Содержание

1 Нити **1**
1.1 Нити и стандартная библиотека Си . . . 3
1.2 Нити в языках программирования . . . 4

2 Введение в библиотеку pthread **5**
2.1 Модель планирования нитей в pthread . . . 5
2.2 Завершение процесса и нити . . . 6
2.3 Идентификатор нити . . . 7
2.4 Главная функция нити . . . 8
2.5 Локальная для нити память . . . 9
2.6 Запуск и ожидание завершения нити . . . 10
2.7 Атрибуты создания нити . . . 12
2.7.1 Управление стеком . . . 13
2.7.2 Управление фоновым режимом . . . 14
2.8 Принудительное завершение нити . . . 14
2.9 Нити и сигналы . . . 16
2.10 Нити и fork . . . 18

3 Синхронизация в pthread **18**
3.1 Мьютексы . . . 19
3.2 Мьютексы в pthread . . . 20
3.3 Рекурсивные мьютексы . . . 21
3.4 Типы мьютексов . . . 23
3.5 Условные переменные . . . 24
3.6 Условные переменные pthread . . . 27

4 Классические задачи синхронизации **27**
4.1 Обедающие философы . . . 28
4.2 Барьер . . . 30
4.3 Читатели-писатели . . . 31

1. Нити

Нить (также используется термины «поток выполнения», «тред», «легковесные процессы»; англ. *thread* — нить) — субъект распределения процессорного времени ядром операционной системы. Планировщик (scheduler) ядра операционной системы распределяет процессорное время всех процессоров и ядер компьютера между нитями, находящимися в очереди нитей, готовых к выполнению.

Процесс — субъект распределения всех ресурсов, а не только процессорного времени ядром операционной системы. По запросу процесса ядро выделяет ему различные ресурсы, например, виртуальное адресное пространство, файловые дескрипторы и т. д. В частности, по запросу процесса ядро может выделить процессу больше процессорного времени, создав новую нить в рамках этого процесса. При создании процесса ему выделяются начальные ресурсы, которыми процесс в дальнейшем может распоряжаться.

При создании процесса создаётся одна «главная» нить, в рамках которой выполняется функция `main`. Если процесс не создает других нитей, удобно считать, что планировщик выделяет время такому процессу в целом, а не главной нити этого процесса. Поэтому, когда говорят об однонитевом процессе, обычно не выделяют понятие главной нити процесса.

Все нити, созданные в рамках одного процесса, используют ресурсы, выделенные этому процессу (за исключением процессорного времени, которое выделяется каждой нити по отдельности). Все нити разделяют общее виртуальное адресное пространство, общий набор открытых файловых дескрипторов и т. д. Так, новые страницы виртуального адресного пространства, добавленные в одной нити, могут использоваться во всех нитях, как и файловый дескриптор, открытый в одной нити, можно использовать во всех нитях.

Ядро операционной системы хранит для каждой нити следующие ее уникальные ресурсы:

- Контекст нити, который представляет собой содержимое регистров центрального процессора в момент снятия нити с выполнения на процессоре. Контекст нити необходим, чтобы вернуться к выполнению с точки, в которой выполнение нити было прервано.

- Маска блокируемых сигналов для нити и множество сигналов, ожидающих доставки. Процесс может управлять маской (множеством) блокируемых сигналов как на уровне процесса целиком, так и на уровне каждой нити процесса.

- Настройки планировщика (тип планирования, приоритет, привязка к ядрам процессора).

Каждая нить работает со своим стеком для организации хранения локальных переменных и вызова подпрограмм. Но стеки всех нитей, как и стек главной нити находятся в едином адресном пространстве процесса, и доступны всем нитям. Поэтому одна нить может передать другой нити адрес переменной, расположенной в своем стеке, и вторая нить сможет работать с этой переменной, при условии, что переменная не будет уничтожена.

При создании главной нити ядро операционной системы выделяет только необходимую виртуальную память для стека главной нити. В стек главной нити помещаются аргументы командной строки и переменные окружения, после чего управление передается в точку входа процесса. В дальнейшем ядро автоматически расширяет стек главной нити, когда она выходит за пределы текущего выделенного стека в сторону роста стека (на большинстве современных процессорных архитектур — в сторону уменьшения адреса). Автоматический рост стека главной нити процесса останавливается, когда достигается либо ограничение размера стека (на Linux по умолчанию установлено ограничение размера стека равное 8MiB), либо ограничение размера виртуального адресного пространства, либо заканчивается виртуальная или физическая память.

В отличие от главной нити все нити, созданные процессом, получают стек фиксированного размера при создании нити. Память под стек нити выделяется один раз, и в дальнейшем размер стека нити нельзя изменить. Поэтому при создании нити важно задать правильный размер стека: не слишком большой, чтобы эффективно использовать виртуальное адресное пространство, но и не слишком маленький, чтобы стека хватило для всех вызываемых функций в этой нити.

Каждая нить имеет доступ к глобальным переменным процесса, но кроме того, каждая нить может работать с локальными для нити (thread-local) переменными. Локальные для нити переменные являются по сути глобальными, то есть размещаются не на стеке и не в области динамических данных, но создаются в момент создания нити и уничтожаются в момент завершения нити. Поскольку адресное пространство всех нитей общее, некоторая нить может работать с thread-local переменными другой нити, при условии, что она получит каким-нибудь образом адрес переменных.

1.1. Нити и стандартная библиотека Си

Многие функции стандартной библиотеки Си изменяют состояние объектов, находящихся в глобальной или динамической области памяти. Например, функция `malloc` выделяет память в динамической области, при этом модифицируются вспомогательные структуры (списки блоков памяти), используемые функциями работы с динамической памятью. Функция `printf` модифицирует выходной буфер стандартного потока вывода. Если модификация состояния буфера будет выполняться одновременно в двух нитях, буфер может оказаться в нецелостном состоянии (ошибка race condition). Поэтому для удобства программиста большинство функций стандартной библиотеки языка Си корректно работают, если вызываются из параллельно работающих нитей одновременно. Сначала выполнится функция в одной нити, затем в другой нити.

Если функция корректно работает, когда вызывается одновременно из нескольких нитей, такая функция называется *потокобезопасной* (thread-safe). Потокобезопасность функции обязательно указывается в ее документации.

Некоторые функции не являются потокобезопасными. Как правило, такие функции используют в своей работе или возвращают указатель на значение, расположенное в глобальной области видимости. Например, функция `localtime` преобразовывает время в формате Unix во время в григорианском календаре и возвращает указатель на структуру `struct tm`. Эта структура размещается в глобальной области памяти. Таким образом, если две нити одновременно вызывают и используют указатель, возвращенный этой функцией, будет получен неверный результат работы. В случае функции `localtime` стандартная библиотека предоставляет функцию `localtime_r`, которая не модифицирует глобальные переменные и поэтому является thread-safe. Для каждой функции стандартной библиотеки, которая не является потокобезопасной существует ее потокобезопасный аналог.

Это не значит, что если функция не является потокобезопасной, она никогда не должна использоваться в многопоточных программах. В случае использования непотокобезопасной функции контроль за тем, что в нескольких нитях одновременно не происходит модификация состояния одного и того же объекта посредством непотокобезопасных функций возлагается на программиста. Например, если работа с некоторым файлом с помощью дескриптора потока `FILE*` идет строго в одной нити, нет необходимости использовать потокобезопасные функции ввода-вывода.

Обеспечение потокобезопасности требует дополнительных накладных расходов на каждый библиотечный вызов. Например, рассмотрим цикл

```
int c;
while ((c = getc(f))) {
}
```

в этом цикле выполняется посимвольное чтение из дескриптора потока `f`. Функция

`getc` — потокобезопасная, и при каждом вызове она выполняет блокировку переменной дескриптора потока `f`. Если программа выполняется только в одну (главную) нить, или работа с дескриптором потока `f` ведется только в одной нити, можно использовать функцию `getc_unlocked`, которая не блокирует дескриптор потока и поэтому не является потокобезопасной.

```
int c;
while ((c = getc_unlocked(f))) {
}
```

За счет отсутствия блокировок второй вариант выполняет чтение из файла примерно в пять раз быстрее, чем вариант с использованием функции `getc`.

Обратите внимание, что свойство потокобезопасности функции (thread safety) не имеет никакого отношения к свойству асинхронной безопасности (async-signal safety). Асинхронная безопасности позволяет вызывать функции из обработчиков сигналов, а потокобезопасность позволяет вызывать функции из параллельно выполняющихся нитей. Например, функция выделения динамической памяти `malloc` является потокобезопасной, но не является асинхронно-безопасной.

1.2. Нити в языках программирования

Поддержка нитей появлялась в относительно новых языках программирования, таких как Java, C# и другие одновременно с выходом первых версий языка. В них изначально была заложена некоторая модель многопоточного программирования отраженная и в ядре языка, и в стандартной библиотеке.

Язык Си развивался параллельно с развитием UNIX-подобных операционных систем. Каждая операционная система, которая реализовывала поддержку нитей, предоставляла библиотеку для использования нитей в языке Си. Языковая поддержка нитей реализовывалась расширениями языка Си, специфичными для компилятора, операционной системы, аппаратной платформы. В интерфейсе прикладного программирования Windows WinAPI поддержка нитей была встроена с самых первых версий.

В середине 90-х годов была проведена работа по стандартизации интерфейса прикладного программирования нитей в UNIX-подобных системах и появился стандарт POSIX Thread (pthread), используемый до настоящего времени. Реализации этого стандарта доступны на всех операционных системах, поддерживающих POSIX, а это Linux, разные версии BSD, Solaris и т. д.

Стандарт языка Си 99 года (C99) не содержал описания поддержки нитей. Программа на Си должна была опираться на средства реализации нитей, предоставляемые платформой, для которой она предназначена. Стандарт языка Си++ 98 года (C++98) также не содержал поддержки нитей, программа на Си++ должна была использовать библиотеки языка Си для работы с нитями.

Стандарты языков Си и Си++, выпущенные в 2011 году, изменили эту ситуацию. В них содержалось описание модели памяти программы на Си или Си++. Оба стандарта описали общую модель памяти, отличающуюся только в языково-специфичных деталях реализации. Так, в C11 для описания переменной, локальной для нити используется ключевое слово `_Thread_local`, а в C++11 — ключевое слово `thread_local`. Каждый из стандартов добавил в стандартную библиотеку языка средства работы с нитями, которые должны одинаково работать на всех платформах, поддерживающих нити.

