ОПЕРАЦИОННЫЕ СИСТЕМЫ И СИСТЕМНОЕ ПРОГРАММИРОВАНИЕ

Лекция 16 — Мьютексы POSIX

Преподаватель: Поденок Леонид Петрович, 505a-5

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

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

Кафедра ЭВМ, 2022

2024.05.23

Оглавление

Компромисс между проверкой ошибок и производительностью

Стандарт не требует обнаружения многих возможных ошибок, что дает возможность иметь компромисс между производительностью и уровнем проверки ошибок в соответствии с потребностями конкретных приложений и среды выполнения.

Как правило, библиотека обнаруживает обстоятельства, вызванные системой (например, недостаточный объем памяти).

Обстоятельства, вызванные ошибками кодирования в приложении (например, необеспечение адекватной синхронизации для предотвращения удаления мьютекса во время использования), просто упоминаются в документации в том смысле, что поведение в таких случаях неопределенно (как кривая вывезет).

Таким образом, существуют возможности широкого спектра реализаций.

Например, реализация, предназначенная для отладки приложений, может реализовывать все проверки ошибок.

Реализация, выполняющая одно *доказуемо* правильное приложение при очень жестких ограничениях производительности на встроенном компьютере может выполнять минимальные проверки.

Реализация может быть даже представлена в двух версиях, аналогичных параметрам, предоставляемым компиляторами — полная проверка, но более медленная версия, и версия с ограниченным числом проверок, но более быстрая.

Стандарт POSIX.1-2017 тщательно ограничивает использование т.н. «неопределенного поведения» только тем, что может делать ошибочное (плохо кодированное) приложение. Стандарт не заставляет все реализации добавлять накладные расходы для проверки множества вещей, которых никогда не делает правильная программа.

Однако, стандарт определяет, что ошибки недоступности ресурсов являются обязательными.

Такой подход данный стандарт гарантирует, что полностью соответствующее стандарту приложение будет переносимо для всего диапазона реализаций.

Если поведение не определено, номер ошибки, который должен возвращаться в реализациях, обнаруживающих условие, не указывается (абзац «Функция ... может завершиться ошибкой, если:»).

Это связано с тем, что поведение *undefined* означает, что может произойти что угодно, включая возврат с любым значением (которое может оказаться действительным, но другим номером ошибки).

Однако, поскольку номер ошибки может быть полезен разработчикам приложений при диагностике проблем во время разработки приложения, в качестве обоснования дается рекомендация о том, что разработчики должны возвращать конкретный номер ошибки, если их реализация действительно обнаруживает такое условие.

Почему не определены ограничения

Также рассматривались определения символов для максимального количества мьютексов и условных переменных, но были отклонены, поскольку количество этих объектов может динамически изменяться. Более того, многие реализации помещают эти объекты в память приложения — таким образом, явного максимума нет.

Синхронизация потоков POSIX на барьере

Барьерная синхронизация — метод синхронизации в распределённых/параллельных вычислениях, при котором выполнение параллельного алгоритма или его части можно разделить на несколько этапов, разделённых барьерами.

В частности, с помощью барьера можно организовать точку сбора частичных результатов вычислений, в которой подводится итог.

Барьер для группы потоков/процессов означает, что каждый поток/процесс должен остановиться в этой точке и подождать достижения барьера всеми потоками/процессами группы. Когда все потоки/процессы достигнут барьера, их выполнение продолжится.

`pthread_barrier_init/destroy()` — инициализировать/уничтожить объект барьера
`pthread_barrier_wait()` — синхронизироваться у барьера

pthread_barrier_init/destroy(3) – инициализировать/уничтожить объект барьера

```
#include <pthread.h>

int pthread_barrier_init(pthread_barrier_t           *restrict barrier,
                         const pthread_barrierattr_t *restrict attr,
                         unsigned                     count);

int pthread_barrier_destroy(pthread_barrier_t *barrier);
```

Функция **pthread_barrier_init()** выделяет ресурсы, необходимые для использования барьера, на который ссылается параметр **barrier**, и инициализирует барьер с атрибутами, на которые ссылается **attr**.

Если **attr** равен **NULL**, будут использоваться атрибуты барьера по умолчанию. Действие такое же, как при передаче в **attr** адреса объекта атрибутов барьера по умолчанию.

count – количество потоков, которые должны вызвать функцию **pthread_barrier_wait()**, прежде чем любой из них успешно вернется из вызова. Значение, указанное в **count**, должно быть больше нуля.

Важно! UB:

Если функция **pthread_barrier_init()** дает сбой, барьер не будет проинициализирован и содержимое барьера будет не определено.

Если **pthread_barrier_init()** вызывается, когда какой-либо поток заблокирован на барьере **barrier**, то есть не вернулся из вызова **pthread_barrier_wait()**, результат не определен.

Если барьер используется без предварительной инициализации, результат не определен.

Если вызывается **pthread_barrier_init()** с указанием уже инициализированного барьера, результат не определен.

Функция **pthread_barrier_destroy()** разрушает барьер, на который ссылается **barrier**, и освобождает все ресурсы, используемые барьером.

Если **pthread_barrier_destroy()** вызывается, когда какой-либо поток заблокирован на барьере, или если эта функция вызывается с неинициализированным барьером, результат не определен.

Возвращаемое значение

После успешного завершения эти функции вернут 0.

В противном случае будет возвращено -1, а значение **errno** будет указывать на ошибку.

EAGAIN — системе не хватает ресурсов, необходимых для инициализации другого барьера.

ENOMEM — недостаточно памяти для инициализации барьера.

EINVAL — значение, указанное в **count**, равно нулю.

