Потоки

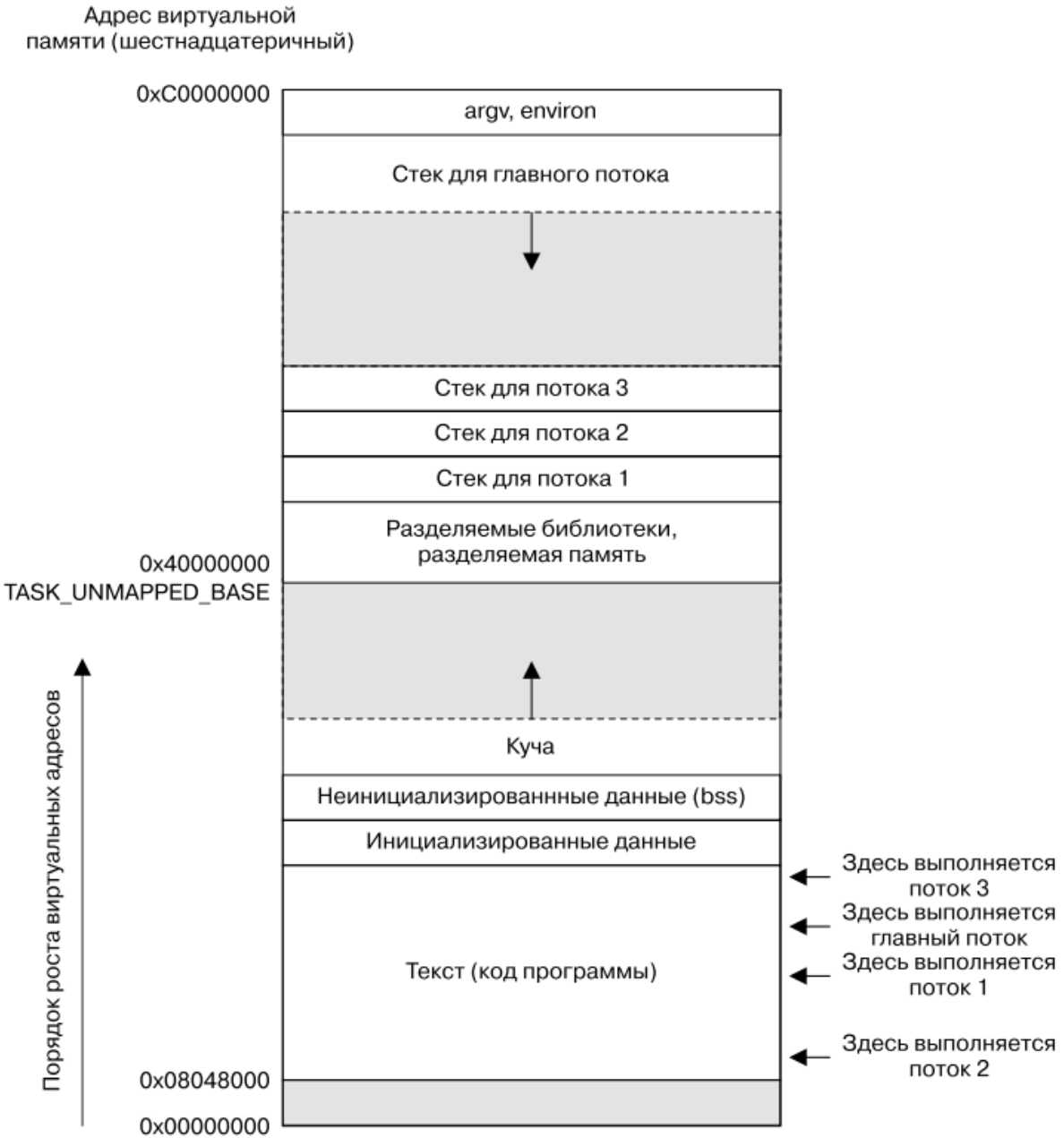
Наумов Д.А., доц. каф. КТ

Операционные системы и системное программное обеспечение, 2020

Содержание лекции

Обзор

Потоки выполнения представляют собой механизм для одновременного выполнения нескольких параллельных задач в рамках одного приложения. Один процесс может содержать несколько потоков. Все они выполняются внутри одной программы независимо друг от друга, разделяя общую глобальную память — в том числе инициализированные/неинициализированные данные и сегменты кучи.