В стандарте языка C11 поддержка нитей отмечена как необязательная. То есть, компилятор и среда выполнения C11 может реализовывать, а может и не реализовывать нити в стандарте C11. Например, компилятор gcc реализует языковую поддержку нитей и модели памяти стандарта C11 (`_Thread_local`, `<stdatomic.h>`, `_Atomic` и соответствующие операции), а стандартная библиотека Linux glibc до сих пор (середина 2017 года) не поддерживает заголовочные файлы и библиотечные функции, описанные в стандарте C11. Причина, скорее всего, в том, что ни у кого не возникла потребность в использовании нитей C11. Как известно, язык C на компиляторах Windows является второстепенным, а на POSIX системах всем достаточно интерфейса POSIX Thread.

С другой стороны, поддержка нитей стандарта C++11 была реализована достаточно быстро. C++ предлагает достаточно удобный (существенно удобнее POSIX Thread) интерфейс для работы с нитями, предоставляет высокий уровень абстракций. Язык C++ является одним из основных при разработке и на Windows, и на POSIX-системах и повсеместно используется крупнейшими корпорациями, которые во многом и контролируют стандартизацию C++. Интерфейс нитей C++11 будет отдельно нами рассмотрен.

2. Введение в библиотеку pthread

Библиотека POSIX Thread (pthread) — стандартный интерфейс работы с нитями в операционных системах, поддерживающих стандарт POSIX (то есть в UNIX-подобных операционных системах). Типы данных, константы, прототипы функций определяются в заголовочном файле `<pthread.h>`, который необходимо подключать в начале программы.

```
#include <pthread.h>
```

Тела функций библиотеки pthread не находятся в стандартной библиотеке libc, а вынесены в отдельный бинарный файл. Поэтому для подключения библиотеки необходимо в строке компиляции программы передать компоновщику опцию `-lpthread`, или лучше использовать специальный флаг `-pthread` компилятора gcc.

```
gcc -pthread -O2 program.c -o program
```

Идентификаторы функций, констант, типов данных и переменных, определенные в библиотеке pthread начинаются с префикса `pthread_` или `PTHREAD_`.

2.1. Модель планирования нитей в pthread

Выше было дано определение нити как единицы планирования времени процессора планировщиком ядра операционной системы. Стандарт pthread, тем не менее, не требует такого способа реализации. Допускается реализация, в которой нити не будут выполняться параллельно, либо в которой передача выполнения от нити к нити будет выполняться только явно. В частности, механизм сопрограмм (coroutines) может быть реализован в терминах Pthread API.

Реализация pthread может выполнять все операции по созданию, уничтожению и поддержке выполнения нитей только средствами пользовательского режима, без привлечения ядра ОС с помощью системных вызовов. В этом случае с точки зрения ядра

процессорное время выделяется только одному потоку выполнения процесса, который своими силами распределяет это процессорное время между своими нитями. Такая модель выполнения называется *модель n:1*, иногда ее называют «зеленые» нити (green threads). В этой модели отсутствуют накладные расходы на системные вызовы, на планирование нитей внутри ядра ОС. Кроме того, нити могут выполняться в любой желаемой для приложения последовательности. Недостатком модели является невозможность использования всех ядер многоядерной/многопроцессорной системы. Поскольку ядро операционной системы знает только об одном потоке выполнения, ядро ОС и будет выделять время только на одном процессоре.

Если реализация pthread для создания каждой нити использует системные вызовы, предоставляемые ядром операционной системы, и нити процесса планируются на уровне планировщика ядра операционной системы, говорят о *модели 1:1* выполнения нитей. Такая модель реализована в большинстве современных операционных систем, например, в Linux. В дальнейшем, рассматривая средства библиотеки pthread, мы будем подразумевать именно такую модель выполнения.

Возможна и гибридная модель (*модель n:m*), в которой нити объединяются в группы. Внутри группы нитей они планируются на пользовательском уровне, а уже группы нитей планируются ядром операционной системы. Такая модель реализована в Solaris.

2.2. Завершение процесса и нити

Процесс, в рамках которого выполняются несколько нитей, завершается в одном из следующих случаев:

- Завершились все нити процесса.
- Какая-либо нить вызвала явно или неявно библиотечные функции или системные вызовы завершения процесса: `exit`, `_exit`, `_Exit`.
- В процесс поступил сигнал, обрабатываемый по умолчанию, обработка которого по умолчанию вызывает завершение процесса (таковы большинство сигналов: SIGINT, SIGTERM, SIGKILL, SIGSEGV и т. д.).

Если процесс завершается, то завершаются и все выполняющиеся в текущий момент нити. Из-за этого выполнение нити может быть завершено асинхронно, например, при обновлении нитью информации, доступной извне процесса, то есть в разделяемой памяти или в файловой системе. Поэтому завершение процесса при работающих нитях может приводить к тому, что данные на внешних носителях или разделяемой памяти могут остаться в нецелостном состоянии.

Когда выполнение функции `main` завершается «естественным» путем по достижению конца тела функции `main` или с помощью оператора `return`, неявно выполняется функция `exit`. То есть завершение `main` приводит к завершению всего процесса, включая все выполняющиеся в данный момент нити.

Однако, если функция `main` вызывает функцию `pthread_exit`, которая завершает не весь процесс, а только одну текущую нить, то выполнение процесса продолжится до наступления одного из условий, описанных выше.

Нить завершается в одном из следующих случаев:

- По достижению конца тела функции или с помощью оператора `return` завершилась главная функция нити.

- Нить вызвала функцию `pthread_exit`.
- Выполнение нити было прекращено какой-то другой нитью с помощью библиотечной функции `pthread_cancel`.

Обратите внимание, что сигналы доставляются в процесс целиком, а не в отдельную нить. Поэтому, если какая-либо нить выполнит некорректную операцию с памятью, сигнал `SIGSEGV` будет доставлен в процесс в целом, что вызовет завершение всего процесса, а не одной «аварийной» нити. Исключения языка C++, выбрасываемые в нити, могут по-разному влиять на выполнение всего процесса и отдельной нити. Об этом будет рассказано в соответствующем разделе.

2.3. Идентификатор нити

Каждая нить в процессе идентифицируется значением, представимым библиотечным типом `pthread_t`. Например, следующий фрагмент получает идентификатор текущей исполняющейся нити с помощью функции `pthread_self`:

```
pthread_t self = pthread_self();
```

Тип `pthread_t` может реализовываться по-разному. Например, на Linux это — указатель (opaque pointer) в виртуальном адресном пространстве процесса, по которому располагается служебная структура описателя нити. Со значениями типа `pthread_t` определена только операция копирования при инициализации или присваивания. Мы можем использовать значение `0` или `NULL` для начальной инициализации переменной типа `pthread_t` для указания «пустого» идентификатора нити.

Новое значение идентификатора нити создается при создании новой нити с помощью функции `pthread_create`. После создания нити можно использовать ее идентификатор для выполнения операций с этой нитью. Идентификатор нити может копироваться, передаваться в качестве аргумента в функции, сохраняться в глобальные массивы и использоваться другими нитями чем та нить, которая создала новую нить с этим идентификатором.

Когда нить завершается, область памяти, занимаемая ее описателем, указатель на которую соответствует значению идентификатора нити, не освобождается немедленно. Это сделано, чтобы завершившаяся нить могла бы передать какой-то другой нити результат своей работы. Функция `pthread_join` должна использоваться, чтобы дождаться завершения работы указанной нити, получить результат, переданный нитью. Функция `pthread_join` освобождает память, занимаемую описателем нити, делая значение идентификатора нити некорректным. Например, если попытаться применить `pthread_join` второй раз к одной и той же нити, это, скорее всего, приведет к обращению в память по «висячему» указателю и неопределенному поведению программы. Функция `pthread_join` — это единственный способ полностью освободить ресурсы, которые использовались нитью. Если не выполнять `pthread_join` для завершившихся нитей, память постепенно будет засоряться неосвобожденными блоками, то есть память постепенно будет «утекать».

Однако нить может быть запущена или в процессе работы переведена в «фоновый» режим (detached state). Если нить выполняется в фоновом режиме, память, занимаемая описателем нити, освобождается немедленно при завершении нити. К фоновой нити нельзя применить операцию `pthread_join`. Таким образом, значение идентификатора фоновой нити может стать недопустимым в любой момент времени, что

означает, что идентификатор фоновой нити вообще нельзя использовать нигде, кроме в самой этой нити.

2.4. Главная функция нити

Главная функция нити выполняет роль, аналогичную функции `main`. При создании нити указывается имя функции, которая будет запущена в новой нити. Завершение этой функции означает завершение нити. Главная функция нити имеет следующий прототип:

```
void *thread_func(void *ptr);
```

Входным параметром главной функции нити передается указатель на область памяти со входными данными, подготовленными для нити. Поскольку все нити процесса работают в одном адресном пространстве, передаваться могут данные любого типа: целые числа, вещественные числа, указатели, структуры. В простых случаях можно в указателе передать целое значение. Для этого при передаче параметра в нить нужно преобразовать целый тип в указательный, а в функции нити выполнить обратное преобразование:

```
void *thread_func(void *ptr)
{
    int serial = (intptr_t) ptr;
}
```

Здесь преобразование выполняется в два шага: `void*` - `intptr_t` - `int`. Тип `intptr_t` определен в заголовочном файле `<stdint.h>`. Это — целочисленный тип, размер которого совпадает с размером указателя, таким образом преобразования из указателя в этот тип и из этого типа в указатель выполняются без потери битовой точности. Если не использовать промежуточное преобразование:

```
void *thread_func(void *ptr)
{
    int serial = ptr;   // WRONG!
}
```

компиляция программы завершится с ошибкой на 64-битной платформе из-за того, что размер целого типа `int` не совпадает с размером указательного типа `void*`. Мы предполагаем, что размер целого типа не больше размера указательного типа.

Если в нить не требуется передать никакого значения, можно передать указатель NULL.

Главная функция нити возвращает указатель типа `void*`, который может быть получен в нити, которая выполняет ожидание завершения этой нити. Работать с возвращаемым значением можно аналогично передаваемому значению. Если главная функция нити не передает результат работы ожидающей нити, в качестве возвращаемого значения можно использовать NULL.

При возврате значения учитывайте время жизни переменных:

```
void *thread_func(void *ptr)
{
    long long result = 0;
    // ...
```

```
    return &result;     // WRONG!
}
```

В этом примере допущена грубая ошибка: возвращается адрес локальной переменной. Локальная переменная будет уничтожена после завершения функции, и ожидающая нить получит висящий указатель.

Аналогично в примере:

```
_Thread_local long long result;
void *thread_func(void *ptr)
{
    return &result;     // WRONG!
}
```

возвращается указатель на блок локальных для нити данных (thread-local storage), который будет освобожден после завершения функции нити, и ожидающая нить получит висящий указатель.

