КОНСТРУИРОВАНИЕ ПРОГРАММ И ЯЗЫКИ ПРОГРАМИРОВАНИЯ

Лекция № 24 — Параллельные вычисления

Преподаватель: Поденок Леонид Петрович, 505а-5

+375 17 293 8039 (505а-5)
+375 17 320 7402 (ОИПИ НАНБ)
prep@lsi.bas-net.by

ftp://student:2ok*uK2@Rwox@lsi.bas-net.by

Кафедра ЭВМ, 2021

Оглавление

Библиотека поддержки потоков <threads>

Библиотека предоставляет следующие компоненты для
- создания потоков (threads) и управления ими;
- реализации взаимных исключений (mutual exclusion);
- работы с условными переменными;
- передачи значений между потоками.

Системные возможности компьютеров (аппаратура и ОС)

Современные вычислительные системы обладают т.н. **многозадачными**[1] **возможностями**.

Многозадачность — это один из основных параметров всех современных вычислительных систем (ВС). Это когда процессор выполняет какую-то одну программу (процесс или задача). По истечении некоторого времени (микросекунды), операционная система переключает процессор на другую программу. При этом все регистры текущей программы и прочее состояние (контексты) сохраняются. Через некоторое время вновь передается управление этой программе. Программа при этом не замечает каких либо изменений — для нее процесс переключения незаметен.

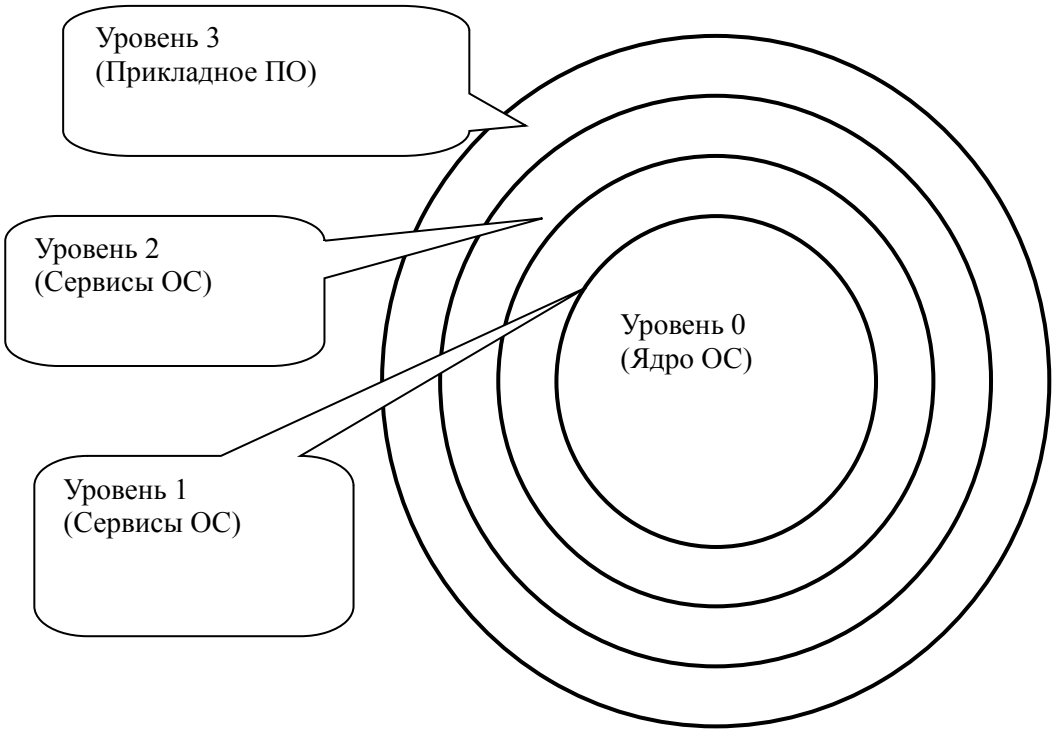
Для того чтобы программы, работающие в таком режиме, не могли каким либо образом нарушить работоспособность вычислительной системы или других программ, предусмотрен ряд механизмов защиты, в частности, уровни привилегий и изоляция программ друг от друга.

Уровень привилегий

1 мультизадачность – multitasking

Классическая схема работы программ по уровням привилегий имеет вид:

уровень 0: ядро операционной системы;
уровень 1: драйверы ОС;
уровень 2: интерфейс ОС;
уровень 3: прикладные программы.



Использование всех четырех уровней привилегий не является необходимым. Существующие системы, спроектированные с меньшим количеством уровней, могут просто игнорировать другие допустимые уровни.

***NIX** и **Windows**, например, используют только два уровня привилегий:

0 — ядро системы;
3 — все остальное.

OS/2 использует уровни:

0 — ядро системы;
2 — процедуры ввода-вывода;
3 — прикладные программы.

