременной интерпретации кода приложений в двух или более потоках одновременно, что фактически ограничивает параллелизм на многоядерных системах (в основном для потоков, связанных через процессор, а не для потоков, связанных через сеть).

2.7. См. также

- Поток данных
- Обмен сообщениями
- Thread-safety
- Неблокирующая синхронизация

2.8. Примечания

[1] Sergey Ignatchenko. Single-Threading: Back to the Future? Overload #97

[2] *Edward A. Lee.* The Problem with Threads. *Technical Report No. UCB/EECS-2006-1*. EECS Department, University of California, Berkeley (10 января 2006). Проверено 30 мая 2012. Архивировано из первоисточника 26 июня 2012.

[3] Oracle and Sun

[4] CreateFiber MSDN

2.9. Литература

- David R. Butenhof. Programming with POSIX Threads. Addison-Wesley. ISBN 0-201-63392-2
- Bradford Nichols, Dick Buttlar, Jacqueline Proulx Farell. Pthreads Programming. O'Reilly & Associates. ISBN 1-56592-115-1
- Charles J. Northrup. Programming with UNIX Threads. John Wiley & Sons. ISBN 0-471-13751-0

- Mark Walmsley. Multi-Threaded Programming in C++. Springer. ISBN 1-85233-146-1
- Paul Hyde. Java Thread Programming. Sams. ISBN 0-672-31585-8
- Bill Lewis. Threads Primer: A Guide to Multithreaded Programming. Prentice Hall. ISBN 0-13-443698-9
- Steve Kleiman, Devang Shah, Bart Smaalders. Programming With Threads, SunSoft Press. ISBN 0-13-172389-8
- Pat Villani. Advanced WIN32 Programming: Files, Threads, and Process Synchronization. Harpercollins Publishers. ISBN 0-87930-563-0
- Jim Beveridge, Robert Wiener. Multithreading Applications in Win32. Addison-Wesley. ISBN 0-201-44234-5
- Thuan Q. Pham, Pankaj K. Garg. Multithreaded Programming with Windows NT. Prentice Hall. ISBN 0-13-120643-5
- Len Dorfman, Marc J. Neuberger. Effective Multithreading in OS/2. McGraw-Hill Osborne Media. ISBN 0-07-017841-0
- Alan Burns, Andy Wellings. Concurrency in ADA. Cambridge University Press. ISBN 0-521-62911-X
- Uresh Vahalia. Unix Internals: the New Frontiers. Prentice Hall. ISBN 0-13-101908-2
- Alan L. Dennis. .Net Multithreading. Manning Publications Company. ISBN 1-930110-54-5
- Tobin Titus, Fabio Claudio Ferracchiati, Srinivasa Sivakumar, Tejaswi Redkar, Sandra Gopikrishna. C# Threading Handbook. Peer Information. ISBN 1-86100-829-5
- Tobin Titus, Fabio Claudio Ferracchiati, Srinivasa Sivakumar, Tejaswi Redkar, Sandra Gopikrishna. Visual Basic .Net Threading Handbook. Wrox Press. ISBN 1-86100-713-2

2.10. Ссылки

- Answers to frequently asked questions for comp.programming.threads
- What makes multi-threaded programming hard?
- Binildas C. A. Query by Slice, Parallel Execute, and Join: A Thread Pool Pattern in Java
- Herb Sutter. The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software
- Concepts of Multithreading
- ConTest — A Tool for Testing Multithreaded Java Applications by IBM
- Debugging and Optimizing Multithreaded OpenMP Programs
-
- *Multithreading* в каталоге ссылок Open Directory Project (dmoz).
- Multithreading in the Solaris Operating Environment
- Parallel computing community
- Daniel Robbins. POSIX threads explained
- The C10K problem

Глава 3

POSIX Threads

POSIX Threads — стандарт POSIX реализации потоков (нитей) выполнения. 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_»).

3.1. Основные функции стандарта

Pthreads определяет набор типов и функций на языке программирования Си. Заголовочный файл — pthread.h.