2.5. Локальная для нити память

В обычной однонитевой программе различаются три класса памяти переменных: глобальный, автоматический и динамический. Глобальные переменные создаются в момент запуска процесса и уничтожаются в момент завершения процесса, автоматические переменные создаются при входе в блок и уничтожаются при выходе из блока, а динамические переменные создаются и уничтожаются по явному запросу программиста.

В многонитевой программе возникает еще один класс памяти: локальная для нити память. Локальные для нити переменные создаются при создании нити и уничтожаются при завершении нити. Например,

```
_Thread_local int var = 10;

void func1(void)
{
    printf("%d\n", ++var);
}

void *thread_func(void *ptr)
{
    printf("%d\n", ++var);
    func1();
}
```

Класс локальной для нити памяти обозначается в языке Си с помощью ключевого слова `_Thread_local`, а в языке Си++ с помощью ключевого слова `thread_local`. Переменная `var` в данном примере видна для всех функций, определенных ниже этой переменной, то есть и в функции `func1`, и в функции `thread_func`.

Каждый раз, когда создается нить будет выделена память под переменную `var`, переменная будет проинициализирована значением 10. Соответственно, каждая нить сначала выведет значение 11, затем значение 12. Все нити разделяют общее адресное пространство, поэтому переменная `var` в каждой нити будет размещаться по уникаль-

ному адресу. Если передать адрес локальной для нити переменной в другую нить, переменная может быть прочитана и модифицирована другой нитью.

Когда нить завершает работу, память под переменную `var` освобождается. Если указатель на уничтоженную переменную был где-то сохранен, попытка обращения по такому указателю приведет к undefined behavior, который, скорее всего, выразится в ошибке доступа к памяти и аварийном завершении программы.

2.6. Запуск и ожидание завершения нити

Новая нить создается с помощью функции `pthread_create`, имеющей следующий прототип:

```
int pthread_create(pthread_t *thread,
                   const pthread_attr_t *attr,
                   void *(*start_routine) (void *),
                   void *arg);
```

Параметр `thread` — это адрес переменной, в которую будет записан идентификатор созданной нити. Параметр `attr` — это дополнительные параметры создания нити. Они будут рассмотрены далее. Если нить создается с параметрами по умолчанию, в качестве значения `attr` можно передавать NULL. Параметр `start_routine` — это главная функция нити, а параметр `arg` — это дополнительный параметр, передаваемый в главную функцию нити.

Функция `pthread_create` возвращает 0 в случае успешного завершения, тогда переменная, на которую указывает `thread` будет содержать идентификатор созданной нити. В случае ошибки функция возвращает код ошибки. Переменная `errno` не модифицируется.

Например, следующий вызов создает новую нить с параметрами по умолчанию. В новой нити запускается функция `thread_func`, которой в качестве параметра передается NULL. В случае ошибки на стандартный поток ошибок выводится сообщение об ошибке.

```
    pthread_t id;
    int err;
    if ((err = pthread_create(&id, NULL, thread_func, NULL))) {
        fprintf(stderr, "cannot create new thread: %s\n", strerror(err));
    }
```

Если требуется создать несколько нитей в цикле, передавая каждой нити ее порядковый номер, нельзя передавать адрес переменной цикла:

```
    pthread_t ids[N];
    for (int i = 0; i < N; ++i) {
        pthread_create(&ids[i], NULL, thread_func, &i);  // WRONG!
    }
```

Все созданные нити получат адрес одной и той же локальной переменной `i`, которая к тому же параллельно изменяется в главной программе. Возникает ситуация Data Race, и поведение программы неопределено.

Правильно в этом случае либо создавать для каждой нити свою переменную с ее порядковым номером:

```
pthread_t ids[N];
int nums[N];
for (int i = 0; i < N; ++i) {
    nums[i] = i;
    pthread_create(&ids[i], NULL, thread_func, &nums[i]); // каждой нити
передаем разный адрес
}
```

либо передавать в нить порядковый номер преобразованием его к указателю.

```
pthread_t ids[N];
for (int i = 0; i < N; ++i) {
    pthread_create(&ids[i], NULL, thread_func, (void*) (intptr_t) i); //
передаем номер по значению
}
```

Относительный порядок выполнения нитей после создания новой нити неопределен. Нити, созданные в цикле, могут выполняться в произвольном порядке, как и сама нить, создающая новые нити.

Дождаться завершения нити и получить результат ее выполнения можно с помощью функции `pthread_join`.

```
int pthread_join(pthread_t thread, void **retval);
```

Обратите внимание, что в функцию `pthread_create` первым параметром передается адрес переменной, в которую записывается идентификатор новой нити, а в функцию `pthread_join` первым параметром передается значение идентификатора нити, окончания которой необходимо дождаться.

Текущая нить приостанавливает свое выполнение до завершения нити с идентификатором, переданным в параметре `thread`. Если целевая нить уже завершилась, функция возвращается немедленно. По адресу, переданному в параметре `retval` будет записано значение, возвращенное из функции нити. Поскольку функция нити возвращает значение типа `void*`, то и параметр `retval` должен быть указателем на переменную типа `void*`, отсюда двойной указатель. Если возвращаемое значение функции не нужно, в качестве параметра `retval` можно передать NULL, но тогда возвращаемое значение потеряется. Это может привести к утечке памяти, если возвращался указатель в динамическую память.

Если целевая нить была завершена принудительно с помощью вызова функции `pthread_cancel`, возвращаемое значение нити будет равно `PTHREAD_CANCELED`.

Если одновременно несколько нитей пытаются выполнить join для одной и той же нити, или для этой нити уже была выполнена операция join, или нить не является стандартной (joinable) нитью, а является фоновой, поведение функции `pthread_join` неопределено (undefined behavior). Скорее всего, программа завершится из-за некорректной работы с памятью.

Функция `pthread_join` возвращает 0 в случае успеха и код ошибки в случае ошибки. Например, ошибочной является ситуация, когда нить ждет сама себя, или две нити ждут друг друга. В остальном большинство ошибочных ситуаций при вызове функции `pthread_join` приводят к неопределенному поведению, как описано выше.

Все нити в процессе являются равноправными. Иерархия «отец-сын» отсутствует. Поэтому, например, нить, порожденная другой нитью, может вполне ждать завершения нити, которая ее породила, либо две нити-сестры могут ждать друг друга и т. д.

Функция pthread_join позволяет дождаться одну конкретную нить. Средствами pthread невозможно подождать любую из выполняющихся нитей. Если необходимо ждать любую нить, нужно пересматривать логику программы чтобы необходимость ожидания любой нити не возникала.

Напомним, что процесс завершается, когда завершается функция main. Поэтому следующая функция main будет неправильной:

```
int main(void)
{
    pthread_t ids[N];
    for (int i = 0; i < N; ++i) {
        pthread_create(&ids[i], NULL, thread_func, (void*) (intptr_t) i);
    }
    // WRONG!
}
```

Процесс завершится как только в функции main будут созданы все N нитей. Скорее всего при этом некоторые нити даже не успеют выполниться.

Правильный вариант программы — дождаться завершения всех нитей:

```
int main(void)
{
    pthread_t ids[N];
    for (int i = 0; i < N; ++i) {
        pthread_create(&ids[i], NULL, thread_func, (void*) (intptr_t) i);
    }
    for (int i = 0; i < N; ++i) {
        pthread_join(ids[i], NULL); // возвращаемое значение игнорируется
    }
}
```

Или завершить функцию main как нить:

```
int main(void)
{
    pthread_t ids[N];
    for (int i = 0; i < N; ++i) {
        pthread_create(&ids[i], NULL, thread_func, (void*) (intptr_t) i);
    }
    pthread_exit(NULL);
}
```

2.7. Атрибуты создания нити

При создании нити с помощью функции pthread_create вторым аргументом attr передаются дополнительные атрибуты создания нити. Чтобы передать дополнительные атрибуты нужно проинициализировать структуру с помощью pthread_attr_init, затем настроить требуемые параметры, затем выполнить pthread_create, затем освободить память, занимаемую атрибутами.

```
    pthread_attr_t attr;      // неинициализированная структура атрибутов
    pthread_attr_init(&attr); // инициализируем структуру
```

```
    // здесь настраиваем атрибуты
    pthread_t tid;
    pthread_create(&tid, &attr, func, NULL); // создаем нить с атрибутами attr
    pthread_attr_destroy(&attr); // освобождаем память атрибутов
```

Подготовленную структуру атрибутов можно использовать для создания нескольких нитей, так как функция `pthread_create` не модифицирует структуру атрибутов. Структуру атрибутов можно уничтожить сразу же после создания нити, если такие атрибуты в дальнейшем не потребуются.

2.7.1. Управление стеком

Как было сказано выше, при создании каждой нити создается стек нити фиксированного размера. Стек нити не расширяется и не перемещается, пока нить выполняется. Если стек нити будет слишком велик, то много виртуального адресного пространства будет использовано неэффективно. Если стек нити окажется слишком мал, при выполнении нити возникнет переполнение стека, и весь процесс завершится аварийно. По умолчанию при создании нити размер стека берется равным размеру ограничения стека для процесса в целом. Ограничения процесса в целом можно посмотреть, выполнив команду Unix `ulimit -a`. При выполнении команды будет получен примерно следующий результат:

```
core file size          (blocks, -c) 0
data seg size           (kbytes, -d) unlimited
scheduling priority             (-e) 0
file size               (blocks, -f) unlimited
pending signals                 (-i) 24076
max locked memory       (kbytes, -l) 64
max memory size         (kbytes, -m) unlimited
open files                      (-n) 1024
pipe size            (512 bytes, -p) 8
POSIX message queues     (bytes, -q) 819200
real-time priority              (-r) 0
stack size              (kbytes, -s) 8192
cpu time               (seconds, -t) unlimited
max user processes              (-u) 24076
virtual memory          (kbytes, -v) unlimited
file locks                      (-x) unlimited
```

В строке `stack size` выводится ограничение стека для процесса в Linux. Как видно, что по умолчанию это ограничение выставлено в 8MiB. Поэтому при создании нити с атрибутами по умолчанию будет создан стек размером 8MiB.