EINVAL — значение, указанное аргументом барьера для **pthread_barrier_destroy()**, не относится к инициализированному объекту барьера.

EINVAL — значение, указанное аргументом **attr** для **pthread_barrier_init()**, не относится к инициализированному объекту атрибутов барьера.

EBUSY — значение, указанное аргументом **barrier** для **pthread_barrier_destroy()** или **pthread_barrier_init()**, ссылается на барьер, который используется (например, в вызове **pthread_barrier_wait()**) другим потоком, или значение, указанное аргумент барьера для **pthread_barrier_init()** относится к уже инициализированному объекту барьера.

pthread_barrier_wait() — синхронизироваться у барьера

```
#include <pthread.h>

int pthread_barrier_wait(pthread_barrier_t *barrier);
```

Функция **pthread_barrier_wait()** синхронизирует участвующие потоки на барьере, на который ссылается **barrier**.

Вызывающий поток будет блокироваться до тех пор, пока необходимое количество потоков не вызовет **pthread_barrier_wait()**, указав барьер.

Когда необходимое количество потоков вызвало **pthread_barrier_wait()** с указанием данного барьера, потоки будут разблокированы и одному из участвующих потоков будет возвращена константа **PTHREAD_BARRIER_SERIAL_THREAD**, а каждому из оставшихся потоков будет возвращен ноль.

На этом этапе барьер сбрасывается в состояние, которое он имел в результате вызова последней функции **pthread_barrier_init()**, которая на него ссылалась.

Константа **PTHREAD_BARRIER_SERIAL_THREAD** определена в **<pthread.h>**, и ее значение отличается от любого другого значения, возвращаемого **pthread_barrier_wait()**.

UB: Если эта функция вызывается с неинициализированным барьером, результат не определен.

Если потоку, заблокированному на барьере, доставляется сигнал, по возвращении из обработчика сигнала поток возобновит ожидание на барьере, если ожидание барьера не было завершено, т.е. если во время выполнения обработчика сигнала требуемое количество потоков барьера не достигло.

В противном случае поток будет продолжаться как это имеет место после завершения ожидания барьера.

Могут ли другие потоки пройти через барьер, когда все они его достигнут, если поток в обработчике сигнала не вернется из него, не определено.

Поток, который заблокирован на барьере, не должен препятствовать любому незаблокированному потоку, который имеет право использовать те же ресурсы обработки, в конечном итоге продвигаться вперед в своем выполнении.

Возвращаемое значение

После успешного завершения функция **pthread_barrier_wait()** вернет одному (произвольному) потоку, синхронизирующемуся на барьере, значение **PTHREAD_BARRIER_SERIAL_THREAD** и ноль каждому из других потоков.

В противном случае должен возвращается номер ошибки.

EINVAL — значение, указанное аргументом **barrier** в **pthread_barrier_wait()**, не относится к инициализированному объекту барьера.

Мьютексы POSIX

Мьютекс (mutex, mutual exclusion — «взаимное исключение») — аналог одноместного семафора, служащий для синхронизации одновременно выполняющихся потоков.

Мьютекс отличается от семафора тем, что его освободить может только владеющий им поток.

Цель использования мьютексов — защита данных от повреждения в результате асинхронных изменений (состояние гонки).

Основное назначение мьютексов — организация взаимного исключения для потоков из одного и того же или из разных процессов.

Мьютексы могут находиться в одном из двух состояний — открытом или закрытом.

Когда какой-либо поток, принадлежащий любому процессу, становится владельцем объекта **`mutex`**, мьютекс переводится в закрытое состояние. Если задача освобождает мьютекс, его состояние становится открытым.

Задача мьютекса — защита объекта от доступа к нему других потоков, отличных от того, который завладел мьютексом.

В каждый конкретный момент только один поток может владеть объектом, защищённым мьютексом.

Если другому потоку будет нужен доступ к переменной, защищённой мьютексом, то этот поток блокируется до тех пор, пока мьютекс не будет освобождён.

Базовое использование

`pthread_mutex_init()`	— инициализировать мьютекс
`pthread_mutex_destroy()`	— уничтожить мьютекс
`pthread_mutex_lock()`	— заблокировать мьютекс
`pthread_mutex_trylock()`	— попытаться заблокировать мьютекс с немедленным возвратом
`pthread_mutex_timedlock()`	— заблокировать мьютекс с ограниченным ожиданием
`pthread_mutex_unlock()`	— разблокировать мьютекс