Процессы

Фундаментальным понятием для изучения работы современных ВС является понятие **процесса** — основного динамического объекта, над которыми операционная система (ОС) выполняет определенные действия.

По ходу работы программы ВС выполняет различные инструкции (команды) и преобразует значения переменных (содержимое ячеек памяти). Для этого ОС должна выделить программе определенное количество оперативной памяти, закрепить за ней определенные устройства ввода/вывода или файлы (откуда должны поступать входные данные и куда нужно доставить полученные результаты), то есть зарезервировать определенные ресурсы из общего числа ресурсов всей ВС. Количество и конфигурация ресурсов могут изменяться с течением времени.

Для описания таких активных объектов внутри вычислительной системы используется термин **«процесс»** — программа, загруженная в память и выполняющаяся.

Стандарт ISO 9000:2000 определяет процесс как совокупность взаимосвязанных и взаимодействующих действий, преобразующих входящие данные в исходящие.

Компьютерная программа — только пассивная совокупность инструкций.

Процесс — непосредственное выполнение программных инструкций.

Иногда процессом называют выполняющуюся программу и все её элементы:

- адресное пространство;
- глобальные переменные;
- регистры;
- стек;
- открытые файлы и прочие ресурсы.

Понятие процесса охватывает:

- некоторую совокупность исполняющихся команд (инструкций);

- совокупность ассоциированных с процессом ресурсов — выделенная для исполнения память или адресное пространство, состояние стека, значения переменных, используемые файлы, устройства ввода-вывода, и прочие ресурсы (прикладной контекст, системный контекст).

- текущий момент выполнения — значения регистров и программного счетчика, (регистровый контекст).

Взаимно однозначного соответствия между процессами и программами нет. Обычно в рамках программы можно организовывать более одного процесса.

Процесс находится под управлением операционной системы и поэтому в нем может выполняться часть кода ее ядра, которая не находится в исполняемом файле. Это происходит как в случаях, специально запланированных авторами программы, например, при использовании системных вызовов, так и в непредусмотренных ими ситуациях, например, при обработке внешних прерываний).

Процесс — некоторая совокупность набора исполняющихся команд, ассоциированных с ним ресурсов и текущего момента его выполнения, находящуюся под управлением ОС.

Все, что выполняется в вычислительных системах (программы пользователей и определенные части операционных систем), организовано в виде набора процессов.

Состояния процесса

На однопроцессорной компьютерной системе в каждый момент времени может исполняться только один процесс.

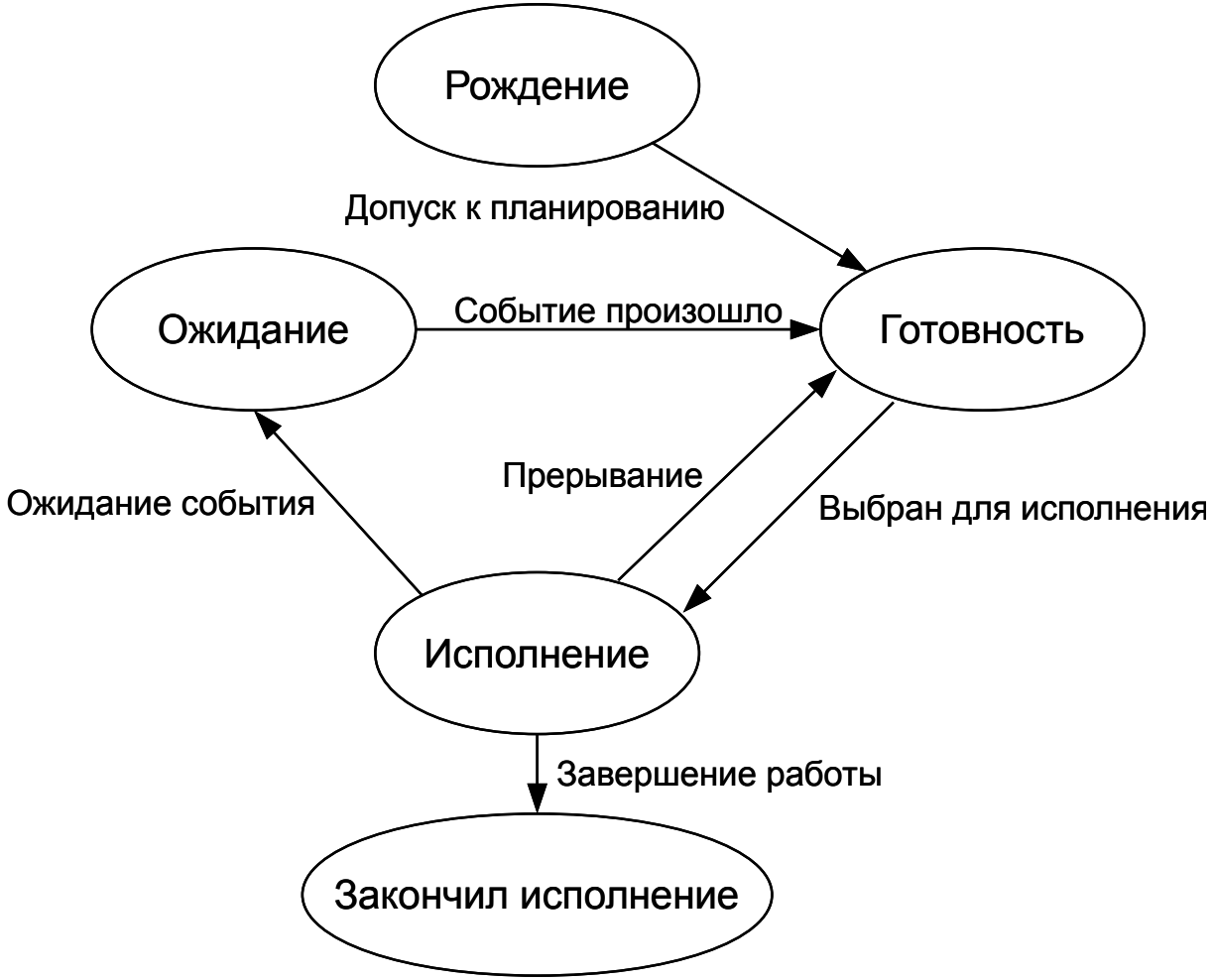
Для мультипрограммных вычислительных систем **псевдопараллельная** обработка нескольких процессов достигается с помощью **переключения процессора с одного процесса на другой** — пока один процесс выполняется, остальные ждут своей очереди на доступ к процессору.