Адрес виртуальной
памяти (шестнадцатеричный)
0xC0000000
argv, environ
Стек для главного потока
Стек для потока 3
Стек для потока 2
Стек для потока 1
Разделяемые библиотеки,
разделяемая память
0x40000000
TASK_UNMAPPED_BASE
Порядок роста виртуальных адресов
Куча
Неинициализированнные данные (bss)
Инициализированные данные
Текст (код программы)
0x08048000
0x00000000
Здесь выполняется
поток 3
Здесь выполняется
главный поток
Здесь выполняется
поток 1
Здесь выполняется
поток 2

Разделяемые атрибуты потоков

- Идентификаторы процесса и его родителя
- Идентификаторы группы процессов и сессии
- Управляющий терминал
- Учетные данные процесса
- Дескрипторы открытых файлов
- Блокировки записей, созданные с помощью вызова `fcntl()`
- Действия сигналов
- Информация, относящаяся к файловой системе
- Интервальные таймеры и POSIX-таймеры
- Ограничения на ресурсы
- Потребленное процессорное время
- Потребленные ресурсы
- Значение `nice`

Уникальные атрибуты потоков

- Идентификатор потока
- Маска сигнала
- Данные, относящиеся к определенному потоку
- Альтернативный стек сигналов
- Переменная `errno`
- Настройки плавающей запятой
- Политика и приоритет планирования в режиме реального времени
- Стек

Создание потоков

Для создания нового потока используется функция `pthread_create()`.

```
#include <pthread.h>
int pthread_create(pthread_t *thread,
                   const pthread_attr_t *attr,
                   void *(*start)(void *), void *arg);
// Возвращает 0 при успешном завершении
// и положительное число при ошибке
```

Новый поток начинает выполнение с вызова функции, указанной в виде значения `start` и принимающей аргумент `arg`.

Создание потоков

Аргумент `thread` указывает на буфер, в который записывается уникальный идентификатор созданного потока.
Аргумент `arg` объявлен как `void *`. Это означает, что мы можем передать функции `start` указатель на объект любого типа. Если нам нужно передать функции `start` несколько аргументов, мы можем предоставить в качестве `arg` указатель на структуру, содержащую эти аргументы в виде отдельных полей.
В аргументе `attr` передаются различные атрибуты нового потока.

Завершение потоков

Выполнение потока прекращается по одной из следующих причин:

- Начальная функция выполняет инструкцию `return`, указывая возвращаемое значение для потока.
- Поток вызывает функцию `pthread_exit()`.
- Поток отменяется с помощью функции `pthread_cancel()`.
- Любой из потоков вызывает `exit()` или главный поток выполняет инструкцию return в функции main()

Завершение потоков

Функция `pthread_exit()` завершает вызывающий поток и указывает возвращаемое значение.

```
#include <pthread.h>
void pthread_exit(void *retval);
```

Аргумент `retval` хранит значение, возвращаемое потоком. Значение, на которое указывает `retval`, не должно находиться в стеке самого потока, поскольку по окончании вызова `pthread_exit()` его содержимое становится неопределенным.
Если главный поток вызовет `pthread_exit()` вместо `exit()` или инструкции `return`, остальные потоки продолжат выполнение.

Идентификаторы потоков

Каждый поток внутри процесса имеет свой уникальный идентификатор. Поток может получить свой идентификатор с помощью функции pthread_self().

```
#include <pthread.h>
pthread_t pthread_self(void);
// Возвращает идентификатор вызывающего потока
```

Идентификаторы потоков внутри приложения можно использовать следующим образом.