pthread_mutex_init/destroy(3p) — инициализировать и уничтожить мьютекс

```
#include <pthread.h>

int pthread_mutex_init(pthread_mutex_t restrict           *mutex, // #1
                       const pthread_mutexattr_t restrict *attr);
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;                 // #2

int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

Функция **pthread_mutex_init()** инициализирует мьютекс, на который ссылается **mutex**, с атрибутами, указанными в **attr**.

Если **attr** равен **NULL**, используются атрибуты мьютекса по умолчанию. Эффект такой же, как если бы был передан адрес объекта атрибутов мьютекса по умолчанию.

После успешной инициализации состояние мьютекса становится инициализированным и разблокированным.

Попытка инициализировать уже инициализированный мьютекс приводит к неопределенному поведению.

В случаях, когда допускаются атрибуты мьютекса по умолчанию, для инициализации мьютексов можно использовать макрос **PTHREAD_MUTEX_INITIALIZER** (см. «Динамическая инициализация пакета/библиотеки»).

При его использовании поведение эквивалентно динамической инициализации вызовом **pthread_mutex_init()** с параметром **attr**, заданным как **NULL**, за исключением того, что в случае макроса не выполняются проверки на ошибки.

В случае успеха функция **pthread_mutex_init()** возвращает ноль.

В противном случае возвращается номер ошибки.

Функция **pthread_mutex_destroy()** уничтожает объект мьютекса, на который ссылается **mutex**. Фактически, объект мьютекса после вызова становится неинициализированным.

В некоторых реализациях **pthread_mutex_destroy()** может установить объект, на который ссылается **mutex**, просто в недопустимое значение.

В случае успеха функция **pthread_mutex_destroy()** возвращает ноль.

В противном случае возвращается номер ошибки.

Уничтоженный объект мьютекса можно повторно инициализировать с помощью повторного вызова **pthread_mutex_init()**.

Уничтожение инициализированного разблокированного мьютекса является безопасным.

UB:

Результаты любых других ссылок на объект после его уничтожения не определены.

К неопределенному поведению приводит попытка уничтожить :

- заблокированный мьютекс;
- мьютекс, который пытается заблокировать другой поток;
- мьютекс, который используется другим потоком в вызове **pthread_cond_timedwait()** или **pthread_cond_wait()**.

Если значение, указанное аргументом **mutex** при вызове **pthread_mutex_destroy()**, не относится к инициализированному мьютексу, поведение не определено.

Если значение, указанное аргументом **attr** для **pthread_mutex_init()**, не относится к инициализированному объекту атрибутов мьютекса, поведение не определено

Ошибки

Функция **pthread_mutex_init()** ***завершается*** ошибкой, если:

EAGAIN — Системе не хватало необходимых ресурсов (кроме памяти) для инициализации другого мьютекса.

ENOMEM — Недостаточно памяти для инициализации мьютекса.

EPERM — Вызывающий не имеет права выполнять операцию.

Функция **pthread_mutex_init()** ***может завершиться*** ошибкой, если:

EINVAL — у объекта атрибутов, на который ссылается **attr**, установлен атрибут робастности мьютекса без установленного атрибута общего доступа.

EINVAL — значение, указанное аргументом **mutex** для **pthread_mutex_destroy()**, не относится к инициализированному мьютексу.

EBUSY — значение, указанное аргументом **mutex** для **pthread_mutex_destroy()** или **pthread_mutex_init()**, относится к заблокированному мьютексу.

EBUSY — значение, указанное аргументом **mutex** для **pthread_mutex_destroy()** или **pthread_mutex_init()**, относится к мьютексу, на который ссылается (например, при использовании в **pthread_cond_timedwait()** или **pthread_cond_wait()**) другой поток,

EBUSY — значение, указанное аргументом **mutex** для **pthread_mutex_init()**, относится к уже инициализированному мьютексу.

EINVAL — значение, указанное аргументом **attr** для **pthread_mutex_init()**, не относится к инициализированному объекту атрибутов мьютекса.

Альтернативные реализации

Последняя версия стандарта POSIX.1-2017 поддерживает несколько альтернативных реализаций мьютексов.

Реализация может хранить блокировку непосредственно в объекте типа **pthread_mutex_t**.

В качестве альтернативы реализация может хранить блокировку в куче, а в объекте мьютекса просто хранить указатель, дескриптор или уникальный идентификатор.

Каждая реализация имеет свои преимущества или может потребоваться для определенных конфигураций оборудования.

Чтобы можно было написать переносимый код, инвариантный к выбору альтернативных реализаций, POSIX.1-2017 не определяет для типа **pthread_mutex_t** ни операций присвоения, ни операций равенства, вместо этого используется термин `` *инициализировать* '', чтобы усилить (более ограничительное) понятие что блокировка может фактически находиться в самом объекте мьютекса.

Следует обратить внимание, что такой подход предотвращает чрезмерную и явную спецификацию типа **pthread_mutex_t** или условной переменной и стимулирует использование в качестве **pthread_mutex_t** непрозрачного (opaque) типа.

Стандартом допускается, но не является обязательным, чтобы **pthread_mutex_destroy()** могла сохранять недопустимое значение в мьютексе.

Такое поведение может помочь обнаружить ошибочные программы, которые пытаются заблокировать (или иным образом ссылаются) на уже уничтоженный мьютекс.

Уничтожение мьютексов

Мьютекс можно уничтожить сразу после разблокировки. Однако, поскольку попытка уничтожить заблокированный мьютекс или мьютекс, который пытается заблокировать другой поток, или мьютекс, который используется в вызове **pthread_cond_timedwait()** или **pthread_cond_wait()** другим потоком, приводит к неопределенному поведению, необходимо соблюдать осторожность.

Забота о том, чтобы никакой другой поток не мог ссылаться на уничтожаемый мьютекс возлагается на программиста.

Динамическая инициализация пакета/библиотеки

Некоторые библиотеки C разработаны таким образом, что используют динамическую инициализацию. Это значит, что глобальная инициализация библиотеки выполняется при вызове первой процедуры в этой библиотеке, а не при старте программы.

В однопоточной программе это обычно реализуется с использованием статической переменной, значение которой проверяется при входе в процедуру, как показано ниже:

```
static int random_is_initialized = 0; // флаг инициализации
extern void initialize_random(void);  // функция инициализации

int random_function( ) {
    if (random_is_initialized == 0) {
        initialize_random( );
        random_is_initialized = 1;
    }
    ... // Тут разный рабочий код
}
```

Чтобы сохранить такую же структуру в многопоточной программе, необходим новый примитив. В противном случае инициализация библиотеки должна выполняться явным вызовом функции инициализации из экспортируемой библиотеки до любого использования этой библиотеки.

Если в многопоточном процессе для динамической инициализации библиотеки используется флаг инициализации, этот флаг должен быть защищен от модификации несколькими потоками, одновременно вызывающими библиотеку. Это можно сделать с помощью мьютекса, который инициализируется назначением **PTHREAD_MUTEX_INITIALIZER**.

Однако лучшим решением будет использование **pthread_once( )**, которая предназначена именно для этой цели.

pthread_once(3) — динамическая инициализация пакета/библиотеки