Состояния процесса

Для появления в вычислительной системе процесс должен пройти через состояние **«рождение»**.

При рождении процессу выделяются адресное пространство, в которое загружается программный код процесса, стек и системные ресурсы, устанавливается начальное значение программного счетчика этого процесса и т. д.

После этого родившийся процесс переводится в состояние «готовность».



Контекст процесса

Для того чтобы операционная система могла выполнять операции над процессами, каждый процесс представляется в ней некоторой структурой данных — **блок управления процессом.**

Блок управления процессом содержит всю информацию, необходимую для выполнения операций над процессом — **он является моделью процесса для ОС**.

Эта структура содержит информацию, специфическую для данного процесса:

— состояние, в котором находится процесс;

— программный счетчик (адрес команды, которая будет выполнена следующей);

— содержимое регистров процессора;

— данные, необходимые для планирования использования процессора и управления памятью (приоритет , размер и расположение адресного пространства и т. д.);

— учетные данные (идентификационный номер процесса, какой пользователь инициировал его работу, общее время использования процессора данным процессом и т. д.);

— информацию об устройствах ввода-вывода, связанных с процессом (например, какие устройства закреплены за процессом, таблицу открытых файлов).

Во многих ОС информация, характеризующая процесс, хранится не в одной, а в нескольких связанных структурах данных. Эти структуры могут иметь различные наименования, содержать дополнительную информацию или, наоборот, лишь часть описанной информации (аппаратный вариант — TSS). Любая операция, производимая ОС над процессом, вызывает определенные изменения в PCB.

Информацию, для хранения которой предназначен блок управления процессом, можно разделить на две части:

- регистровый контекст — содержимое всех регистров процессора (включая значение программного счетчика);

- системный контекст — все остальное.

Знания регистрового и системного контекстов процесса достаточно для того, чтобы управлять его поведением в операционной системе, совершая над ним операции, однако недостаточно, чтобы полностью характеризовать процесс.

Пользовательский контекст

Операционную систему не интересует, какими именно вычислениями занимается процесс, т.е. какой код и какие данные находятся в его адресном пространстве.

С точки зрения пользователя, наоборот, наибольший интерес представляет содержимое адресного пространства процесса, возможно наряду с регистровым контекстом, определяющее последовательность преобразования данных и полученные результаты.

Пользовательский контекст — код и данные, находящиеся в адресном пространстве процесса.

Контекст процесса — совокупность регистрового, системного и пользовательского контекстов.

В любой момент времени процесс полностью характеризуется своим контекстом.

Создание и управление процессами

Взаимодействие процессов

Жизнь процессов в вычислительной системе состоит из постоянного ожидания в очереди к процессору и в постоянной борьбе за другие ресурсы. Для нормального функционирования процессов ОС старается максимально изолировать их друг от друга:

- каждый процесс имеет свое собственное адресное пространство, нарушение которого, как правило, приводит к аварийной остановке процесса (исключение);
- каждому процессу, по возможности, предоставляются свои собственные дополнительные ресурсы;

Тем не менее, для решения некоторых задач процессы имеют необходимость объединять свои усилия.