- В различных функциях из состава Pthreads для определения того, в каком потоке они выполняются.
- Для маркировки динамических структур данных идентификатором определенного потока. Так мы можем определить создателя и «владельца» структуры или определить поток, который должен выполнить какие-то последующие действия со структурой данных.

Идентификаторы потоков

Функция `pthread_equal()` позволяет проверить на тождественность два идентификатора потоков.

```
#include <pthread.h>
int pthread_equal(pthread_t t1, pthread_t t2);
// Возвращает ненулевое значение, если t1 и t2 равны,
// в противном случае возвращает 0
```

Необходимость в функции `pthread_equal()` возникает из-за того, что тип данных `pthread_t` должен восприниматься как непрозрачный. В различных системах он может быть целым числом, указателем или структурой.

Присоединение потока

Функция `pthread_join()` ждет завершения потока, обозначенного аргументом `thread`.

```
#include <pthread.h>
int pthread_join(pthread_t thread, void **retval);
// Возвращает 0 при успешном завершении
// или положительное число при ошибке
```

Если аргумент `retval` является ненулевым указателем, функция получает копию возвращаемого значения завершенного потока.

Присоединение потока

Процедура, которую выполняет функция `pthread_join()` для потоков, похожа на действие вызова `waitpid()` для процессов. Но между ними есть и заметные различия.

- Потоки не имеют иерархии. Любой поток в процессе может воспользоваться функцией `pthread_join()` для присоединения другого потока в том же процессе. Это отличается от иерархических отношений между процессами. Только родительский процесс может ожидать завершения своего потомка.
- Невозможно присоединиться к любому потоку. Функция `pthread_join()` может присоединить поток только при наличии его идентификатора. В случае с процессами мы можем это сделать с помощью вызова `waitpid(-1, status, options)`.

Отсоединение потока

Иногда статус, возвращаемый потоком, не имеет значения; нам просто нужно, чтобы система автоматически освободила ресурсы и удалила поток, когда тот завершится. В этом случае мы можем пометить поток как отсоединенный, воспользовавшись функцией `pthread_detach()`.

```
#include <pthread.h>
int pthread_detach(pthread_t thread);
// Возвращает 0 при успешном завершении
// или положительное число, если возникла ошибка
```

Если поток уже был отсоединен, мы больше не можем получить его возвращаемый статус с помощью функции `pthread_join()`. Мы также не можем снова сделать его присоединяемым.

Атрибуты потока

Атрибуты потока содержат такие сведения, как местоположение и размер стека потока, политику его планирования и приоритет), а также информацию о том, является ли поток присоединяемым или отсоединенным.

Создание отсоединенного потока

```
pthread_attr_t attr;
pthread_attr_init(&attr);  // инициализация атрибутов
// указываем, что поток будет создан отсоединенным
pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);
pthread_t tid;
// создание нового потока с заданными атрибутами
pthread_create(&tid, &attr, tfunc, (void *) args);
// удаление структуры с атрибутами
pthread_attr_destroy(&attr);
```

- Обмен информацией между процессами имеет свои сложности. Поскольку родитель и потомок не разделяют память, мы вынуждены использовать некую форму межпроцессного взаимодействия для обмена данными. Обмен информации между потоками является простым и быстрым. Для этого всего лишь нужно скопировать данные в общие переменные
- Создание процесса с помощью `fork()` потребляет относительно много ресурсов. Приходится дублировать различные атрибуты процесса. Создание потока занимает меньше времени, поскольку многие атрибуты вместо непосредственного копирования просто разделяются.
- Создавая программу на основе потоков, мы должны следить за тем, чтобы используемые функции были потокобезопасными. Многопроцессные приложения могут об этом не заботиться.

- Ошибка в одном потоке может повредить остальные потоки в процессе, поскольку все они используют общее адресное пространство и некоторые другие атрибуты. Процессы являются более изолированными друг от друга.
- Каждый поток соперничает за возможность использования конечного адресного пространства своего процесса. В частности, стек и локальное хранилище каждого потока потребляет часть виртуальной памяти процесса, что делает ее недоступной для других потоков. Отдельные процессы, с другой стороны, могут задействовать весь диапазон свободной виртуальной памяти, ограниченный лишь размерами RAM и пространства подкачки.
- Работа с сигналами в многопоточном приложении требует тщательного проектирования. Как правило, рекомендуется избегать использования сигналов в многопоточных программах.