```
#include <pthread.h>

int pthread_once(pthread_once_t *once_control,   // ссылка на статический флаг
                 void (*init_routine)(void));    // подпрограмма инициализации

pthread_once_t once_control = PTHREAD_ONCE_INIT; // установка флага
```

Первый вызов **pthread_once()** любым потоком процесса с некоторым значением параметра **once_control** вызывает **init_routine()** без аргументов.

Последующие вызовы **pthread_once()** с тем же значением **once_control** к вызову **init_routine()** не приводят.

При возврате из **pthread_once()** функция **init_routine()** будет завершена.

Параметр **once_control** определяет, была ли вызвана соответствующая процедура инициализации.

Функция **pthread_once()** не является точкой отмены. Однако, если точкой отмены является **init_routine()** и поток отменяется, действие на **once_control** будет таким, как если бы **pthread_once()** никогда не вызывалась.

Константа **PTHREAD_ONCE_INIT** определена в заголовке **<pthread.h>**.

UB:

Если вызов **init_routine()** завершается вызовом **longjmp()**, **_longjmp()** или **siglongjmp()**, поведение не определено.

Если **once_control** имеет автоматическую продолжительность хранения или не инициализируется значением **PTHREAD_ONCE_INIT**, поведение **pthread_once()** не определено.

Возвращаемое значение

После успешного завершения **pthread_once()** вернет ноль.
В противном случае будет возвращен номер ошибки.

Ошибки

Если **init_routine()** рекурсивно вызывает **pthread_once()** с тем же **once_control**, рекурсивный вызов не будет вызывать указанную **init_routine()** и, таким образом, данная **init_routine()** не завершится и, следовательно, рекурсивный вызов **pthread_once()** не возвратит управление.

EINVAL — если реализация обнаруживает, что значение, указанное аргументом **once_control** для **pthread_once()**, не относится к объекту типа **pthread_once_t**, инициализированному с помощью **PTHREAD_ONCE_INIT**.

Итак, для динамической инициализации библиотеки в многопоточном процессе, если используется флаг инициализации, этот флаг должен быть защищен от модификации несколькими потоками, одновременно вызывающими библиотеку. Это можно сделать с помощью мьютекса, который инициализируется назначением **PTHREAD_MUTEX_INITIALIZER**.

Однако лучшим решением будет использование **pthread_once()**, которая предназначена именно для этой цели, а именно:

```
#include <pthread.h>
static pthread_once_t random_is_initialized = PTHREAD_ONCE_INIT;
extern void initialize_random(void);

int random_function() {
    (void)pthread_once(&random_is_initialized, initialize_random);
    ... // Операции, выполняемые после инициализации
}
```

Статическая инициализация мьютексов и условных переменных

(см. `pthread_once( )`)

Существует возможность статической инициализации статически выделенных объектов синхронизации. Это позволяет модулям с приватными статическими переменными синхронизации избегать проверок инициализации во время выполнения, что снижает накладные расходы.

Кроме того, такой подход упрощает кодирование самоинициализирующихся модулей.

Такие модули распространены в библиотеках C, где по разным причинам дизайн требует самоинициализации вместо того, чтобы требовать явного вызова функции инициализации модуля.

Ниже приводится пример использования такой статической инициализации.

Без статической инициализации самоинициализирующаяся подпрограмма **`foo( )`** может выглядеть следующим образом:

```
static pthread_once_t  foo_once = PTHREAD_ONCE_INIT; // флаг инициализации
static pthread_mutex_t foo_mutex;                    // мьютекс

void foo_init( ) {                          // функция инициализации
    pthread_mutex_init(&foo_mutex, NULL); // инициализация мьютекса
}

void foo( ) {
    pthread_once(&foo_once, foo_init);    // динамическая инициализация подпрограммы
    pthread_mutex_lock(&foo_mutex);       // захват
    // Ту делаем свои дела
    pthread_mutex_unlock(&foo_mutex);     // освобождение
}
```

При статической инициализации ту же процедуру можно было бы закодировать так:

```
static pthread_mutex_t foo_mutex = PTHREAD_MUTEX_INITIALIZER;

void foo() {
    pthread_mutex_lock(&foo_mutex);
    // Ту делаем свои дела
    pthread_mutex_unlock(&foo_mutex);
}
```

Статическая инициализация устраняет необходимость в проверке инициализации внутри **pthread_once()** и выборке **&foo_mutex**, чтобы узнать адрес, который будет передан в **pthread_mutex_lock()** или **pthread_mutex_unlock()**.

Таким образом, код С, написанный для инициализации статических объектов, во всех системах будет проще, а также быстрее в большом классе систем, где (весь) объект синхронизации может храниться в памяти приложения, а не в ядре.

pthread_mutex_lock/trylock/unlock(3p) — заблокировать и разблокировать мьютекс