Для достижения поставленной цели различные процессы (возможно, принадлежащие разным пользователям) могут исполняться *псевдопараллельно* на одной ВС или параллельно на разных ВС, взаимодействуя между собой.

Причины для их взаимодействия:

- повышение скорости работы. Когда один процесс ожидает наступления некоторого события (например, окончания операции ввода-вывода), другие в это время могут заниматься полезной работой, направленной на решение общей задачи. В многопроцессорных вычислительных системах программа разделяется на отдельные кусочки, каждый из которых будет исполняться на своем процессоре.

- совместное использование данных. Различные процессы могут, к примеру, работать с одной и той же динамической базой данных или с совместно используемым файлом, изменяя их содержимое;

- модульная конструкция какой-либо системы. Типичным примером может служить микроядерный способ построения операционной системы, когда ее различные части представляют собой отдельные процессы, общающиеся путем передачи сообщений через микроядро.

- просто для удобства работы пользователя, желающего, например, редактировать и отлаживать программу одновременно. В этой ситуации процессы редактора и отладчика должны уметь взаимодействовать друг с другом.

Взаимодействие процессов — это обмен некоторой информацией, в результате чего процессы изменяют свое поведение. Если деятельность процессов не меняется при получении ими любой информации, это означает, что они на самом деле не нуждаются во взаимодействии.

Процессы, которые влияют на поведение друг друга путем обмена информацией, принято называть *кооперативными или взаимодействующими процессами*, в отличие от независимых процессов, которые оказывают друг на друга никакого воздействия и ничего не знают о взаимном сосуществовании в ВС.

Поскольку работа одного процесса не должна приводить к нарушению работы другого процесса, в частности, разделены их адресные пространства и системные ресурсы. Поэтому для обеспечения корректного взаимодействия процессов требуются специальные средства и действия операционной системы. Нельзя просто так взять и поместить значение, вычисленное в одном процессе, в область памяти, соответствующую переменной в другом процессе — для этого нужно предпринять определенные дополнительные организационные усилия.

Категории средств обмена информацией

По объему передаваемой информации и степени возможного воздействия на поведение другого процесса все средства такого обмена можно разделить на три категории:

Сигнальные (signal)

Передается минимальное количество информации — один бит, «да» или «нет».

Используются, как правило, для извещения процесса о наступлении какого-либо события.

Канальные (pipe)

Общение процессов происходит через коммуникационные линии, предоставляемые операционной системой. Объем передаваемой информации в единицу времени ограничен пропускной способностью линий связи. С увеличением количества информации увеличивается и возможность влияния на поведение другого процесса.

Совместно используемая память (shared memory)

Два или более процессов могут совместно использовать некоторую область оперативной памяти (возможно, в разных местах своего адресного пространства).

Созданием совместно используемой памяти занимается операционная система (по запросу процесса). Возможность обмена информацией в этом случае максимальна, как и влияние на поведение другого процесса.

Совместно используемая память представляет собой наиболее быстрый способ взаимодействия процессов в одной вычислительной системе.

Потоки (нити)

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*, как будто они являются чуть ли не обособленными процессами (за исключением общего адресного пространства).

Таблица 2.4. 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.

Подобно традиционному процессу (то есть процессу только с одним потоком), поток должен быть в одном из следующих состояний:

- выполняемый;
- заблокированный;
- готовый;
- завершенный.

Выполняемый поток занимает центральный процессор и является активным в данный момент.

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.

Каждый поток имеет собственный стек. 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.

Начало

Процесс обычно начинается с запуска одного потока. Этот поток, в свою очередь, может создавать новые потоки, вызвав библиотечную процедуру, к примеру **`thread_create()`**. В параметре **`thread_create()`** обычно указывается имя процедуры, запускаемой в новом потоке.

Поток 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 создающему потоку обычно возвращается идентификатор потока, который является именем нового потока.

Завершение

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.

Ожидание

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.

Освобождение процессора

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.

Реализация потоков в пользовательском пространстве

Есть три способа реализации набора потоков — в пользовательском пространстве, в ядре и гибридный.

Первый способ — это поместить весь набор потоков в пользовательском пространстве.

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

Потоки запускаются поверх ***системы поддержки исполнения программ*** (run-time system), которая представляет собой набор процедур, управляющих потоками.

Четыре из них: **`pthread_create()`**, **`pthread_exit()`**, **`pthread_join()`** и **`pthread_yield()`** — мы уже рассмотрели, но обычно в наборе есть и другие процедуры.

Когда потоки управляются в пользовательском пространстве, каждому процессу необходимо иметь собственную ***таблицу потоков***, чтобы отслеживать потоки, имеющиеся в этом процессе.