- В многопоточном приложении все потоки должны выполнять одну и ту же программу. В многопроцессных приложениях разные процессы способны выполнять разные программы.
- Помимо самих данных, потоки разделяют и другую информацию. Это может быть как преимуществом, так и недостатком, в зависимости от приложения.

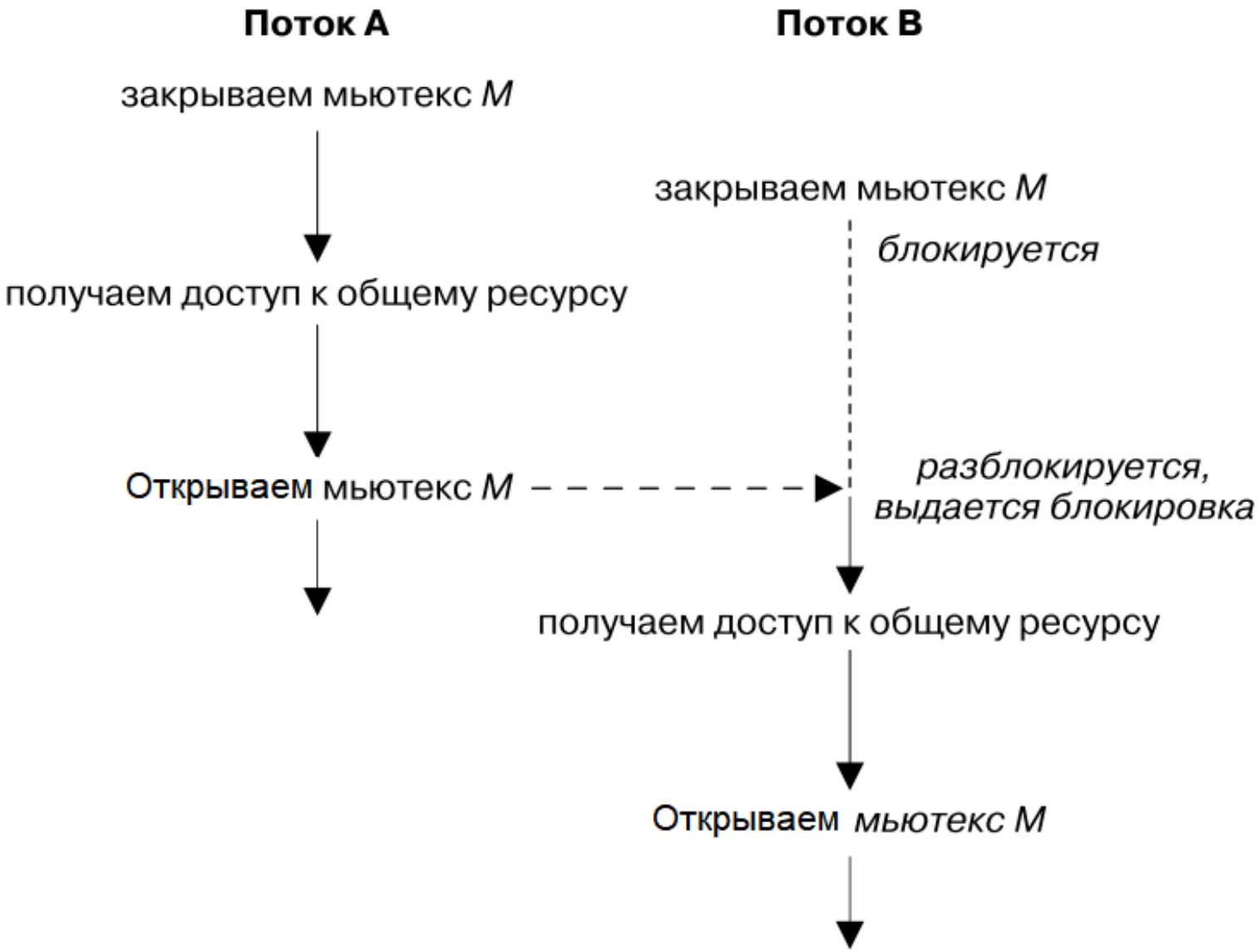
Мьютексы

Мьютексы являются простейшей формой синхронизации. Они используются для защиты критической критической области, предотвращая одновременное выполнение участка кода несколькими потоками.
Мьютекс имеет два состояния: закрытое (блокированное) и открытое (разблокированное). В любой момент времени только один поток может удерживать мьютекс закрытым. Попытки закрыть уже закрытый мьютекс либо отклоняются, либо приводят к ошибке. Когда поток закрывает мьютекс, он становится его владельцем. Только владелец мьютекса может его открыть.

Мьютексы

Хотя мы говорим о критической области кода, на самом деле речь идет о защите разделяемых (совместно используемых) данных. Для доступа к разделяемому ресурсу поток использует такую последовательность действий:

- закрыть мьютекс для разделяемого ресурса;
- получить доступ к разделяемому ресурсу;
- открыть мьютекс.


Поток A
Поток B
закрываем мьютекс M
закрываем мьютекс M
блокируется
получаем доступ к общему ресурсу
Открываем мьютекс M
разблокируется,
выдается блокировка
получаем доступ к общему ресурсу
Открываем мьютекс M

Мьютексы

Стоит отметить, что закрытие мьютекса является скорее рекомендацией, а не требованием. Иными словами, поток может проигнорировать использование мьютекса и просто обратиться к соответствующим разделяемым переменным. Но, чтобы задействовать эти переменные безопасным образом, все потоки должны совместно пользоваться мьютексом.

Создание мьютекса

Мьютекс может быть создан статически или динамически.
Если мьютекс создается статически, то перед использованием должен быть проинициализирован значением `PTHREAD_MUTEX_INITIALIZER`.