```
#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); //
```

Объект мьютекса, на который ссылается **mutex**, блокируется вызовом **pthread_mutex_lock()**, который возвращает ноль или ошибку **EOWNERDEAD**.

Если мьютекс уже заблокирован другим потоком, вызывающий поток будет заблокирован до тех пор, пока мьютекс не станет доступным.

Если этот вызов возвращается нормально, объект мьютекса, на который ссылается **mutex**, будет в заблокированном состоянии, а его владельцем будет вызывающий поток.

Робастность

Атрибут робастности определяет поведение мьютекса, когда поток-владелец умирает без разблокировки мьютекса. Для робастности допустимы следующие значения:

PTHREAD_MUTEX_STALLED (значение по умолчанию для объекта атрибутов мьютекса)

Если мьютекс неробастный и его владелец умирает, не разблокировав его, мьютекс остается заблокированным после этого, и любые будущие попытки **вызвать pthread_mutex_lock(3)** на этом мьютексе будут блокироваться на неопределенный срок.

PTHREAD_MUTEX_ROBUST

Если мьютекс робастный и его владелец умирает, не разблокировав его, любые будущие попытки вызвать **pthread_mutex_lock(3)** на этом мьютексе будут успешными и вернут ошибку **EOWNERDEAD**, чтобы указать, что первоначальный владелец больше не существует и мьютекс находится в несогласованном состоянии. В этом случае следующий владелец на полученном мьютексе должен вызвать **pthread_mutex_consistent(3)**, чтобы снова сделать его согласованным, прежде чем его в дальнейшем использовать.

Если поток пытается повторно заблокировать мьютекс, который он уже заблокировал, функция **`pthread_mutex_lock()`** ведет себя, как описано в столбце «Повторная блокировка» в таблице ниже.

Если поток пытается разблокировать мьютекс, который он не заблокировал, или мьютекс, который разблокирован, **`pthread_mutex_unlock()`** будет вести себя, как описано в столбце «Разблокировать, когда не владелец» в таблице.

Тип мьютекса	Робастность	Повторная блокировка	Разблокировка невладельцем
NORMAL	неробастный	deadlock	**поведение не определено**
NORMAL	робастный	deadlock	возвращается ошибка
ERRORCHECK	любая	возвращается ошибка	возвращается ошибка
RECURSIVE	любая	рекурсивное поведение	возвращается ошибка
DEFAULT	неробастный	поведение не определено†	**поведение не определено**†
DEFAULT	робастный	поведение не определено†	возвращается ошибка

Типы мьютексов – NORMAL, ERRORCHECK, RECURSIVE, DEFAULT.

Если тип мьютекса — **`PTHREAD_MUTEX_DEFAULT`**, поведение вызова **`pthread_mutex_lock()`** *может* соответствовать одному из трех других стандартных типов мьютекса, как описано в таблице выше. Если он не соответствует ни одному из этих трех, для случаев, отмеченных **†**, поведение не определено

В случае рекурсивного поведения, мьютекс поддерживает концепцию ***счетчика блокировок***.

Когда поток успешно получает мьютекс в первый раз, счетчик блокировок устанавливается в единицу. Каждый раз, когда поток повторно блокирует этот мьютекс, счетчик блокировок на единицу увеличивается. Каждый раз, когда поток разблокирует мьютекс, счетчик блокировок уменьшается на единицу. Когда счетчик блокировок достигает нуля, мьютекс становится доступным для получения другими потоками.

pthread_mutex_trylock()

Функция **pthread_mutex_trylock()** эквивалентна **pthread_mutex_lock()**, за исключением того, что если объект мьютекса, на который ссылается **mutex**, в настоящее время заблокирован (любым потоком, включая текущий поток), вызов немедленно возвращается.

Если тип мьютекса **PTHREAD_MUTEX_RECURSIVE** и мьютекс в настоящее время принадлежит вызывающему потоку, счетчик блокировок мьютекса будет увеличен на единицу, и вызов функции **pthread_mutex_trylock()** немедленно успешно завершится.

pthread_mutex_unlock()

Функция **pthread_mutex_unlock()** освобождает объект мьютекса, на который ссылается **mutex**.

Способ освобождения мьютекса зависит от атрибута типа мьютекса.

Если есть потоки, заблокированные на объекте мьютекса, на который ссылается **mutex** при вызове **pthread_mutex_unlock()**, то после того, как мьютекс становится доступным, какой поток получит этот мьютекс, определяется политикой планирования.

В случае мьютексов **PTHREAD_MUTEX_RECURSIVE** мьютекс должен стать доступным, когда счетчик достигнет нуля и у вызывающего потока больше нет блокировок на этом мьютексе.

Если потоку, ожидающему мьютекс, доставляется сигнал, по возвращении от обработчика сигнала поток возобновляет ожидание мьютекса, как если бы он не был прерван.

Если мьютекс является робастным мьютексом и *процесс*, содержащий поток-владелец, завершился, удерживая блокировку мьютекса, вызов **pthread_mutex_lock()** вернет значение ошибки **EOWNERDEAD**.

Если мьютекс является робастным мьютексом и *поток-владелец завершился*, удерживая блокировку мьютекса, вызов **pthread_mutex_lock()** *может* вернуть значение ошибки **EOWNERDEAD**, даже если процесс, в котором находится поток-владелец, не завершился.

В этих случаях мьютекс блокируется потоком, но состояние, которое он защищает, помечается как ***несогласованное***.

Ответственность за то, что состояние согласовано и мьютекс может повторно использоваться, несет приложение. После того, как приложение все вернет в согласованное состояние, оно должно вызвать **pthread_mutex_consistent()**.

Если приложение не может восстановить состояние, оно должно разблокировать мьютекс без предварительного вызова **pthread_mutex_consistent()**, после чего мьютекс помечается как постоянно непригодный для использования.

Если мьютекс не относится к инициализированному объекту мьютекса, поведение **pthread_mutex_lock()**, **pthread_mutex_trylock()** и **pthread_mutex_unlock()** не определено.

Возвращаемое значение

В случае успеха функции **pthread_mutex_lock()**, **pthread_mutex_trylock()** и **pthread_mutex_unlock()** возвращают ноль. В противном случае возвращается номер ошибки.

Ошибки (общие)

EAGAIN — Невозможно получить мьютекс, поскольку превышено максимальное количество рекурсивных блокировок для мьютекса.

EINVAL — Мьютекс был создан с атрибутом протокола, имеющим значение **PTHREAD_PRIO_PROTECT**, а приоритет вызывающего потока выше, чем текущий потолок приоритета мьютекса.

ENOTRECOVERABLE — состояние, защищенное мьютексом, восстановить невозможно.

EOWNERDEAD — мьютекс является робастным мьютексом, и процесс, содержащий предыдущий поток-владелец, завершается, удерживая блокировку мьютекса.

В этом случае блокировка мьютекса будет получена вызывающим потоком, и новый владелец должен сделать состояние согласованным.

Ошибки функции `pthread_mutex_lock()`

EDEADLK — тип мьютекса **ERRORCHECK**, и текущий поток уже владеет мьютексом.

Функция **`pthread_mutex_lock()`** ***может*** завершиться ошибкой, если:

EDEADLK — обнаружено состояние взаимоблокировки.

Ошибки функции `pthread_mutex_trylock()`

EBUSY — невозможно получить мьютекс, потому что он уже заблокирован.

Ошибки функции `pthread_mutex_unlock()`

EPERM — тип мьютекса **PTHREAD_MUTEX_ERRORCHECK** или **PTHREAD_MUTEX_RECURSIVE**, или мьютекс является робастным мьютексом, при этом текущий поток мьютексом не владеет.

EOWNERDEAD — мьютекс является робастным мьютексом, и предыдущий поток-владелец завершил работу, удерживая блокировку мьютекса. Блокировка мьютекса будет получена вызывающим потоком, и новый владелец должен сделать состояние согласованным.

Прикладное примерение

Приложения, в которых предполагалось, что ненулевые возвращаемые значения являются ошибками, нуждаются в обновлении, чтобы их можно было использовать с робастными мьютексами, поскольку допускается возврат **EOWNERDEAD** для потока, который получил мьютекс, защищающий текущее несогласованное состояние.

Приложения, которые не проверяют возврат ошибок из-за исключения возможности возникновения таких ошибок, не должны использовать робастные мьютексы.

Если приложение должно работать с обычными и робастными мьютексами, оно должно проверять все возвращаемые значения на наличие ошибок и при необходимости предпринимать соответствующие действия.

Обоснование

Объекты типа «мьютекс» предназначены для использования в качестве низкоуровневого примитива, на основе которого могут быть построены другие функции синхронизации потоков.

Таким образом, реализация мьютексов должна быть максимально эффективной, на сколько это возможно, и иметь альтернативы для функций, доступных в интерфейсе.

Функции мьютексов и особые настройки атрибутов мьютекса по умолчанию были мотивированы желанием не препятствовать быстрым, встроенным реализациям блокировок и разблокировок мьютексов.

Поскольку большинство атрибутов ОС проверяет только тогда, когда поток будет заблокирован, использование атрибутов не замедляет (общий) случай блокировки мьютекса.

Аналогичным образом, хотя возможность извлечения идентификатора потока владельца мьютекса может быть желательной, это может потребовать сохранения идентификатора текущего потока в случае, когда каждый мьютекс заблокирован, и это может привести к недопустимым уровням накладных расходов. Аналогичные аргументы применимы к операции **`mutex_tryunlock()`**.

pthread_mutex_timedlock(3p) — заблокировать с ограниченным ожиданием

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

int pthread_mutex_timedlock(pthread_mutex_t       *restrict mutex,    // мьютекс
                            const struct timespec *restrict abstime); // таймаут
```