Эта таблица является аналогом таблицы процессов, имеющейся в ядре, за исключением того, что в ней содержатся лишь свойства, принадлежащие каждому потоку, такие как счетчик команд потока, указатель стека, регистры, состояние и т. д.

Таблица потоков управляется системой поддержки исполнения программ. Когда поток переводится в состояние готовности или блокируется, информация, необходимая для возобновления его выполнения, сохраняется в таблице потоков, точно так же, как ядро хранит информацию о процессах в таблице процессов.

Переключение потоков, по крайней мере на порядок, а может быть, и больше, быстрее, чем перехват управления ядром, что является преимуществом. Мало того, такая организация позволяет каждому потоку иметь собственные настройки алгоритма планирования. Для приложений, которые имеют поток сборщика мусора, есть еще один плюс — им не следует беспокоиться о потоках, остановленных в неподходящий момент. Эти потоки также лучше масштабируются, поскольку потоки в памяти ядра безусловно требуют в ядре пространства для таблицы и стека, что при очень большом количестве потоков может вызвать затруднения.

Однако несмотря на все эти преимущества, у потоков, реализованных на пользовательском уровне, есть ряд существенных проблем. Например, блокирующие системные вызовы приостанавливают выполнение процесса, т.е. всех потоков, до завершения вызова. Например, это может быть взаимодействие с внешними объектами, например, с коммуникационными или файловыми.

А программистам потоки обычно требуются именно в тех приложениях, где они часто блокируются, как, к примеру, в многопоточном веб-сервере.

Еще одна проблема — если начинается выполнение одного из потоков, то никакой другой поток, принадлежащий этому процессу, не сможет выполняться до тех пор, пока первый поток добровольно не уступит центральный процессор.

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

Потоки в POSIX

Чтобы предоставить возможность создания переносимых многопоточных программ, в отношении потоков институтом IEEE был определен стандарт IEEE standard 1003.1c.

Определенный в нем пакет, касающийся потоков, называется Pthreads. Он поддерживается большинством UNIX-систем. В стандарте определено более 60 вызовов функций. В таблице ниже ниже представлен ряд самых основных функций, чтобы дать представление о том, как они работают. Остальные — в курсе СПОВМ.

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

Все потоки ***Pthreads*** имеют определенные свойства. У каждого потока есть свои идентификатор, набор регистров (включая счетчик команд) и набор атрибутов, которые сохраняются в определенной структуре.

Атрибуты включают размер стека, параметры планирования и другие элементы, необходимые при использовании 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.

Чтобы лучше понять, как работают функции пакета ***Pthread***, рассмотрим простой пример, показанный в листинге ниже. Основная программа этого примера работает в цикле столько раз, сколько указано в константе **NUMBER_OF_THREADS** (количество потоков), создавая при каждой итерации новый поток и предварительно сообщив о своих намерениях. Если создать поток не удастся, она выводит сообщение об ошибке и выполняет выход. После создания всех потоков осуществляется выход из основной программы.

При создании поток выводит однострочное сообщение, объявляя о своем существовании, после чего осуществляет выход.

Порядок, в котором выводятся различные сообщения, не определен и при нескольких запусках программы может быть разным.

Пример

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#define NUMBER_OF_THREADS 10

// выводит идентификатор потока, а затем выходит
void *print_hello_world(void *tid) {
    printf("    поток No %lu\n", (size_t)tid);
    pthread_exit(NULL);
}

//---------------------------------------------------------------------
// Основная программа создает 10 потоков, а затем осуществляет выход.
//---------------------------------------------------------------------
int main(int argc, char *argv[]) {

    pthread_t threads[NUMBER_OF_THREADS];
    for (size_t i = 0; i < NUMBER_OF_THREADS; i++) {
        printf("Создание потока No %lu\n", i);
        int status = pthread_create(&threads[i], NULL, print_hello_world, (void *)i);
        if (status != 0) {
            printf("Ахтунг, функция pthread_create вернула код ошибки %d\n", status);
            exit(-1);
        }
    }
    exit(0);
}
```

Компиляция, сборка и запуск