- Типы данных:
  - pthread_t: дескриптор потока;
  - pthread_attr_t: перечень атрибутов потока;
  - pthread_barrier_t: барьер;
  - pthread_barrierattr_t: атрибуты барьера;
  - pthread_cond_t: условная переменная;
  - pthread_condattr_t: атрибуты условной переменной;
  - pthread_key_t: данные, специфичные для потока;
  - pthread_mutex_t: мьютекс;
  - pthread_mutexattr_t: атрибуты мьютекса;
  - pthread_rwlock_t: ;
  - pthread_rwlockattr_t: ;
  - pthread_spinlock_t: спинлок;
- Функции управления потоками:
  - pthread_create(): создание потока.
  - pthread_exit(): завершение потока (должна вызываться функцией потока при завершении).
  - pthread_cancel(): отмена потока.
  - pthread_join(): подключиться к другому потоку и ожидать его завершения; поток, к которому необходимо подключиться, должен быть создан с возможностью подключения (PTHREAD_CREATE_JOINABLE).
  - pthread_detach(): отключиться от потока, сделав его при этом отдельным (PTHREAD_CREATE_DETACHED).
  - pthread_attr_init(): инициализировать структуру атрибутов потока.

  - pthread_attr_setdetachstate(): указывает параметр "отделимости" потока (detach state), который говорит о возможности подключения к нему (при помощи pthread_join) других потоков (значение PTHREAD_CREATE_JOINABLE) для ожидания окончания или о запрете подключения (значение PTHREAD_CREATE_DETACHED); ресурсы отдельного потока (PTHREAD_CREATE_DETACHED) при завершении автоматически освобождаются и возвращаются системе.
  - pthread_attr_destroy(): освободить память от структуры атрибутов потока (уничтожить дескриптор).

- Функции синхронизации потоков:
  - pthread_mutex_init(), pthread_mutex_destroy(), pthread_mutex_lock(), pthread_mutex_trylock(), pthread_mutex_unlock() с помощью мьютексов.
  - pthread_cond_init(), pthread_cond_signal(), pthread_cond_wait(): с помощью условных переменных.

3.2. Пример

Пример использования потоков на языке C:

#include <stdio.h> #include <stdlib.h> #include <time.h> #include <pthread.h> static void wait_thread(void) { time_t start_time = time(NULL); while (time(NULL) == start_time) { /* do nothing except chew CPU slices for up to one second. */ } } static void *thread_func(void *vptr_args) { int i; for (i = 0; i < 20; i++) { fputs(" b\n", stderr); wait_thread(); } return NULL; } int main(void) { int i; pthread_t thread; if (pthread_create(&thread, NULL, thread_func, NULL) != 0) { return EXIT_FAILURE; } for (i = 0; i < 20; i++) { puts("a"); wait_thread(); } if (pthread_join(thread, NULL) != 0) { return EXIT_FAILURE; } return EXIT_SUCCESS; }

Пример использования потоков на языке C++:

#include <cstdlib> #include <iostream> #include <memory> #include <unistd.h> #include <pthread.h> class Thread { private: pthread_t thread; Thread(const Thread& copy); // copy constructor denied static void *thread_func(void *d) { ((Thread *)d)->run(); return NULL; } public: Thread() {} virtual ~Thread() {} virtual void run() = 0; int start() { return pthread_create(&thread, NULL, Thread::thread_func, (void*)this); } int wait () { return pthread_join (thread, NULL); } }; typedef std::auto_ptr<Thread> ThreadPtr; int main(void) { class Thread_a:public Thread { public: void run() { for (int i=0; i<20; i++, sleep(1)) std::cout << "a " << std::endl; } }; class Thread_b:public Thread { public: void run() { for(int i=0; i<20; i++, sleep(1)) std::cout << " b" << std::endl; } }; ThreadPtr a(new Thread_a()); ThreadPtr b(new Thread_b()); if (a->start() != 0 || b->start() != 0) return EXIT_FAILURE; if (a->wait() != 0 || b->wait() != 0) return EXIT_FAILURE; return EXIT_SUCCESS; }

Представленные программы используют два потока, печатающих в консоль сообщения, один, печатающий 'a', второй — 'b'. Вывод сообщений смешивается в результате переключения выполнения между потоками или одновременном выполнении на мультипроцессорных системах.

Отличие состоит в том, что программа на C создает один новый поток для печати 'b', а основной поток печатает 'a'. Основной поток (после печати 'aaaaa….') ждёт завершения дочернего потока.

Программа на C++ создает два новых потока, один печатает 'a', второй, соответственно, — 'b'. Основной поток ждёт завершения обоих дочерних потоков.