Функция **pthread_mutex_timedlock()** блокирует объект мьютекса, на который ссылается **mutex**. Если мьютекс уже заблокирован, вызывающий поток блокируется до тех пор, пока мьютекс не станет доступным, как это имеет место для функции **pthread_mutex_lock()**.

Если по истечении указанного тайм-аута мьютекс не дождался блокирования, ожидание прекращается.

Тайм-аут истекает, когда проходит абсолютное время, указанное в **abstime**, которое измеряется часами, на которых основаны таймауты (то есть, когда значение этих часов равно или превышает **abstime**), или если абсолютное время, указанное в **abstime**, уже было прошло на момент вызова.

Тайм-аут основан на часах **CLOCK_REALTIME**.

Структура **timespec** определена следующим образом:

```
struct timespec {
    time_t tv_sec;      /* секунды */
    long   tv_nsec;     /* наносекунды [0 .. 999999999] */
};
```

Разрешение (resolution) тайм-аута является разрешением часов, на которых он основан. Тип данных **timepec** определяется в заголовке **<time.h>**.

Ни при каких обстоятельствах функция не завершается с ошибкой по таймауту, если мьютекс можно заблокировать немедленно. Правильность параметра **abstime** в случае, если мьютекс можно заблокировать немедленно, не проверяется.

Если мьютекс является робастным мьютексом и процесс, содержащий поток-владелец, завершился, удерживая блокировку мьютекса, вызов **pthread_mutex_timedlock()** вернет значение ошибки **EOWNERDEAD**.

Если мьютекс является робастным мьютексом и поток-владелец завершился, удерживая блокировку мьютекса, вызов **pthread_mutex_timedlock()** *может* вернуть значение ошибки **EOWNERDEAD**, даже если процесс, в котором находится поток-владелец, не завершился.