Чтобы изменить размер стека нити используется функция `pthread_attr_setstacksize`.

```
int pthread_attr_setstacksize(pthread_attr_t *attr, size_t stacksize);
```

Например, следующий вызов установит размер стека нити в 1MiB.

```
    pthread_attr_setstacksize(&attr, 1024 * 1024);
```

Как и все функции в библиотеке pthread функция возвращает 0 при успешном выполнении и ненулевое значение в случае ошибки. Одна из возможных причин ошибки: слишком маленький запрошенный размер стека. То есть нельзя задать размер стека меньше системного ограничения. Системное ограничение можно получить с помощью системного вызова `sysconf`:

```
#include <unistd.h>

    size_t min_stack_size = sysconf(_SC_THREAD_STACK_MIN);
```

На Linux минимальный размер стека равен (почему-то) 16KiB. Поэтому установить размер стека нити равный минимальному можно следующим образом:

```
    pthread_attr_setstacksize(&attr, sysconf(_SC_THREAD_STACK_MIN));
```

Каков бы ни был размер стека, выделенного одной нити, возможна ошибка переполнения стека. Например, если нить войдет в бесконечную рекурсию. Чтобы ошибка бесконечной рекурсии приводила к аварийной остановке программы внизу каждого стека нити добавляется специальная страница виртуальной памяти, недоступная ни на запись, ни на чтение. Она называется «защитная страница» (guard page). Когда выполняющаяся нить исчерпывает свой стек, значение регистра указателя стека попадает в диапазон адресов защитной страницы, обращение к памяти вызывает ошибку доступа к памяти (segmentation fault) и аварийное завершение программы.

Если защитная страница отсутствует, возможна ситуация, что стеки двух нитей будут размещаться в виртуальной память один за другим. Если стеки нити, который расположен «выше» в виртуальной памяти, переполнится, значение регистра указателя стека попадет в диапазон адресов стека другой нити, и стек другой нити будет испорчен.

Управлять размером защитной страницы можно с помощью библиотечной функции `pthread_attr_setguardsize`. По умолчанию в Linux выделяется одна защитная страница (4KiB на x86/x64). Чтобы установить размер защитной странцы равным 0, то есть отключить защитную страницу вообще, можно выполнить следующий код:

```
    pthread_attr_setguardsize(&attr, 0);
```

2.7.2. Управление фоновым режимом

По умолчанию нить создается в обычном (joinable) режиме. Завершение таких нитей нужно отслеживать с помощью `pthread_join`. Чтобы нить была создана в фоновом (detached) режиме, используется функция `pthread_attr_setdetachstate` следующим образом:

```
    pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);
```

2.8. Принудительное завершение нити

Любой процесс может быть немедленно завершен отправкой ему сигнала SIGKILL. Принудительное завершение процесса может привести к тому, что останутся неудаленные временные файлы, семафоры останутся закрытыми и т. д. Нити взаимодействуют друг с другом гораздо теснее, чем процессы, поэтому принудительное завершение нити намного более опасно, чем принудительное завершение процесса.

Тем не менее, послать запрос на принудительное завершение процесса можно с помощью функции `pthread_cancel`.

```
int pthread_cancel(pthread_t thread);
```

В параметре `thread` передается идентификатор нити, которой отправляется запрос на завершение. Эта функция аналогична системному вызову `kill` для оправки сигнала процессам.

Запрос на принудительное завершение нити может не быть обработан немедленно. По умолчанию нить работает в режиме *отложенного* (deferred) завершения. В режиме отложенного завершения запрос на завершение нити выполняется, когда нить вызывает библиотечные функции, содержащие так называемые «точки завершения» (cancellation point). Точки завершения расположены в системных вызовах и библиотечных функциях.

Таким образом, если нить выполняет вычисления в течении длительного времени, исполнение запроса на принудительное завершение будет отложено до вызова первой библиотечной функции.

Нить можно переключить в режим *асинхронного* (asynchronous) завершения. В режиме асинхронного завершения нить может быть принудительно завершена в произвольный момент времени. Из-за асинхронного принудительного завершения не гарантируется целостность служебных структур данных, используемых такими функциями как `malloc`, `printf`. Принудительно завершаемая нить не может использовать никаких функций стандартной библиотеки, кроме функций библиотеки pthread для переключения режима завершения и блокировки запросов на завершение. По сути единственное применение режима асинхронного завершения нити — это длительные вычисления, которые не вызывают библиотечные функции с побочным эффектом.

Для переключения режима завершения текущей нити используется функция библиотеки pthread `pthread_setcanceltype`.

```
int pthread_setcanceltype(int type, int *oldtype);
```

В параметре `type` передается новый режим завершения нити, который может быть либо константой `PTHREAD_CANCEL_DEFERRED` — стандартный режим отложенного завершения, либо константой `PTHREAD_CANCEL_ASYNCHRONOUS` — режим асинхронного завершения. В параметре `oldtype` передается указатель на переменную, в которую сохраняется старый режим завершения нити. Допускается передавать в качестве параметра `oldtype` значение NULL, тогда старый режим никуда не сохраняется.

В любом режиме завершения нити она может временно блокировать запросы на принудительное завершение нити. Если запрос поступит в момент, когда нить блокирует принудительное завершение, он будет отложен до момента разблокировки, при этом даже функции, являющиеся точками принудительного завершения (cancellation points), не будут завершать нить, пока принудительное завершение заблокировано.

Для переключения режима блокировки запросов на принудительное завершение используется функция `pthread_setcancelstate`.

```
int pthread_setcancelstate(int state, int *oldstate);
```

В параметре `state` передается новый режим блокировки запросов, который может быть либо константой `PTHREAD_CANCEL_ENABLE`, разрешающей обработку запросов на принудительное завершение, либо константой `PTHREAD_CANCEL_DISABLE`, приостанавливающей обработку запросов на принудительное завершение. В параметре `oldstate`

передается указатель на переменную, в которую сохраняется старый режим блокировки запросов. Допускается передавать значение NULL, тогда старый режим никуда не сохраняется.

2.9. Нити и сигналы

Сигналы обрабатываются на уровне процесса, а не на уровне отдельной нити. Как было сказано выше, поступление в процесс сигнала, который не блокируется, для которого действует стандартная реакция на сигнал, которая заключается в завершении работы процесса, завершает весь процесс. Например, переполнение стека в одной из нитей приведет к отправке в процесс сигнала SIGSEGV и последующему завершению всего процесса.

Системные вызовы `signal` или `sigaction` устанавливают реакцию на сигнал для всего процесса в целом, независимо от нити, из которой они вызваны. Так, установка пользовательского обработчика сигнала устанавливает его для всего процесса.

Блокировка сигналов с помощью `sigprocmask` также влияет на весь процесс в целом. Заблокированный на уровне процесса сигнал не доставляется немедленно, а остается в множестве сигналов, ожидающих доставки.

Если установлена функция-обработчик некоторого сигнала и этот сигнал не блокируется, то когда в процесс поступает этот сигнал, выбирается какая-то одна нить процесса и в ее контексте запускается обработчик сигнала. Каждый раз для вызова обработчика сигнала может выбираться другая нить.

При настройках обработки сигнала по умолчанию повторное получение сигнала блокируется на время обработки этого сигнала. Однако блокировка сигнала в этом случае работает на уровне отдельной нити, а не процесса в целом. Поэтому если некоторый сигнал поступает быстрее, чем отрабатывает обработчик этого сигнала, обработчик будет запущен в другой нити. Таким образом, может возникнуть ситуация, когда в разных нитях будут параллельно работать обработчики одного и того же сигнала. Это может быть серьезной проблемой, если обработчик сигнала сложнее, чем установка значения переменной-флага.

В стандарте pthread каждая нить может индивидуально блокировать доставку сигналов. Помимо маски заблокированных сигналов на уровне процесса у каждой нити есть своя маска заблокированных сигналов. Если сигнал не блокируется на уровне процесса, выбирается одна из нитей, у которых этот сигнал не заблокирован на уровне нити, и обработчик сигнала будет вызван в контексте одной из этих нитей.

Для манипулирования маской заблокированных сигналов отдельной нити используется функция `pthread_sigmask`.

```
int pthread_sigmask(int how, const sigset_t *set, sigset_t *oldset);
```

Функция полностью аналогична функции `sigprocmask` за исключением того, что работа идет с маской сигналов текущей нити, а не процесса в целом.

По адресу, передаваемому в параметре `oldset` сохраняется старая маска блокируемых сигналов. Если параметр равен NULL, старая маска никуда не сохраняется.

Параметр `how` управляет, как множество сигналов, адрес на которое передается в параметре `set` модифицирует маску блокируемых сигналов текущей нити. Если параметр `set` равен NULL, текущая маска блокируемых сигналов не изменяется. Если параметр `how` равен `SIG_BLOCK`, множество сигналов в `set` добавляется в маску блокируемых сигналов нити. Если параметр `how` равен `SIG_UNBLOCK`, множество сигналов в `set` удаля-

ется из маски блокируемых сигналов нити. Если параметр `how` равен `SIG_SETMASK`, множество сигналов в `set` копируется в маску блокируемых сигналов нити. Как обычно, сигналы SIGKILL и SIGSTOP нельзя блокировать. Попытка заблокировать эти сигналы с помощью `pthread_sigmask` не будет иметь эффекта.

Один из рекомендуемых способов обработки сигналов в многонитевых приложениях состоит в том, что для каждого сигнала выделяется одна нить, которая будет его обрабатывать. Возможно, что одна и та же нить будет обрабатывать все сигналы. В этой нити маска блокируемых сигналов нити разрешает обработку необходимых сигналов, а во всех других нитях все сигналы блокируются. При таком подходе нужно следить, чтобы у каждой создаваемой нити была правильная маска блокируемых сигналов.

Альтернативный (и более современный) способ обработки сигналов заключается в том, что все сигналы вообще блокируются на уровне процесса, а для синхронного уведомления о поступлении сигнала используются системные вызовы `sigwait`, `sigwaitinfo` или `signalfd`. Поступление сигналов можно отслеживать в отдельной нити или в основном цикле обработки сообщений, реализованном с помощью `pselect` или `ppoll` или `epoll`.

В следующем примере создается отдельная нить, выполняющая функцию `signal_thread`, которая обрабатывает поступающие в процесс сигналы.

```
#include <stdio.h>
#include <pthread.h>
#include <signal.h>
#include <unistd.h>

void *signal_thread(void *ptr)
{
    sigset_t ss;
    sigfillset(&ss);
    while (1) {
        // ждем любого сигнала
        siginfo_t info;
        int s = sigwaitinfo(&ss, &info);
        // печатаем номер поступившего сигнала
        printf("%d\n", s);
    }
}

int main(void)
{
    // блокируем все сигналы
    sigset_t ss;
    sigfillset(&ss);
    sigprocmask(SIG_BLOCK, &ss, NULL);
    // создаем новую нить
    pthread_t tid;
    pthread_create(&tid, NULL, signal_thread, NULL);
    // ничего не делаем
    sleep(100);
}
```

2.10. Нити и fork

Системный вызов `fork` создает новый процесс, являющийся копией текущего процесса. Новый процесс в начале работы получает то же самое состояние виртуальной памяти, что у родительского процесса. В частности, у созданного процесса будут скопированы стеки всех выполняющихся нитей родительского процесса. Однако из всех нитей родительского процесса работу в процессе-сыне продолжит **только одна нить**: та, которая выполняла системный вызов `fork`. В нее вернется выполнение после `fork` с возвращаемым значением 0. Все другие нити в процессе-сыне не будут существовать. С точки зрения процесса-сына это можно описать так, что в момент выполнения `fork` всем другим нитям был послан запрос на асинхронное завершение и все прочие нити были немедленно принудительно завершены.