3.3. См. также

- Native POSIX Thread Library (NPTL)
- GNU Portable Threads
- Список многопоточных библиотек C++

3.4. Ссылки

- Статья «Объясняя потоки POSIX», Даниэля Роббинса (основателя проекта *Gentoo*) (англ.)
- Интервью «10 вопросов Девиду Бутенхофу о параллельном программировании и потоках POSIX» с Майклом Суиссом (англ.)
- The Open Group Base Specifications Issue 6, IEEE Std 1003.1 (англ.)
- Pthreads Presentation at 2007 OSCON (O'Reilly Open Source Convention) by Adrien Lamothe. An overview of Pthreads with current trends (англ.)

3.5. Источники текстов и изображения, авторы и лицензии

3.5.1. Текст

- **Процесс (информатика)** *Источник:* http://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D1%86%D0%B5%D1%81%D1%81%20(%D0%B8%D0%BD%D1%84%D0%BE%D1%80%D0%BC%D0%B0%D1%82%D0%B8%D0%BA%D0%B0)?oldid=65180044 *Авторы:* Dims, Plest, Denver, Amoses, SergeyPosokhov, Mospehraict, Azh7, Thijs!bot, JAnDbot, DrCroco, Ficus, VolkovBot, Idiomabot, РоманСузи, RaMaXa, TarzanASG, Cantor, SieBot, Sirnik, Coob, Aspid812, Менязовут, VanBot, Fractaler, Zorrobot, Muro Bot, Luckas-bot, Ivan A. Krestinin, 4th-otaku, Rcrvano, DenisKrivosheev, D'ohBot, Tretyak, EmausBot, Beta M, Ebrambot, WikitanvirBot, Movses-bot, MerlIwBot, Robiteria, Addbot и Аноним: 9
- **Поток выполнения** *Источник:* http://ru.wikipedia.org/wiki/%D0%9F%D0%BE%D1%82%D0%BE%D0%BA%20%D0%B2%D1%8B%D0%BF%D0%BE%D0%BB%D0%BD%D0%B5%D0%BD%D0%B8%D1%8F?oldid=64349290 *Авторы:* Softy, Incnis Mrsi, РоманСузи, РобоСтася, CarsracBot, Beefeather, Zorrobot, AVB, Лев Дубовой, IlyaMart, Tpyvvikky, Ripchip Bot, EmausBot, Radislav72, H2Bot, WikitanvirBot, WebCite Archiver, Robiteria, Addbot, MarchHare1977 и Аноним: 9
- **POSIX Threads** *Источник:* http://ru.wikipedia.org/wiki/POSIX%20Threads?oldid=64589408 *Авторы:* Starling13, Velikodniy, Yuriybrisk, Chobot, Incnis Mrsi, Infovarius, Lazil, Rei-bot, VolkovBot, Важнов Алексей Геннадьевич, Peni, Gribozavr, AlleborgoBot, Aessone, Jsg08, Animist, JackieBot, MaxMax, SlavMFM, Polymorphm, DSisyphBot, DenisKrivosheev, DenMoren, X7q, Ботильда, DmitryVinokurov, Annulen, RedBot, EmausBot, CapBuran, VanchesterUnited, KPu3uC B Poccuu, Well-Informed Optimist, Adil nurimanov, Addbot, EvRubot и Аноним: 19

3.5.2. Изображения

- **Файл:Computer.svg** *Источник:* http://upload.wikimedia.org/wikipedia/commons/d/d7/Computer.svg *Лицензия:* Public domain *Авторы:* The Tango! Desktop Project *Художник:* The people from the Tango! project
- **Файл:Multithreaded_process.svg** *Источник:* http://upload.wikimedia.org/wikipedia/commons/a/a5/Multithreaded_process.svg *Лицензия:* CC-BY-SA-3.0 *Авторы:* Own work in Inkscape *Художник:* en:User:Cburnett
- **Файл:Process_states.ru.svg** *Источник:* http://upload.wikimedia.org/wikipedia/commons/0/0c/Process_states.ru.svg *Лицензия:* FAL *Авторы:* собственная работа *Художник:* VolodyA! V Anarhist
- **Файл:Searchtool.svg** *Источник:* http://upload.wikimedia.org/wikipedia/commons/6/61/Searchtool.svg *Лицензия:* LGPL *Авторы:* http://ftp.gnome.org/pub/GNOME/sources/gnome-themes-extras/0.9/gnome-themes-extras-0.9.0.tar.gz *Художник:* David Vignoni, Ysangkok

3.5.3. Лицензия

- Creative Commons Attribution-Share Alike 3.0