Динамическая инициализация мьютекса выполняется с помощью функции `pthread_mutex_init()`. Мьютекс инициализируется с помощью этой функции с следующих случаях.

- Мьютекс был динамически выделен в куче.
- Мьютекс является автоматической переменной, выделенной в стеке.
- Мы хотим инициализировать статически выделенный мьютекс нестандартными атрибутами.

Создание мьютекса

```
#include <pthread.h>
int pthread_mutex_init(pthread_mutex_t *mutex,
                       const pthread_mutexattr_t *attr);
// Возвращает 0 при успешном выполнении
// или положительное число, если случилась ошибка
```

Аргумент `mutex` обозначает мьютекс, который должен быть инициализирован.
Аргумент `attr` обозначает атрибуты мьютекса. В зависимости от реализации, данные атрибуты могут отличаться. Среди них могут быть атрибуты, указывающие может ли мьютекс использоваться для синхронизации потоков, определяющие поведение мьютекса при закрытии, устанавливающие приоритет потока, который закрыл мьютекс, и другие.

Закрытие и открытие мьютекса

Для закрытия и открытия мьютекса используются функции `pthread_mutex_lock()` и`pthread_mutex_unlock()`.

```
#include <pthread.h>
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_trylock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
// Все возвращают 0 при успешном завершении
// или положительное число, если случилась ошибка
```

Если мьютекс открыт, то вызов `pthread_mutex_lock()` сразу возвращается. Если же мьютекс закрыт другим потоком, `pthread_mutex_lock()` блокируется, пока тот не откроется. Если открытый мьютекс ожидает несколько потоков, то закрыть его может любой из них (может зависеть от приоритета потоков).

Закрытие и открытие мьютекса

Функция `pthread_mutex_trylock()` аналогична функции `pthread_mutex_lock()`, за исключением того, что при попытке закрыть мьютекс, который уже закрыт другим потоком, она вернет ошибку.
Функция `pthread_mutex_unlock()` открывает мьютекс, закрытый ранее вызывающим потоком. Если мьютекс уже открыт или был закрыт другим потоком, данная функция возвращает ошибку.