Предположим, что в момент, когда одна нить выполнила `fork`, другая нить выполняла `printf`, то есть стандартный поток вывода `stdout` был заблокирован. В процессе-сыне только первая нить продолжит выполнение, поэтому стандартный поток вывода `stdout` останется заблокированным навсегда. Если первая нить вызовет в сыне `printf`, возникнет ситуация дедлока, так как первая нить в сыне будет ждать освобождения блокировки `stdout`, которое никогда не произойдет.

Поэтому в процессе-сыне после возврата из `fork` разрешается использовать только асинхронно-безопасные функции стандартной библиотеки, то есть функции, которые можно безопасно использовать в обработчиках сигналов. К асинхронно-безопасным функциям относится большинство системных вызовов, в частности, `execve`, `dup`, `chdir` и т. д.

Поэтому, пожалуй единственное разумное и корректное применение системного вызова `fork` в многонитевых программах — это в сыне сразу после `fork` настроить с помощью системных вызовов атрибуты процесса-сына, то есть, выполнить перенаправление стандартных потоков ввода-вывода, закрыть лишние файловые дескрипторы, поменять текущий каталог и т. п., а затем вызвать на выполнение другую программу с помощью функции семейства `exec`. Помимо низкоуровневых системных вызовов можно использовать и `system`, а вот `popen` использовать нельзя, так как эта функция использует `malloc` и не является асинхронно-безопасной.

3. Синхронизация в pthread

Разработка программ, использующих параллелизм выполнения на уровне нитей существенно более сложна, чем разработка программ, параллельных на уровне процессов. Нити выполняются в общем адресном пространстве, параллельно выполняющиеся нити имеют доступ не только к глобальным, но даже иногда и к локальным переменным (см. пример о создании нитей выше). Поэтому потенциально любой доступ к оперативной памяти может приводить к ошибке.

Если несколько нитей одновременно обращаются к одной и той же ячейке памяти, причем хотя бы одна из нитей выполняет запись в эту ячейку, такая ситуация называется *data race*. Если при выполнении программы возникает data race, дальнейшее поведение программы не определено (undefined behavior). Программы, при выполнении которых возможны data race, являются некорректными программами на C или C++.

Ситуация data race менее типична для многопроцессных программ, так как обычно каждый процесс выполняется в своем полностью изолированном адресном простран-

стве. Другие типы ошибок параллельных программ: гонки (race conditions) на системных вызовах и файловой системе, тупики (deadlocks) вполне возможны и в многонитевых программах.

Для синхронизации доступа в критические секции в нитях можно использовать средства синхронизации процессов: семафоры POSIX (`sem_open`) или семафоры SysV IPC (`semget`), каналы (`pipe`), `eventfd`. Тогда синхронизация доступа в критическую секцию в случае нитей не будет отличаться от случая процессов.

Однако средства синхронизации процессов не подходят для серьезного использования в нитях. Каждый механизм синхронизации: семафор, канал и т. д. обрабатывается на уровне ядра операционной системы, то есть в ядре выделяется память и создается объект ядра. Каждый раз при входе в критическую секцию и выходе из критической секции будет выполняться системный вызов. Это неприемлимо по двум причинам:

- В многонитевой программе каждая структура данных, совместно используемая нитями, должна защищаться от одновременного доступа или модификации. Таких структур данных может быть тысячи или десятки тысяч. Заводить для каждой из них объект в ядре — слишком накладно с точки зрения использования ограниченных ресурсов ядра операционной системы.

- Выполнять системный вызов каждый раз при входе в критическую секцию — слишком медленно.

Средства синхронизации нитей должны удовлетворять следующим требованиям:

- В процессе должно поддерживаться большое количество (десятки и сотни тысяч и более) одновременно существующих объектов синхронизации нитей. Каждый объект синхронизации не должен требовать ресурсов ядра, если он неактивен (то есть если нет нитей, ждущих на нем).

- Число системных вызовов при использовании объектов синхронизации должно быть минимальным. Например, типичный случай использования семафора, защищающего вход в критическую секцию, когда семафор открыт и вход в критическую секцию свободен, не должен требовать системного вызова для прохождения семафора.

Тем не менее, полностью обойтись без поддержки ядра операционной системы невозможно. Поскольку планировщик нитей (в модели 1:1) находится в ядре операционной системы, ядро должно знать, что, например, нить находится в ожидании открытия семафора, и должен убрать эту нить из очереди нитей, готовых к выполнению на процессоре. Для каждого объекта синхронизации нитей, на котором ждут наступления события одна или несколько нитей, ядро должно поддерживать список ожидающих нитей. Одна или все нити переводятся ядром в состояние готовности к выполнению, когда наступает событие, которого они ждут.

В следующих разделах мы рассмотрим мьютексы (mutex) и условные переменные (condition variable) — базовые механизмы синхронизации нитей в pthread.

3.1. Мьютексы

Мьютексы являются разновидностью семафоров. Напоминаем, что *семафором* называется тип данных, принимающий целочисленные значения от 0 до MAX — некоторого максимального значения, для которого определены две операции *up*(s, v) и *down*(s, v), где s — это переменная-семафор, а v — положительное целое значение.

Операция *down*(s, v) определена следующим образом:

- Если $s \geq v$, значение s уменьшается на v. Для итогового значения s выполняется $s \geq 0$. Операция выполняется атомарно в целом.
- Если $s < v$, процесс или нить, выполняющий операцию down помещается в список процессов/нитей, ожидающих увеличения значения s. Процесс/нить переводится в состояние ожидания и для него не будет выделяться процессорное время. С точки зрения процесса/нити его выполнение будет приостановлено на неограниченное время. Операция down считается незавершенной. Операция сравнения значения s и перевода процесса в состояние ожидания выполняется атомарно в целом.

Операция *up*(s, v) определена следующим образом:

- Если $s + v >$ MAX, операция неопределена.
- Если $s+v \leq$ MAX, значение семафора s увеличивается на v и разблокируются все процессы/нити, ожидающие увеличения значения s. Ожидающие процессы/нити переводятся в очередь процессов, готовых к выполнению, и им будет выделено процессорное время. Когда активированные процессы/нити будут запущены на процессоре, они снова попытаются выполнить операцию *down*. Операция увеличения значения s и активации ожидающих процессов/нитей атомарна в целом.

Операция *up* будет все процессы/нити, которые ждали на этом семафоре. Скорее всего из всех разбуженных нитей только одна из нитей сможет выполнить операцию *down*, а остальные нити снова «заснут» на семафоре. Поскольку значение v в операции *down* может быть разным у всех ожидающих нитей, будить только одну какую-то нить нельзя, необходимо разбудить все нити.

Семафор называется *бинарным*, если MAX $= 1$. Операции *up* и *down* имеют смысл только при $v = 1$, поэтому параметр v можно опускать, и операции записываются в виде up(s) и down(s). Операция down также называется lock, а операция up — unlock.

Мьютекс (англ. *mutex* — mutual exclusion) — это вариант бинарного семафора. Для мьютекса определены операции lock и unlock, но операция unlock определена только в случае, если операцию unlock выполняет тот же самый процесс/нить, который выполнял lock. Если операцию unlock выполняет какой-то другой процесс, результат операции неопределен. Таким образом, если семафоры общего вида можно использовать и для блокировки критической секции, и для передачи уведомлений в другие процессы/нити, мьютексы используются только для блокировки критической секции.

3.2. Мьютексы в pthread

В библиотеке pthread для мьютексов определен тип данных `pthread_mutex_t`. Переменная этого типа должна быть корректно проинициализирована перед использованием. Если переменная-мьютекс глобальная или локальная, проинициализировать ее можно специальной конструкцией вида:

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER; // инициализация
```

Если память под мьютекс выделяется динамически (с помощью `malloc`), или требуется проинициализировать мьютекс в нестандартном режиме, нужно использовать функцию `pthread_mutex_init`.

```
int pthread_mutex_init(
        pthread_mutex_t *mutex,
        const pthread_mutexattr_t *attr);
```

Параметр `mutex` — указатель на переменную-мьютекс, а параметр `attr` — указатель на переменную с дополнительными атрибутами создания мьютекса, которые будут рассмотрены ниже.

Если переменная-мьютекс является локальной, или память под мьютекс выделяется динамически, перед освобождением памяти мьютекс должен быть уничтожен, иначе возможна утечка ресурсов. Для уничтожения мьютекса используется функция `pthread_mutex_destroy`.

```
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

Мьютекс может быть уничтожен только один раз. Если функция вызывается на уже уничтоженном мьютексе, поведение функции не определено (undefined behavior).

Операция lock в библиотеке pthread выполняется функцией `pthread_mutex_lock`, определенной следующим образом:

```
int pthread_mutex_lock(pthread_mutex_t *mutex);
```

Параметр `mutex` — это указатель на мьютекс, над которым выполняется операция.

Если нить повторно пытается выполнить операцию lock на мьютекс, который она ранее уже захватила (такая операция называется relock), поведение операции lock зависит от типа мьютекса. Для стандартных мьютексов, то есть созданных без дополнительных атрибутов, поведение при relock неопределено (undefined behavior). Скорее всего, нить попадет в состояние тупика (deadlock).

Операция unlock выполняется функцией `pthread_mutex_unlock`, определенной следующим образом:

```
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

Параметр `mutex` — это указатель на мьютекс, над которым выполняется операция.

Если нить пытается освободить мьютекс, который она не захватывала, поведение операции unlock зависит от типа мьютекса. Для стандартных мьютексов поведение при освобождении «чужого» мьютекса неопределено (undefined behavior). Поведение при освобождении не занятого мьютекса также не определено (undefined behavior).

Функция `pthread_mutex_trylock` пытается захватить мьютекс, но не блокирует вызывающую нить, если мьютекс уже занят.