```
$ gcc -Wall -W -Wextra -Wno-unused-parameter -pthread -o threads threads.c
$ ./threads
Создание потока No 0
Создание потока No 1
    поток No 0
Создание потока No 2
    поток No 1
    поток No 2
Создание потока No 3
Создание потока No 4
    поток No 3
Создание потока No 5
    поток No 4
Создание потока No 6
    поток No 5
Создание потока No 7
    поток No 6
Создание потока No 8
    поток No 7
Создание потока No 9
    поток No 8
    поток No 9
```

Потоки C++ — <thread>

Классы

`thread` — класс потоков
`this_thread` — пространство имен

std::thread — класс для представления отдельных потоков выполнения.

Поток выполнения — это последовательность инструкций, которая может выполняться одновременно с другими такими последовательностями в многопоточных средах, в то же время разделяя одно и то же адресное пространство.

Инициализированный объект потока представляет активный поток выполнения.

Такой объект потока может быть присоединен (joinable) и имеет уникальный идентификатор потока (tid).

Созданный с помощью конструктора по умолчанию (неинициализированный) объект потока не может быть присоединен, и его идентификатор является одним общим для всех не присоединяемых потоков.

Присоединяемый поток становится не присоединяемым, если из него было выполнено перемещение или для него вызывается соединение (join) или отсоединение (detach).

Пример

```
#include <iostream>       // std::cout
#include <thread>         // std::thread

void foo() {
  // do stuff...
}

void bar(int x) {
  // do stuff...
}

int main() {

  std::thread first(foo);      // spawn new thread that calls foo()
  std::thread second(bar, 0);  // spawn new thread that calls bar(0)

  std::cout << "main, foo and bar now execute concurrently...\n";

  // synchronize threads:
  first.join();                // pauses until first finishes
  second.join();               // pauses until second finishes

  std::cout << "foo and bar completed.\n";

  return 0;
}
```

std::thread::id — идентификатор потока

```
class thread::id;
```

Значения этого типа идентифицируют поток и возвращаются функциями **thread::get_id** и **this_thread::get_id**.

Значение сконструированного по умолчанию объекта **thread::id** идентифицирует неприсоединяемые потоки и оно одинаково для всех потоков, сконструированных по умолчанию.

Для присоединяемых потоков **thread::get_id** возвращает уникальное значение этого типа, которое не сравнимо ни с каким другим присоединяемым или не присоединяемым потоком.

Некоторые реализации библиотеки могут повторно использовать значение **thread::id** , принадлежа{щее|вшее} завершенному потоку, который больше не может быть присоединен.

Для id перегружен ряд реляционных операторов, не являющихся функциями -членами

```
bool operator== (thread::id lhs, thread::id rhs) noexcept;
bool operator!= (thread::id lhs, thread::id rhs) noexcept;
bool operator< (thread::id lhs, thread::id rhs) noexcept;
bool operator<= (thread::id lhs, thread::id rhs) noexcept;
bool operator> (thread::id lhs, thread::id rhs) noexcept;
bool operator>= (thread::id lhs, thread::id rhs) noexcept;
```

Два объекта **thread::id** считаются эквивалентными, если они представляют один и тот же присоединяемый поток, или если они оба представляют не присоединяемые потоки.

Порядок, устанавливаемый реляционными операторами, является стабильным общим порядком, что позволяет использовать его в качестве ключей для ассоциативных контейнеров.

Конструкторы

(1) по умолчанию

```
thread() noexcept;
```

Создает объект потока, который не представляет никакой выполняющийся поток.

(2) с инициализацией

```
template <class Fn, class... Args>
explicit thread(Fn&& fn, Args&&... args);
```

Создает объект потока, который представляет новый присоединяемый поток выполнения.

Новый поток выполнения вызывает **fn**, передавая **args** в качестве аргументов (используя распадающиеся копии своих ссылок **lvalue** или **rvalue**).

(3) перемещения

```
thread(thread&& x) noexcept;
```

Создает объект потока, который получает поток выполнения, представленный аргументом **x** (если есть). Эта операция никак не влияет на выполнение перемещаемого потока, она просто передает его обработчик. Объект **x** больше не представляет какой-либо поток выполнения.

Объекты потока, прежде чем они будут уничтожены должны быть либо присоединены, либо отсоединены (иначе зомби).

fn — указатель на функцию, указатель на член или любой тип функционального объекта, который можно переместить (т.е. объект, класс которого определяет **operator()**.

Возвращаемое значение (если есть) игнорируется.

args... — аргументы, которые передаются при вызове функции **fn** (если есть). Их типы должны иметь конструкторы перемещения (быть *move-constructible*).

Если **fn** является указателем на член, первым аргументом должен быть объект, для которого этот член определен (или ссылка на него или указатель).

x — объект потока, состояние которого перемещается в созданный объект.

Пример

```
#include <iostream>       // std::cout
#include <atomic>         // std::atomic
#include <thread>         // std::thread
#include <vector>         // std::vector

std::atomic<int> global_counter(0);

void increase_global(int n) {
  for (int i = 0; i < n; ++i) {
    ++global_counter;
  }
}
```

```
int main () {

  std::vector<std::thread> threads;

  std::cout << "increase global counter with 10 threads...\n";
  for (int i = 1; i <= 10; ++i) {
    threads.push_back(std::thread(increase_global, 1000));
  }

  std::cout << "synchronizing all threads...\n";
  for (auto& th : threads) th.join();

  std::cout << "global_counter: " << global_counter << '\n';
}
```

Вывод

```
increase global counter using 10 threads...
synchronizing all threads...
global_counter: 10000
```

std::thread::get_id — вернуть идентификатор потока