Взаимное блокирование

Иногда потоку нужно получить доступ сразу к двум или более разделяемым ресурсам, каждый из которых управляется отдельным мьютексом. Если один и тот же набор мьютексов закрывается несколькими потоками, может произойти взаимное блокирование.

Поток А

```
1. pthread_mutex_lock(mutex1);
2. pthread_mutex_lock(mutex2);
```

блокируется

Поток Б

```
1. pthread_mutex_lock(mutex2);
2. pthread_mutex_lock(mutex1);
```

блокируется

Самый простой способ избежать подобного взаимного блокирования заключается в определении иерархии мьютексов. Если потоки могут закрыть один и тот же набор мьютексов, они всегда должны делать это в одном и том же порядке.

fasterthanlime ✨
@fasterthanlime

~ cheat sheet ~

The plural of "index" is "indices"
The plural of "vertex" is "vertices"
The plural of "mutex" is "deadlock"

Удаление мьютекса

Когда мьютекс, выделенный автоматически или динамически, больше не нужен, его следует уничтожить с помощью функции `pthread_mutex_destroy()`.

```
#include <pthread.h>
int pthread_mutex_destroy(pthread_mutex_t *mutex);
// Возвращает 0 при успешном завершении
// или положительное число, если произошла ошибка
```

Безопасное уничтожение мьютекса возможно, только когда он находится в открытом состоянии и нет ни одного потока, который бы впоследствии планировал его закрыть. Если мьютекс находится на участке динамически выделяемой памяти, его нужно уничтожить до того, как будет освобожден этот участок. Автоматически выделенный мьютекс нужно уничтожать до возврата функции, в которой он выполняется.

Условные переменные

Условные переменные позволяют потокам информировать друг друга об изменениях в состоянии разделяемых ресурсов и ждать (блокируясь) получения таких уведомлений.
Условные переменные всегда используются в сочетании с мьютексами. Мьютекс обеспечивает взаимоисключающий доступ к разделяемому ресурсу, тогда как условная переменная сигнализирует об изменении его состояния.

Создание условных переменных

По аналогии с мьютексами, условные переменные могут выделяться статически и динамически. Статически выделяемые условные переменные должны быть проинициализированы значением `PTHREAD_COND_INITIALIZER`. Для динамической инициализации мьютекса используется функция `pthread_cond_init()`. Случаи применения динамической инициализации условных переменных те же, что для мьютексов.

Создание условных переменных

```
#include <pthread.h>
int pthread_cond_init(pthread_cond_t *cond,
                      const pthread_condattr_t *attr);
// Возвращает 0 при успешном завершении
//или положительное число, если произошла ошибка
```

Аргумент cond определяет условную переменную, которую нужно инициализировать.
В аргументе attr передаются атрибуты условной переменной. Как и для мьютексов, атрибуты условных переменных зависят от реализации. Например, могут быть атрибуты, указывающие, может ли условная переменная использоваться для синхронизации процессов, задающие ограничение времени ожидания сигнала переменной.

Удаление условных переменных

Когда автоматически или динамически выделенная условная переменная больше не нужна, ее необходимо удалить с помощью функции `pthread_cond_destroy()`.

```
#include <pthread.h>
int pthread_cond_destroy(pthread_cond_t *cond);
// Возвращает 0 при успешном завершении
// или положительное число, если произошла ошибка
```

Уничтожение условной переменной является безопасным только в том случае, если ее не ожидает ни один из потоков. Если условная переменная находится на участке динамически выделяемой памяти, ее следует уничтожить до освобождения этого участка. Автоматически выделенную условную переменную нужно уничтожать до возврата функции, в которой она выполняется.

Потоковая безопасность

Функция считается потокобезопасной, если она может быть безопасно вызвана сразу из двух потоков.
Например, следующий код не является потокобезопасным.

```
static int glob = 0;

static void incr(int loops){
    int loc, j;
    for (j = 0; j < loops; j++) {
        loc = glob;
        loc++;
        glob = loc;
    }
}
```

Если несколько потоков одновременно вызовут данную функцию, мы не сможем предсказать итоговое значение переменной glob.