```
int pthread_mutex_trylock(pthread_mutex_t *mutex);
```

Параметр `mutex` — это указатель на мьютекс, над которым выполняется операция.

Если мьютекс был успешно захвачен, то функция возвращает нулевое значение. Если мьютекс уже заблокирован, функция возвращает значение константы `EBUSY`.

3.3. Рекурсивные мьютексы

Предположим, что мы хотим разработать структуру данных и операции для работы с банковским счетом.

```
struct Account
{
    double balance;
    pthread_mutex_t mutex;
};
```

Мьютекс `mutex` используется для блокировки доступа к полям структуры. Например, функции изменения баланса и получения текущего баланса могут быть такими:

```
void add_balance(struct Account *acc, double value)
{
    pthread_mutex_lock(&acc->mutex);
    acc->balance += value;
    pthread_mutex_unlock(&acc->mutex);
}
double get_balance(struct Account *acc)
{
    pthread_mutex_lock(&acc->mutex);
    double value = acc->value;
    pthread_mutex_unlock(&acc->mutex);
    return value;
}
```

А функция сравнения баланса — такой:

```
int compare_balance(struct Account *acc1, struct Account *acc2)
{
    int res = 0;
    pthread_mutex_lock(&acc1->mutex);
    pthread_mutex_lock(&acc2->mutex);
    if (acc1->balance < acc2->balance) res = -1;
    else if (acc1->balance > acc2->balance) res = 1;
    pthread_mutex_unlock(&acc2->mutex);
    pthread_mutex_unlock(&acc1->mutex);
    return res;
}
```

Однако, если указатели acc1 и acc2 равны, то некоторый счет сравнивается сам с собой, операция lock будет выполнена дважды, что приведет к тупику. В функции `compare_balance` исправить ошибку можно с помощью проверки аргументов на равенство, но в более сложных ситуациях устранение ситуации повторного захвата (relock) может быть более трудным.

Если каждая функция, работающая с данными в `struct Account`, захватывает мьютекс структуры, то попытка вызова одной функции из другой тоже приведет к повторному захвату мьютекса. Получается, что либо нужно как-то проверять, что мьютекс уже захвачен той же самой нитью, либо использовать две версии функций: одну с блокировками, другую без блокировок.

Для решения задачи повторного захвата мьютекса той же самой нитью предназначены *рекурсивные мьютексы*. В рекурсивном мьютексе поддерживается счетчик захватов мьютекса нитью. Когда нить успешно захватывает мьютекс, счетчик захватов устанавливается в 1. Когда нить, захватившая мьютекс, снова выполняет захват мьютекса, счетчик захватов увеличивается на 1. При освобождении мьютекса счетчик захватов

уменьшается на 1, и когда счетчик захватов становится равным 0, мьютекс освобождается.

Создать рекурсивный мьютекс можно с помощью указания дополнительных атрибутов создания мьютекса в вызове функции `pthread_mutex_init`. Дополнительные атрибуты создания мьютекса хранятся в типе данных `pthread_mutexattr_t`, правила работы с которым аналогичны правилам работы с типом `pthread_attr_t`. Инициализация рекурсивного мьютекса выглядит следующим образом:

```
pthread_mutex_t mutex;   // неинициализированный мьютекс

    pthread_mutexattr_t mattr;
    pthread_mutexattr_init(&mattr);  // инициализируем структуру атрибутов
    pthread_mutexattr_settype(&mattr, PTHREAD_MUTEX_RECURSIVE);
    pthread_mutex_init(&mutex, &mattr); // инициализируем мьютекс
    pthread_mutexattr_destroy(&mattr); // уничтожаем структуру атрибутов
```

Несмотря на использование рекурсивных мьютексов, описанная выше функция `compare_balance` содержит ошибку неправильного порядка захвата ресурсов, которая может приводить к тупикам. Эта ошибка подробно обсуждается в разделе, посвященном задаче обедающих философов.

3.4. Типы мьютексов

Помимо мьютексов по умолчанию и рекурсивных мьютексов библиотека pthread поддерживает мьютекс с проверкой ошибок. Желаемый тип мьютекса устанавливается в вызове функции `pthread_mutexattr_settype`. Поддерживаются следующие типы мьютексов:

- `PTHREAD_MUTEX_NORMAL` — стандартный мьютекс.
- `PTHREAD_MUTEX_ERRORCHECK` — мьютекс с контролем ошибок.
- `PTHREAD_MUTEX_RECURSIVE` — рекурсивный мьютекс.
- `PTHREAD_MUTEX_DEFAULT` — мьютекс с атрибутами по умолчанию.

Типы мьютексов различаются по реакции на следующие ситуации:

- Relock — повторный захват уже захваченного той же нитью мьютекса.
- Not owner unlock — освобождение мьютекса, захваченного другой нитью.
- Not locked unlock — освобождение незахваченного мьютекса.

Реакции разных типов мьютекса на эти ситуации приведены в таблице.

	Relock	Not owner unlock	Not locked unlock
Normal	UB	UB	UB
Errorcheck	ERR	ERR	ERR
Recursive	SPEC	ERR	ERR
Default	UB	UB	UB

UB — undefined behavior — неопределенное поведение (ошибка), ERR — соответствующая функция возвращает ошибку (ненулевое значение), SPEC — поведение функции специфицировано.

Отличие мьютекса Default от мьютекса Normal в том, что мьютекс типа Normal может быть реализован как мьютекс любого типа: Normal, Errorcheck, Recursive. В реализации на Linux мьютексы по умолчанию реализованы как нормальные мьютексы.

3.5. Условные переменные

Условные переменные (condition variables) — это механизм, с помощью которого одна нить может уведомить о наступлении события одну другую нить или все другие нити. По сути, условная переменная — это список ожидания нитей. Нить может добавить себя в список ожидания, тогда она будет находится в списке ожидания и не будет выполняться пока не будет разбужена другой нитью. Или нить может разбудить одну или несколько нитей в списке ожидания.

Операцию ожидания уведомления обозначим wait, операцию уведомления всех нитей в списке ожидания назовем broadcast, а операцию уведомления какой-то одной нити в списке ожидания назовем signal.

Сложность использования механизма условных переменных в том, что операция уведомления о наступлении события активирует только те нити, которые в данный момент находятся в списке нитей, ожидающих уведомления. Пусть первая нить выполняет операцию signal над некоторой условной переменной `w`, а вторая нить выполняет операцию wait над этой же переменной.

```
// момент времени t = 0          wait(&w); // t = 0
signal(&w); // t = 1              // t = 1
```

Если вторая нить войдет в ожидание поступления уведомления раньше (в момент времени $t = 0$), чем первая нить ее уведомит (в момент времени $t = 1$), вторая нить будет активировано и уведомление дойдет до адресата.

```
signal(&w); // t = 0              // t = 0
// момент времени t = 1          wait(&w); // t = 1
```

Если же первая нить попытается уведомить вторую раньше (в момент времени $t = 0$), чем вторая нить войдет в ожидание (в момент времени $t = 1$), то в момент времени $t = 0$ список нитей, ожидающих на условной переменной `w` не будет содержать вторую нить, поэтому до второй нити уведомление, отосланное первой, не дойдет. Если в момент времени $t = 0$ список ожидающих на `w` нитей пуст, то уведомление не дойдет вообще никому и потеряется. Таким ошибка типа race condition скорее всего приведет к временной или неограниченной блокировке первой нити.

В многонитевой программе невозможно без дополнительных средств синхронизации гарантировать, что операция wait во второй нити всегда начнется до операции signal в первой нити.

Поэтому дополнительно к собственно условной переменной требуется еще обычная переменная, которую мы будем использовать в ситуации, когда первая нить подошла к операции уведомления второй нити раньше, чем вторая нить подошла к операции ожидания.

Объявим дополнительную переменную `int f;` — флаг уведомления. Первая нить устанавливает переменную-флаг в 1 при отправке уведомления, а вторая нить перед входом в ожидание проверяет значение этой переменной.

```
f = 1;                            if (!f) {
signal(&w);                           wait(&w);
                                  }
```

Предполагается, что если первая нить выполнит signal перед тем, как вторая переменная войдет в wait, то первая нить установит переменную-флаг f, таким образом вторая нить немедленно пройдет дальше без входа в ожидание. Однако, если операции в нитях выполнятся в порядке:

```
// ...      // t = 0
f = 1;      // t = 1
signal(&w); // t = 2
// ...      // t = 3
// ...
```

```
if (!f) {     // t = 0
    // ...    // t = 1
    // ...    // t = 2
    wait(&w); // t = 3
}
```

В этом примере вторая нить проверяет значение переменной f в момент времени $t = 0$, когда это значение еще не установлено первой нитью и поэтому равно 0. Поэтому вторая нить входит в оператор if, но в момент времени $t = 1$ и $t = 2$ выполняется только первая нить, а вторая нить по каким-либо причинам не выполняется (например, исчерпан квант времени процесса). В моменты времени $t = 1$ и $t = 2$ первая нить и устанавливает значение переменной f, и уведомляет вторую нить с помощью операции signal. Поскольку вторая нить не успела еще войти в операцию wait, уведомление теряется и вторая нить блокируется на ожидании. Таким образом снова проявляется ошибка типа race condition.

Поэтому операция проверки значения f и перехода в состояние ожидания должна быть атомарной, для чего потребуется дополнительный мьютекс m. Первая нить устанавливает флаг и уведомляет вторую нить под мьютексом. Вторая нить под тем же мьютексом проверят флаг и выполняет ожидание.

```
lock(&m);
f = 1;
signal(&w);
unlock(&m);
```

```
lock(&m);
if (!f) {
    wait(&w);
}
unlock(&m);
```

Однако, если вторая нить зашла в критическую секцию под мьютексом m и вошла в ожидание на переменной w, мьютекс m останется закрытым и первая нить просто не сможет войти в свою критическую секцию.

Поэтому операция wait требует в качестве аргументов не только условной переменной w, но и мьютекса m, того самого, который охраняет проверку значения флага f. На время, пока вторая нить находится в состоянии ожидания, мьютекс m будет открыт, открывая путь первой нити в свою критическую секцию. Но в момент выхода из операции wait мьютекс снова будет закрыт второй нитью. То есть, когда первая нить отправляет второй уведомление, а вторая нить находится в состоянии ожидания в wait, вторая нить активируется, но не может продолжить выполнение, так как продолжение выполнения второй нити требует взятия мьютекса m, который в этот момент времени еще занят первой нитью. Лишь после того, как первая нить выйдет из критической секции, выполнив операцию unlock, вторая нить заберет себе мьютекс m и выйдет из операции wait сама. Операция перевода второй нити в режим ожидания и освобождения мьютекса m выполняется атомарно, то есть в эту операцию не может вклиниться другая нить, которая успеет отправить уведомление, которое потеряется.

Таким образом, отправка и получение уведомления в нитях должны выглядеть следующим образом:

```
lock(&m);
f = 1;
signal(&w);
unlock(&m);
```

```
lock(&m);
if (!f) {
    wait(&w, &m);
}
unlock(&m);
```

Операция wait принимает и условную переменную, на которой нить ждет уведомлений, и мьютекс, который открывает на время ожидания. Этого фрагмента кода достаточно для выполнения однократного ожидания уведомления.

Если вторая нить должна ждать уведомлений несколько раз, нужно не забыть в критической секции второй нити сбросить переменную `f` как показано ниже.