```
id get_id() const noexcept;
```

Если объект потока может быть присоединен, функция возвращает значение, однозначно идентифицирующее поток.

Если объект потока не может быть присоединен, функция возвращает созданный по умолчанию объект типа **`thread::id`**.

Пример

```
#include <iostream>       // std::cout
#include <thread>         // std::thread, std::thread::id,
std::this_thread::get_id
#include <chrono>         // std::chrono::seconds

std::thread::id main_thread_id = std::this_thread::get_id();

void is_main_thread() {
  if ( main_thread_id == std::this_thread::get_id() )
    std::cout << "This is the main thread.\n";
  else
    std::cout << "This is not the main thread.\n";
}

int main()  {

  is_main_thread();
  std::thread th(is_main_thread);
  th.join();
}
```

Вывод

```
This is the main thread.
This is not the main thread.
```

std::thread::join — ждать завершения потока

```
void join();
```

Функция возвращается, когда выполнение потока будет завершено.

Она синхронизирует момент возврата из этой функции с завершением всех операций в указанном потоке.

Поток, который вызывает эту функцию, блокируется до тех пор, пока функция, вызванная конструктором указанного потока, не возвратит управление (если к этому моменту это еще не произошло).

После вызова этой функции объект потока становится неприсоединяемым и может быть безопасно уничтожен.

Пример

```
#include <iostream>        // std::cout
#include <thread>          // std::thread, std::this_thread::sleep_for
#include <chrono>          // std::chrono::seconds

void pause_thread(int n) { // засыпаем на n секунд
  std::this_thread::sleep_for(std::chrono::seconds(n));
  std::cout << "пауза в " << n << " с прошла\n";
}

int main()  {
  std::cout << "Spawning 3 threads...\n";
  std::thread t1 (pause_thread, 1);
  std::thread t2 (pause_thread, 2);
  std::thread t3 (pause_thread, 3);
  std::cout << "Порождение потоков завершено. Ждем завершения:\n";
  t1.join();
  t2.join();
  t3.join();
  std::cout << "Все потоки завершились!\n";

  return 0;
}
```

Вывод (после 3-х сек)

```
Spawning 3 threads...
Порождение потоков завершено. Ждем завершения:
пауза в 1 с прошла
пауза в 2 с прошла
пауза в 3 с прошла ended
Все потоки завершились!
```

std::thread::detach — отсоединить поток

```
void detach();
```

Отсоединяет поток, представленный объектом, от вызывающего потока, позволяя им выполняться независимо друг от друга.

Оба потока продолжаются без каких-либо блокировок и синхронизации.

Следует обратить внимание, что когда любой из них завершает выполнение, его ресурсы освобождаются.

После вызова этой функции объект потока становится неприсоединяемым и может быть безопасно уничтожен.

Пример

```
#include <iostream>        // std::cout
#include <thread>          // std::thread, std::this_thread::sleep_for
#include <chrono>          // std::chrono::seconds

void pause_thread(int n) { // засыпаем на n секунд
  std::this_thread::sleep_for(std::chrono::seconds(n));
  std::cout << "пауза в " << n << " с прошла\n";
}


int main() {

  std::cout << "Порождение и отсоединение 3-х потоков...\n";
  std::thread(pause_thread, 1).detach();
  std::thread(pause_thread, 2).detach();
  std::thread(pause_thread, 3).detach();
  std::cout << "Порождение потоков завершено.\n";

  std::cout << "(the main thread will now pause for 5 seconds)\n";

  // дадим отсоединенным потокам 5 с на завершение (не гарантируется!):
  pause_thread(5);

  return 0;
}
```

Вывод

```
Порождение и отсоединение 3-х потоков...
Порождение потоков завершено.
(the main thread will now pause for 5 seconds)
пауза в 1 с прошла
пауза в 2 с прошла
пауза в 3 с прошла
пауза в 5 с прошла
```

std::thread::joinable — проверить, можно ли присоединиться

```
bool joinable() const noexcept;
```

Возвращает, можно ли присоединить объект потока.
К объекту потока можно присоединиться, если он представляет поток выполнения.
Объект потока не может быть присоединен ни в одном из следующих случаев:
- если он был построен по умолчанию;
- если он был перемещен;
- если был вызван один из его членов **`join()`** или **`detach()`**.

Пример