В этих случаях мьютекс блокируется потоком, но состояние, которое он защищает, помечается как несогласованное.

Приложение должно позаботиться о восстановлении согласованности состояния перед повторным использованием. После того, как состояние согласованности будет востановлено, приложение должно вызвать **pthread_mutex_consistent()**, чтобы сообщить об этом реализации.

Если приложение не может восстановить состояние, оно должно разблокировать мьютекс без предварительного вызова **pthread_mutex_consistent()**, после чего мьютекс помечается как постоянно непригодный для использования.

Если мьютекс не ссылается на инициализированный объект мьютекса, поведение не определено.

Возвращаемое значение

В случае успеха функция **pthread_mutex_timedlock()** возвращает ноль.

В противном случае возвращается номер ошибки.

Ошибки

Функция **pthread_mutex_timedlock()** завершится ошибкой, если:

EAGAIN — невозможно получить мьютекс, поскольку превышено максимальное количество рекурсивных блокировок для мьютекса.

EDEADLK — тип мьютекса **PTHREAD_MUTEX_ERRORCHECK**, и текущий поток уже владеет мьютексом.

EINVAL — мьютекс был создан с атрибутом протокола, имеющим значение **PTHREAD_PRIO_PROTECT**, а приоритет вызывающего потока выше, чем текущий потолок приоритета мьютекса.

EINVAL — процесс или поток были бы заблокированы, а параметр **abstime** определил значение поля наносекунды меньше нуля, либо больше или равно 1000 миллионов.

ENOTRECOVERABLE — состояние, защищенное мьютексом, восстановить невозможно.

EOWNERDEAD — мьютекс является робастным мьютексом, и процесс, содержащий предыдущий поток-владелец, завершается, удерживая блокировку мьютекса.

Блокировка мьютекса будет получена вызывающим потоком, и новый владелец должен сделать состояние согласованным.

ETIMEDOUT — не удалось заблокировать мьютекс до истечения указанного тайм-аута.

Функция **pthread_mutex_timedlock()** *может* завершится ошибкой, если:

EDEADLK — обнаружено состояние взаимоблокировки.

EOWNERDEAD — мьютекс является робастным мьютексом, и предыдущий поток-владелец завершил работу, удерживая блокировку мьютекса. Блокировка мьютекса будет получена вызывающим потоком, и новый владелец должен сделать состояние согласованным.

Атрибуты мьютекса

pthread_mutexattr_init/destroy(3) - инициализирует и уничтожает объект атрибутов мьютекса

```
#include <pthread.h>

int pthread_mutexattr_init(pthread_mutexattr_t *attr);

int pthread_mutexattr_destroy(pthread_mutexattr_t *attr);
```

Функция **pthread_mutexattr_init()** инициализирует объект атрибутов мьютекса, на который указывает **attr**, значениями по умолчанию для всех атрибутов, определенных реализацией.

Результаты инициализации уже инициализированного объекта атрибутов мьютекса не определены.

Функция **pthread_mutexattr_destroy()** уничтожает объект атрибута мьютекса (делая его неинициализированным). После уничтожения объекта атрибутов мьютекса его можно повторно инициализировать с помощью **pthread_mutexattr_init()**.

Результаты уничтожения неинициализированного объекта атрибутов мьютекса не определены.

Возвращаемое значение

В случае успеха эти функции возвращают 0.

В случае ошибки они возвращают положительный номер ошибки.

Ошибки

ENOMEM — недостаточно памяти для инициализации объекта атрибутов мьютекса.

Если реализация обнаруживает, что значение, указанное аргументом **attr** для **pthread_mutexattr_destroy()**, не относится к инициализированному объекту атрибутов мьютекса, функция ***может завершиться*** с ошибкой **EINVAL**.

Атрибуты мьютекса и производительность

Значения атрибутов мьютексов по умолчанию в стандарте определены таким образом, чтобы мьютексы, инициализированные значениями по умолчанию, имели достаточно простую семантику, чтобы блокировку и разблокировку можно было выполнять с помощью эквивалента инструкции проверки и установки (плюс возможно несколько других основных инструкций).

pthread_mutexattr_{get/set}pshared(3) — получить/установить атрибут совместного использования процессами

```
#include <pthread.h>

int pthread_mutexattr_getpshared(const pthread_mutexattr_t *restrict attr,
                                 int                       *restrict pshared);

int pthread_mutexattr_setpshared(pthread_mutexattr_t *attr,
                                 int                  pshared);
```

Эти функции получают и устанавливают атрибут общего доступа к мьютексу из процессов.

Чтобы гарантировать правильную и эффективную работу мьютекса, созданного с использованием объекта атрибутов, этот атрибут должен быть установлен соответствующим образом.

Атрибут общего доступа к процессу может иметь одно из следующих значений:

PTHREAD_PROCESS_PRIVATE Мьютексы, созданные с помощью этого атрибута, должны совместно использоваться только потоками в том же процессе, который инициализировал мьютекс.

Это значение по умолчанию для атрибута мьютекса, который является общим для процесса.

PTHREAD_PROCESS_SHARED

Мьютексы, созданные с помощью этого атрибута, могут совместно использоваться любыми потоками, имеющими доступ к памяти, содержащей объект мьютекса, включая потоки в различных процессах.

Функция **pthread_mutexattr_getpshared()** помещает значение атрибута из объекта атрибутов мьютекса, на который ссылается **attr**, в место, указанное **pshared**.

Функция **pthread_mutexattr_setpshared()** устанавливает значение атрибута в объекте атрибутов мьютекса, на который ссылается **attr**, равным значению, указанному в **pshared**.