```
lock(&m);
f = 1;
signal(&w);
unlock(&m);
```

```
lock(&m);
if (!f) {
    wait(&w, &m);
}
f = 0;
unlock(&m);
```

Рассмотренные выше фрагменты кода предназначены для случая, когда уведомлений ожидает одна нить. Если уведомления могут или должны ожидать несколько нитей, отправка и получение уведомлений немного изменяются.

Если уведомления могут или должны ждать несколько нитей, но все они уведомляются одновременно, то есть все ожидающие нити должны быть переведены из режима ожидания в режим выполнения, то вместо операции signal нужно использовать операцию broadcast следующим образом:

```
lock(&m);
f = 1;
broadcast(&w); // уведомление всем
unlock(&m);
```

```
lock(&m);
if (!f) {
    wait(&w, &m);
}
f = 0;
unlock(&m);
```

Если уведомления могут или должны ждать несколько нитей, но продолжить выполнение после уведомления может только одна нить, в первой нити следует использовать операцию signal, но в ожидающих нитях вместо оператора if следует использовать while.

```
lock(&m);
f = 1;
signal(&w);
unlock(&m);
```

```
lock(&m);
while (!f) {
    wait(&w, &m);
}
f = 0;
unlock(&m);
```

Цикл while защищает от ситуации, когда в списке ожидающих нитей находится одна нить-получатель уведомления, но к моменту отправки уведомления первой нитью ко входу в критическую секцию подошла вторая нить-получатель уведомления. Тогда после отправки уведомления одна нить-получатель будет ждать освобождения критической секции в операции wait, а вторая нить-получатель будет ждать освобождения критической секции в операции lock. Когда первая нить выйдет из критической секции войти в освободившуюся критическую секцию может любая из двух нитей-получателей, в частности, та, которая ждет на операции lock. Она пройдет критическую секцию и сбросит `f`, тогда та нить, которая ждала в операции wait должна снова перейти в ожидание на wait, так как ее уведомление было «перехвачено» другой нитью.

Если есть несколько нитей-отправителей уведомления и несколько нитей-получателей уведомления, то булевского флага будет недостаточно, и необходимо использовать счетчик доступных уведомлений, как показано ниже.

```
lock(&m);                                   lock(&m);
++f;        // счетчик уведомлений          while (!f) {
signal(&w);                                     wait(&w, &m);
unlock(&m);                                 }
                                            --f;       // забрали уведомление
                                            unlock(&m);
```

Этот пример объясняет, почему булевский флаг поступления уведомления и мьютекс не встроены в операции signal и wait. Вынося мьютекс и переменную-значение вовне операций signal и wait мы делаем их более универсальными. С помощью signal и wait можно не только передавать уведомление, то есть информацию о факте наступления некоторого события, но и передавать произвольное значение.

Использование условной переменной всегда предполагает использование мьютекса, блокирующего доступ в критическую секцию работы с условной переменной, и булевского флага или произвольного значения, которое собственно передает информацию о событии от нити-отправителя к нити-получателю.

3.6. Условные переменные pthread

Для условных переменных в pthread используется тип `pthread_cond_t`. Переменная этого типа должна быть инициализирована. Если переменная объявлена как глобальная или локальная, можно использовать статический инициализатор:

```
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
```

Если память под переменную выделяется динамически, нужно использовать функцию инициализации:

```
int pthread_cond_init(pthread_cond_t *cond, const pthread_condattr_t *attr);
```

Параметр `cond` — это указатель на инициализируемую условную переменную, а параметр `attr` — указатель на структуру с дополнительными атрибутами создания условной переменной. В рамках данного пособия дополнительные атрибуты рассматриваться не будут.

Перед освобождением памяти под условную переменную ее следует уничтожить, чтобы не допускать ситуацию утечки ресурсов.

```
int pthread_cond_destroy(pthread_cond_t *cond);
```

Параметр `cond` — указатель на уничтожаемую условную переменную.

Операции signal, broadcast и wait реализуются следующими функциями:

```
int pthread_cond_broadcast(pthread_cond_t *cond);
int pthread_cond_signal(pthread_cond_t *cond);
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);
```

Параметр `cond` — указатель на условную переменную, `mutex` — указатель на связанный с ней мьютекс.

4. Классические задачи синхронизации

В данном разделе мы рассмотрим классические задачи синхронизации процессов, и покажем, как они решаются средствами синхронизации процессов pthread.

4.1. Обедающие философы

«Литературная» формулировка задачи была впервые дана Дейкстрой. За круглым столом расположились N философов ($N \geq 2$). Справа и слева от каждого философа лежит по вилке, причем каждую вилку используют два философа. Например, философ с индексом 0 испольует вилки 0 и 1, философ с индексом 1 использует вилки 1 и 2, философ с индексом $N-1$ использует вилки $N-1$ и 0.

Одна итерация цикла жизни философа состоит из раздумий, во время которых его вилки не используются им и могут быть использованы соседними философами, и еды, когда философ двумя своими вилками ест спагетти из своей тарелки. Соседние философы в это время не могут есть, так как нужные им вилки заняты. Таким образом, совместно используемые философами вилки являются разделяемым ресурсами, корректную работу с которыми и требуется организовать.

Простое решение можно записать следующим образом: пусть глобальный массив forks — это массив мьютексов размера N. Функция `take_forks`, которая захватывает требуемые философу вилки, может выглядеть так:

```
// phil_num — индекс философа, для которого берется вилка
void take_forks(int phil_num)
{
    pthread_mutex_lock(&forks[phil_num]);
    pthread_mutex_lock(&forks[(phil_num + 1) % N]);
}
```

Освобождать ресурсы будем в порядке, обратном к порядку, в котором они были захвачены. Функция `put_forks` освобождает две захваченные вилки.

```
// phil_num — индекс философа, для которого берется вилка
void put_forks(int phil_num)
{
    pthread_mutex_unlock(&forks[(phil_num + 1) % N]);
    pthread_mutex_unlock(&forks[phil_num]);
}
```

Функция нити, моделирующая поведение философа, может тогда быть записана в следующем виде:

```
void *philosopher_func(void *ptr)
{
    int id = (intptr_t) ptr;
    while (1) {
        // думаем
        sleep(THINK_TIME);
        // захватываем вилки
        take_forks(id);
        // едим
        sleep(EAT_TIME);
        // возвращаем вилки
        put_forks(id);
    }
}
```

К сожалению, реализация функций `take_forks` и `put_forks` некорректна и может

приводить к тупику при выполнении. Рассмотрим ситуацию, когда все N нитей одновременно вошли в функцию `take_forks` и одновременно выполнили взятие первого мьютекса. Таким образом, мьютекс 0 захвачен нитью 0, мьютекс 1 захвачен нитью 1 и так далее. В итоге все мьютексы захвачены, открытых мьютексов нет, поэтому ни одна нить не может захватить второй требуемый мьютекс и войти в критическую секцию, то есть приступить к еде.

Если мы рассмотрим задачу для двух нитей (при $N = 2$), в параллельных нитях 0 и 1 будут выполняться следующие операции:

```
pthread_mutex_lock(&forks[0]);          pthread_mutex_lock(&forks[1]);
pthread_mutex_lock(&forks[1]);          pthread_mutex_lock(&forks[0]);
```

Проблема тупика возникает из-за того, что нити захватывают вилки в разном порядке. Если бы нити захватывали вилки в одном и том же порядке, то есть фрагмент работы нитей был бы таким:

```
pthread_mutex_lock(&forks[0]);          pthread_mutex_lock(&forks[0]);
pthread_mutex_lock(&forks[1]);          pthread_mutex_lock(&forks[1]);
```

то ситуация тупика никогда бы не возникала. Обобщая решение задачи при $N = 2$ на произвольный N легко получить, что если все нити захватывают вилки в фиксированном порядке: сначала вилка с меньшим номером, затем вилка с большим номером, то ситуация тупика не возникнет. Нить с номером $N-1$ первой попытается захватить вилку 0. Но если вилка 0 уже захвачена, то нить с номером $N-1$ не будет пытаться захватить вилку $N-1$, разрывая таким образом цикл захватов вилок.

Обратите внимание, что функция `compare_balance`, описанная выше в разделе про рекурсивные мьютексы, содержит ту же самую ошибку.

Корректная реализация функции `take_forks` может быть такой:

```
// phil_num — индекс философа, для которого берется вилка
void take_forks(int phil_num)
{
    if (phil_num == N - 1) {
        pthread_mutex_lock(&forks[0]);
        pthread_mutex_lock(&forks[(phil_num) % N]);
    } else {
        pthread_mutex_lock(&forks[phil_num]);
        pthread_mutex_lock(&forks[(phil_num + 1) % N]);
    }
}
```

В худшем случае, когда все нити одновременно захватят свои первые вилки, то есть нить 0 захватит вилку 0, нить 1 — вилку 1, нить $N-2$ захватит вилку $N-2$, нить с индексом $N-1$ будет ожидать захвата своей первой вилки 0, а нить $N-2$ сможет захватить вилку с индексом $N-1$ и войти в критическую секцию. Этот случай худший, так как кроме нити $N-2$ никакая другая нить не сможет войти в критическую секцию, и в итоге нить 0 будет вынуждена подождать, пока все нити 1, 2, ..., $N-2$ успешно пройдут критическую секцию. То есть, если в оптимальной ситуации в критической секции может находится половина от общего числа нитей, а остальные нити будут ждать на мьютексах, в худшей ситуации в критической секции будет находится только одна нить, а остальные нити будут ждать на мьютексах.

Тем не менее, это решение хорошо тем, что оно не требует каких-то других примитивов синхронизации, кроме мьютексов, и просто обобщается на произвольное количество захватываемых мьютексов.

Задача об обедающих философах — это задача о превильном порядке захвата ресурсов в многонитевой системе. Если на множестве ресурсов каким-то образом определено отношение полного порядка, например, у каждого ресурса есть свой целочисленный идентификатор, то ресурсы должны захватываться в порядке возрастания идентификатора ресурса, а освобождаться в порядке, обратном к захвату. Это справедливок для захвата произвольного числа ресурсов, а не только двух.

4.2. Барьер

Пусть есть N параллельно выполняющихся нитей. Необходимо написать функцию `barrier`, которая бы удовлетворяла следующим условиям:

- Если хотя бы одна из N нитей не вошла в функцию `barrier`, ни одна уже вошедшая нить не может из функции выйти.
- Все N нитей активируются на выход из функции одновременно.

Барьер — одноразовый примитив синхронизации. Если нитям требуется пройти несколько барьеров, нужно создать несколько примитивов. Напишем структуру данных и функции для инициализации и уничтожения этой структуры в стиле pthread, но с префиксом `thr_`.