Потоковая безопасность

Сделать функцию потокобезопасной можно различными способами. Например, к ней можно привязать мьютекс, который будет закрываться при ее вызове и открываться при завершении. Это приводит к тому, что в любой момент времени такую функцию может только один поток. Из-за этого потоки программы больше не могут выполняться параллельно.

Более сложным решением является связывание мьютекса с разделяемой переменной. После этого нужно определить, какие участки функции являются критическими, и затем закрывать/открывать мьютекс только при их выполнении. Это позволяет нескольким потокам одновременно выполнять одну и ту же функцию, действуя параллельно.

Реентерабельность

Однако подход с ипользованием мьютексов для критических участков все равно не очень эффективен, поскольку на закрытие и открытие мьютексов тоже тратятся ресурсы. Реентерабельные функции позволяют достичь потоковой безопасности без применения мьютексов. Это делается за счет отказа от глобальных и статических переменных. Любая информация, которую следует вернуть в вызывающий поток или сохранить между вызовами функции, хранится в буфере, выделенном вызывающим потоком.

Данные уровня потока

Самый эффективный способ обеспечения потоковой безопасности функции — это сделать ее реентерабельной. Но если существующая функция нереентерабельна, данный подход обычно требует изменения ее интерфейса, что влечет за собой изменение всех программ, которые ее используют.
Задействование данных, относящихся к отдельному потоку, позволяет сделать функцию потокобезопасной, не изменяя при этом ее интерфейс. Этот способ позволяет функции иметь отдельную копию переменной для каждого потока, который ее вызывает. Данные уровня потока являются постоянными; они продолжают существовать между вызовами.

Работа с данными уровня потока

- Функция должна выделить отдельный блок хранилища для каждого вызывающего ее потока. Этот блок должен выделяться только один раз — в момент вызова функции из потока.
- При каждом последующем вызове из того же потока функция должна иметь возможность получить адрес блока хранилища, который был выделен во время первого вызова из этого потока.
- Разным функциям могут понадобиться данные уровня потока. Каждой функции может потребоваться механизм определения данных уровня потока, чтобы отличить их от данных, используемых другими функциями.
- Должен существовать некий механизм, который бы позволил автоматически освобождать блок хранилища, выделенный для потока, когда тот завершается.

Работа с данными уровня потока

1. Функция создает ключ, с помощью которого можно отличить элементы данных уровня потока, задействуемые разными функциями. Ключ создается посредством вызова `pthread_key_create()`.
2. Вызов `pthread_key_create()` имеет дополнительное назначение: он позволяет вызывающему потоку указать адрес функции-деструктора, которая используется для освобождения каждого блока хранилища, выделенного с помощью текущего ключа.
3. Функция выделяет блок данных для каждого потока, в котором она вызывается.
4. Для сохранения указателя на хранилище, выделенное в предыдущем шаге, используются две функции: `pthread_setspecific()` и `pthread_getspecific()`. Первая сохраняет указатель и связывает его с определенным потоком и ключом. Вторая возвращает указатель на заданный ключ.

Работа с данными уровня потока

Вызов функции pthread_key_create() приводит к созданию нового ключа, связанного с данными уровня потока; этот ключ возвращается в вызывающий поток внутри буфера, на который указывает аргумент key.

```
#include <pthread.h>
int pthread_key_create(pthread_key_t *key,
                       void (*destructor)(void *));
// Возвращает 0 при успешном завершении
// или положительное число, если произошла ошибка
```

Функция pthread_setspecific() сохраняет копию value в структуре данных, которая связывает ее с вызывающим потоком и ключом key, возвращенным предыдущим вызовом pthread_key_create().

```
#include <pthread.h>
int pthread_setspecific(pthread_key_t key, const void *value);
//Возвращает 0 при успешном завершении
// или положительное число, если случилась ошибка
```