```
#include <iostream>       // std::cout
#include <thread>         // std::thread

void mythread() {
  // do stuff...
}

int main() {

  std::thread foo;
  std::thread bar(mythread);

  std::cout << "Joinable after construction:\n" << std::boolalpha;
  std::cout << "foo: " << foo.joinable() << '\n'; // false (по умолчанию)
  std::cout << "bar: " << bar.joinable() << '\n'; // true

  if (foo.joinable()) foo.join();
  if (bar.joinable()) bar.join();

  std::cout << "Joinable after joining:\n" << std::boolalpha;
  std::cout << "foo: " << foo.joinable() << '\n'; // false (joined)
  std::cout << "bar: " << bar.joinable() << '\n'; // false (joined)

  return 0;
}
```

Вывод

```
Joinable after construction:
foo: false
bar: true
Joinable after joining:
foo: false
bar: false
```

std::this_thread — класс текущего потока

Это пространство имен группирует набор функций, которые обращаются к текущему потоку.

`get_id()` — получить идентификатор потока
`yield()` — уступить другим потокам
`sleep_until()` — заснуть до момента времени
`sleep_for()` — заснуть на время

std::this_thread::yield — уступить другим потокам

```
void yield() noexcept;
```

Вызывающий поток уступает процессор, предлагая реализации возможность выполнить перепланирование .

Эта функция должна вызываться, когда поток ожидает продолжение других потоков без блокирования.

Пример

```
#include <iostream>       // std::cout
#include <thread>         // std::thread, std::this_thread::yield
#include <atomic>         // std::atomic

std::atomic<bool> ready(false); // флаг на старт

void count1m(int id) {

  while (!ready) {              // ждать, покак main() не разрешит начать ...
    std::this_thread::yield(); // уступить другим
  }
  for (volatile int i = 0; i < 1000000; ++i) {}
  std::cout << id;
}

int main () {

  std::thread threads[10]; // массив на 10 потоков (tid)
  std::cout << "гонки 10-и потоков, которые считают до 1 миллиона:\n";
  for (int i = 0; i < 10; ++i) threads[i]=std::thread(count1m, i);
  ready = true;                                                  // Старт!!!
  for (auto& th : threads) th.join(); // ждем всех
  std::cout << '\n';

}
```

Вывод

```
гонки 10-и потоков, котороые считают до 1 миллиона:
7561403928
```

std::this_thread::sleep_until — заснуть до момента времени

```
emplate <class Clock, class Duration>
void sleep_until(const chrono::time_point<Clock, Duration>& abs_time);
```

Блокирует вызывающий поток до **abs_time**.

Выполнение текущего потока останавливается, по крайней мере, до **abs_time**, в то время как другие потоки могут продолжать работу.

abs_time — момент времени, когда вызывающий поток должен возобновить свое выполнение.

Следует заметить, что операции управления многопоточностью могут вызывать определенные задержки сверх этого.

time_point — это объект, представляющий конкретное абсолютное время.

Пример

```
#include <iostream>       // std::cout
#include <iomanip>        // std::put_time
#include <thread>         // std::this_thread::sleep_until
#include <chrono>         // std::chrono::system_clock
#include <ctime>          // std::time_t, std::tm, std::localtime, std::mktime

int main() {

  using std::chrono::system_clock; //
  std::time_t tt = system_clock::to_time_t(system_clock::now());

  struct std::tm *ptm = std::localtime(&tt);
  std::cout << "Текущее время: " << std::put_time(ptm, "%X") << '\n';

  std::cout << "Ожидание начала следующей минуты ...\n";
  ++ptm->tm_min;
  ptm->tm_sec = 0;
  std::this_thread::sleep_until(system_clock::from_time_t(mktime(ptm)));

  std::cout << std::put_time(ptm,"%X") << "\n";

  return 0;
}
Текущее время: 16:33:18
Ожидание начала следующей минуты ...
16:34:00
```

std::this_thread::sleep_for — заснуть на время

```
template <class Rep, class Period>
void sleep_for(const chrono::duration<Rep, Period>& rel_time);
```

Блокирует выполнение вызывающего потока в течение периода времени, указанного аргументом **rel_time**.

Выполнение текущего потока останавливается, пока не пройдет хотя бы **rel_time**.

Остальные потоки продолжают свое выполнение.

rel_time — промежуток времени, по истечении которого вызывающий поток должен возобновить свое выполнение.

Следует заметить, что операции управления многопоточностью могут вызывать определенные задержки сверх этого.

duration — это объект, который представляет точное относительное время.

Пример

```
#include <iostream>       // std::cout, std::endl
#include <thread>         // std::this_thread::sleep_for
#include <chrono>         // std::chrono::seconds

int main() {

  std::cout << "Обратный отсчет:\n";

  for (int i = 10; i > 0; --i) {
    std::cout << i << std::endl;
    std::this_thread::sleep_for(std::chrono::seconds(1));
  }
  std::cout << "Старт!\n";

  return 0;
}
```

Вывод

```
10
9
...
2
1
Старт!
```