```
struct BarrierData
{
    int thread_count;      // общее число нитей
    int cur_count;         // число нитей вошедших в барьер
    pthread_mutex_t mutex;
    pthread_cond_t cond;   // список ожидания нитей в барьере
};

typedef struct BarrierData thr_barrier_t;
```

Напишем макрос для статической инициализации и функции инициализации и разрушения структуры данных:

```
#define THR_BARRIER_INITIALIZER(N) { (N), 0, PTHREAD_MUTEX_INITIALIZER,
    PTHREAD_COND_INITIALIZER }

void thr_barrier_init(thr_barrier_t *b, int thread_count)
{
    b->thread_count = thread_count;
    b->cur_count = 0;
    pthread_mutex_init(&b->mutex, NULL);
    pthread_cond_init(&b->cond, NULL);
}

void thr_barrier_destroy(thr_barrier_t *b)
{
    pthread_mutex_destroy(&b->mutex);
    pthread_cond_destroy(&b->cond);
}
```

Сама функция барьера оказывается очень простой:

```
void thr_barrier(thr_barrier_t *b)
{
    pthread_mutex_lock(&b->mutex);
    if (++b->cur_count < b->thread_count) {
        // если в барьер вошли не все нити, ждем
        pthread_cond_wait(&b->cond, &b->mutex);
    } else {
        // последняя вошедшая нить будет остальных
        pthread_cond_broadcast(&b->cond);
    }
    pthread_mutex_unlock(&b->mutex);
}
```

4.3. Читатели-писатели

Рассмотрим типичную ситуацию, обычно возникающую при работе с разделяемым ресурсом в многонитевой программе. Все функции, манипулирующие с разделяемым ресурсом, могут быть разделены на две группы: *функции-читатели*, которые не изменяют внутреннее состояние объекта, а только возвращают информацию о его состоянии, и *функции-писатели*, которые модифицируют состояние объекта. И функции-читатели, и функции-писатели должны блокировать нежелательный параллельный доступ к разделяемому ресурсу, чтобы возвращать целостные данные, либо чтобы выполнить корректное обновление. Очевидно, что разные функции-читатели в разных нитях вполне могут работать с разделяемым ресурсом одновременно, поскольку они не модифицируют данные, они не мешают друг другу. Но вот когда с разделяемым ресурсом работает функция-писатель, с ним не должна больше работать ни одна другая нить.

Если доступ на чтение к разделяемому ресурсу достаточно простой и быстрый (например, несколько операций чтения из памяти), как правило, нет смысла реализовывать поддержку одновременного нахождения нескольких читателей в критической секции — накладные расходы на реализацию съедят все возможное небольшое повышение производительности. Но если функции-читатели выполняют достаточно сложные операции, и выполнение функций-читателей требует уже заметного времени, от реализации разделяемого доступа на чтение можно получить выигрыш. Например, представим себе таблицу в базе данных, к которой могут выполняться достаточно сложные запросы на выборку данных, которые будут требовать существенных вычислительных ресурсов компьютера.

Корректная реализация задачи читателей-писателей должна удовлетворять следующим требованиям:

- В критической секции одновременно может находиться не более одного писателя.
- В критической секции одновременно может находиться произвольное количество читателей.
- Нити, которые не могут войти в критическую секцию, не потребляют процессорного времени (отсутствует активное ожидание).

Напишем структуру данных и функции инициализации и уничтожения структуры данных в стиле pthread, но с использованием префикса `thr_`.

```
struct SimpleRWLock
{
    pthread_mutex_t mutex; // мьютекс критической секции
    pthread_cond_t cond;   // список ожидающих нитей
    int writer_count;      // число писателей в крит. секции
    int reader_count;      // число читателей в крит. секции
};

typedef struct SimpleRWLock thr_rwlock_t;

#define THR_RWLOCK_INITIALIZER { PTHREAD_MUTEX_INITIALIZER,
    PTHREAD_COND_INITIALIZER, 0, 0 }

void thr_rwlock_init(thr_rwlock_t *rw)
{
    pthread_mutex_init(&rw->mutex, NULL);
    pthread_cond_init(&rw->cond, NULL);
    rw->writer_count = 0;
    rw->reader_count = 0;
}

void thr_rwlock_destroy(thr_rwlock_t *rw)
{
    pthread_cond_destroy(&rw->cond);
    pthread_mutex_destroy(&rw->mutex);
}
```

Функция входа в критическую секцию для читателя должна проверять, что в критической секции нет писателей:

```
void thr_rwlock_rdlock(thr_rwlock_t *rw)
{
    pthread_mutex_lock(&rw->mutex);
    while (rw->writer_count > 0) {
        // если критическая секция занята писателем, ждем
        pthread_cond_wait(&rw->cond, &rw->mutex);
    }
    ++rw->reader_count;
    pthread_mutex_unlock(&rw->mutex);
}
```

Функция входа в критическую секцию для писателя должна проверять, что в критической секции нет ни читателей, ни писателей:

```
void thr_rwlock_wrlock(thr_rwlock_t *rw)
{
    pthread_mutex_lock(&rw->mutex);
    while (rw->writer_count > 0 && rw->reader_count > 0) {
        // если критическая секция кемто- занята, ждем
        pthread_cond_wait(&rw->cond, &rw->mutex);
```

```
    }
    rw->writer_count = 1;
    pthread_mutex_unlock(&rw->mutex);
}
```

Функция выхода из критической секции будет общей для читателей и писателей. Функция проверяет, что если в критической секции никого не осталось, все ждущие нити должны быть разбужены.

```
void thr_rwlock_unlock(thr_rwlock_t *rw)
{
    pthread_mutex_lock(&rw->mutex);
    if (rw->reader_count > 1) {
        // еще остались читатели
        --rw->reader_count;
    } else {
        // никого не осталось
        rw->reader_count = 0;
        rw->writer_count = 0;
        pthread_cond_broadcast(&rw->cond);
    }
    pthread_mutex_unlock(&rw->mutex);
}
```

Это простое решение использует одну условную переменную и для читателей, и для писателей, поэтому при выходе из критической секции мы вынуждены будить все ожидающие нити. Улучшить механизм пробуждения нитей можно, если поддерживать отдельные списки ожидающих нитей-читателей и нитей-писателей и хранить количество ожидающих нитей в каждом списке. Тогда при выходе из критической секции можно будить только одного ждущего писателя (если таковые имеются), либо сразу всех читателей (если есть ждущие читатели).

Задача читателей-писателей настолько распространена на практике, что библиотека pthread содержит тип данных `pthread_rwlock_t` для соответствующей структуры данных и функции входа в критическую секцию и выхода из критической секции, аналогичные описанным выше.

```
int pthread_rwlock_destroy(pthread_rwlock_t *rwlock);
int pthread_rwlock_init(pthread_rwlock_t *rwlock, const pthread_rwlockattr_t *attr);
pthread_rwlock_t rwlock = PTHREAD_RWLOCK_INITIALIZER;

int pthread_rwlock_rdlock(pthread_rwlock_t *rwlock);
int pthread_rwlock_tryrdlock(pthread_rwlock_t *rwlock);
int pthread_rwlock_wrlock(pthread_rwlock_t *rwlock);
int pthread_rwlock_trywrlock(pthread_rwlock_t *rwlock);
int pthread_rwlock_unlock(pthread_rwlock_t *rwlock);
```

За дальнейшим описанием этих функций обращайтесь к документации.

Реализация решения задачи читателей-писателей, описанная выше, обладает следующим недостатком: если функция-читатель находится внутри критической секции заметное время, и к данной структуре идет плотный поток запросов от читателей, возможна ситуация, когда в критической секции неограниченно долго будет находиться более одного читателя одновременно. Поэтому функция unlock при выходе читателя

из критической секции не сможет разблокировать критическую секцию для входа в нее писателя. Поэтому нить-писатель может находиться в ожидании входа в критическую секцию неограниченно долго.

Ситуация, когда какая-то нить в течение долгого или неограниченного времени не может получить доступа к разделяемому ресурсу из-за особенностей реализации алгоритма, называется *голоданием* (resource starvation). Решение, допускающее голодание нитей, называется *несправедливым* (unfair). Решение, рассмотренное выше, является несправедливым по отношению к нитям-писателям, так как допускает голодание при захвате ресурсов нитью-писателем.

Мы можем модифицировать алгоритм входа в критическую секцию для читателя, например, так, что если в списке ожидания есть хотя бы один писатель, читатель не выполняет вход в критическую секцию, а переходит в режим ожидания. Такое решение будет отдавать приоритет писателям, что достаточно логично, так как писателей обычно меньше чем читателей, и мы хотим, чтобы обновление разделяемого ресурса прошло как можно быстрее от момента появления запроса на обновление. Однако такое решение будет несправедливым по отношению к нитям-читателям, так как если будет поступать плотный поток запросов на запись в разделяемый ресурс, приоритет доступа в критическую секцию будет всегда отдаваться писателям, и возникнет ситуация голодания для нитей-читателей.

Справедливое решение задачи читателей-писателей должно удовлетворять следующим свойствам:

- В критической секции одновременно может находиться не более одного писателя.
- В критической секции одновременно может находиться произвольное количество читателей.
- Нити, которые не могут войти в критическую секцию, не потребляют процессорного времени (отсутствует активное ожидание).
- Нить, запросившая вход в критическую секцию позже, не может войти в критическую секцию перед нитью, запросившей критическую секцию раньше. То есть, если в критическую секцию входит некоторая нить, то нити, которые запросили критическую секцию раньше, либо уже вышли из критической секции, либо находятся в критической секции, либо выполняют вход в критическую секцию.
- Нить-читатель может и должна войти в критическую секцию без ожидания, если либо критическая секция свободна, либо в критической секции есть читатели, и в очереди на вход перед данной нитью нет писателей.

Реализацию справедливого решения задачи читателей-писателей мы оставим в качестве упражнения.