Работа с данными уровня потока

Функция `pthread_getspecific()` выполняет обратную операцию, возвращая значение, которое ранее было связано с заданным ключом этого потока.

```
#include <pthread.h>
void *pthread_getspecific(pthread_key_t key);
// Возвращает указатель или NULL,
// если с ключом не связаны данные уровня потока
```

Когда поток впервые вызывает функцию, он должен сначала сделать вызов `pthread_getspecific()`, чтобы проверить, имеет ли он значение, связанное с ключом. Если такого значения нет, функция выделяет блок памяти и сохраняет указатель на него с помощью вызова `pthread_setspecific()`.

Отмена потоков

Иногда возникает необходимость в отмене потока; это выглядит как передача потоку запроса с просьбой немедленно закончить работу. Это может пригодиться в ситуации, когда, например, группа потоков выполняет какие-то вычисления и один из них обнаруживает ошибку, из-за которой нужно завершить все остальные потоки.
Функция `pthread_cancel()` отправляет заданному потоку `thread` запрос отмены.

```
#include <pthread.h>
int pthread_cancel(pthread_t thread);
// Возвращает 0 при успешном завершении
// или положительное число, если произошла ошибка
```

Состояние отмены

Функция `pthread_setcancelstate()` позволяет указать состояние отмены.

```
#include <pthread.h>
int pthread_setcancelstate(int state, int *oldstate);
// Возвращает 0 при успешном завершении
// или положительное число, если произошла ошибка
```

Аргумент `state` может принимать значения

- `PTHREAD_CANCEL_DISABLE` — поток нельзя отменить.
- `PTHREAD_CANCEL_ENABLE` — поток можно отменить.

Предыдущее состояние отмены возвращается по адресу, на который указывает аргумент `oldstate`.

Тип отмены

Если поток можно отменить, реакция на запрос отмены определяется типом отмены потока, который можно задать при вызове `pthread_setcanceltype()`.

```
#include <pthread.h>
int pthread_setcanceltype(int type, int *oldtype);
// Возвращает 0 при успешном завершении
// или положительное число, если произошла ошибка
```

Аргумент `type` может принимать значения

- `PTHREAD_CANCEL_ASYNCHRONOUS` — поток может быть отменен в любой момент.
- `PTHREAD_CANCEL_DEFERRED` — процедура отмены задерживается до определенного момента.

Предыдущий тип отмены потока возвращается по адресу, на который указывает аргумент `oldtype`.

Точки отмены

Если тип отмены потока имеет значение `PTHREAD_CANCEL_DEFERRED`, то поток может быть отменен при достижении функций, которые являются точками отмены. Например

- функции чтения (`read()`, `recv()`, `recvfrom()`, `recvmsg()` и другие)
- функции записи (`write()`, `send()`, `sendto()`, `sendmsg()` и другие)
- функции ожидания (`pause()`, `pthread_cond_wait()`, `pthread_join()`, `wait()` и другие)

Кроме стандартных точек отмены (некоторые из которых перечислены выше), система может определять дополнительные. Это могут быть любые функции, которые могут блокировать выполнение программы.

Проверка возможности отмены потока

Поток, который не содержит точек отмены, может время от времени вызывать `pthread_testcancel()`, чтобы обеспечить своевременный ответ на запрос отмены, отправляемый другим потоком.

```
#include <pthread.h>
void pthread_testcancel(void);
```

Единственное назначение функции `pthread_testcancel()` — быть точкой отмены.

Освобождение ресурсов

Перед завершением потока в результате достижения точки отмены необходимо освободить занятые им ресурсы. Каждый поток может иметь стек обработчиков для очистки ресурсов.
Для добавления и удаления обработчиков используют следующие функции.

```
#include <pthread.h>
void pthread_cleanup_push(void (*routine)(void*), void *arg);
void pthread_cleanup_pop(int execute);
```

Первая функция принимает обработчик и его аргументы.
Аргумент `execute` второй функции указывает, будет ли выполнен обработчик при его удалении.