Если значение, указанное аргументом **attr** для вызова **pthread_mutexattr_getpshared()** или **pthread_mutexattr_setpshared()**, не относится к инициализированному объекту атрибутов мьютекса, поведение не определено.

Возвращаемое значение

После успешного завершения **pthread_mutexattr_setpshared()** вернет ноль.

В противном случае возвращается номер ошибки.

После успешного завершения **pthread_mutexattr_getpshared()** вернет ноль и сохранит значение атрибута **attr** в объекте, на который ссылается параметр **pshared**.

В противном случае возвращается номер ошибки.

Ошибки

Функция **pthread_mutexattr_setpshared()**:

EINVAL — новое значение, указанное для атрибута, выходит за пределы диапазона допустимых значений для этого атрибута.

pthread_mutexattr_{get/set}robust(3) — получить/установить атрибут робастности[1]

```
// _POSIX_C_SOURCE >= 200809L
#include <pthread.h>

int pthread_mutexattr_getrobust(const pthread_mutexattr_t *attr,
                                int                       *robustness);

int pthread_mutexattr_setrobust(const pthread_mutexattr_t *attr,
                                int                        robustness);
```

Функция **pthread_mutexattr_getrobust()** помещает значение атрибута робастности объекта атрибутов мьютекса, на который ссылается **attr**, в ***robustness**.

Функция **pthread_mutexattr_setrobust()** устанавливает значение атрибута робастности объекта атрибутов мьютекса, на который ссылается робастности, равным значению, заданному в **robustness**.

Атрибут робастности определяет поведение мьютекса, когда поток-владелец умирает без разблокировки мьютекса. Для робастности допустимы следующие значения:

PTHREAD_MUTEX_STALLED
PTHREAD_MUTEX_ROBUST

1) Устойчивость, надежность, незыблемость, ...

PTHREAD_MUTEX_STALLED

Это значение по умолчанию для объекта атрибутов мьютекса. Если мьютекс инициализируется атрибутом **PTHREAD_MUTEX_STALLED** и его владелец умирает, не разблокировав его, мьютекс остается заблокированным после этого, и любые будущие попытки вызвать **pthread_mutex_lock(3)** на мьютексе будут блокироваться на неопределенный срок.

PTHREAD_MUTEX_ROBUST

Если мьютекс инициализируется атрибутом **PTHREAD_MUTEX_ROBUST** и его владелец умирает, не разблокировав его, любые будущие попытки вызвать **pthread_mutex_lock(3)** на этом мьютексе будут успешными и вернут **EOWNERDEAD**, чтобы указать, что *первоначальный владелец больше не существует и мьютекс находится в несогласованном состоянии*.

Обычно после того, как возвращается **EOWNERDEAD**, следующий владелец на полученном мьютексе должен попытаться снова сделать его согласованным, после чего, прежде чем его в дальнейшем использовать, вызвать **pthread_mutex_consistent(3)**.

Если вернуть мьютекс в состояние согласованности не удалось/невозможно, этот владелец должен освободить мьютекс.

Если владелец разблокирует мьютекс с помощью **pthread_mutex_unlock(3)** до восстановления согласованности, мьютекс навсегда останется непригоден для использования, и любые последующие попытки заблокировать его с помощью **pthread_mutex_lock(3)** завершатся ошибкой **ENOTRECOVERABLE**.

Единственная разрешенная операция с таким мьютексом — **pthread_mutex_destroy(3)**.

Если следующий владелец завершает работу до вызова **pthread_mutex_consistent(3)**, дальнейшие операции **pthread_mutex_lock(3)** на этом мьютексе все равно будут возвращать **EOWNERDEAD**.

Аргумент **attr** для **pthread_mutexattr_getrobust()** и **pthread_mutexattr_setrobust()** должен ссылаться на объект атрибутов мьютекса, который был инициализирован **pthread_mutexattr_init(3)**, в противном случае поведение не определено.

Возвращаемое значение

В случае успеха эти функции возвращают 0.

В случае ошибки они возвращают положительный номер ошибки.

В реализации glibc **pthread_mutexattr_getrobust()** всегда возвращает ноль.

Ошибки

EINVAL — в **pthread_mutexattr_setrobust()** было передано значение, отличное от **PTHREAD_MUTEX_STALLED** или **PTHREAD_MUTEX_ROBUST**.

В реализации Linux при использовании робастных мьютексов с общим доступом процессов к мьютексу, если владелец робастного мьютекса выполняет **execve(2)** без предварительной разблокировки мьютекса, ожидающий поток получает уведомление **EOWNERDEAD**.

В POSIX.1 эта деталь не указана, но такое же поведение наблюдается по крайней мере в некоторых других реализациях.

До того, как **pthread_mutexattr_getrobust()** и **pthread_mutexattr_setrobust()** появились в POSIX, в **glibc** были определены следующие эквивалентные нестандартные функции (**_GNU_SOURCE**):

int pthread_mutexattr_getrobust_np(const pthread_mutexattr_t *attr,

```
                                   int *robustness);
int pthread_mutexattr_setrobust_np(const pthread_mutexattr_t *attr,
                                   int robustness);
```

Соответственно, также были определены константы **PTHREAD_MUTEX_STALLED_NP** и **PTHREAD_MUTEX_ROBUST_NP**.

Эти специфичные для GNU API вызовы, которые впервые появились в glibc 2.4, в настоящее время устарели и не должны использоваться в новых программах.

Пример

Программа ниже демонстрирует использование атрибута робастности объекта атрибутов мьютекса.

В этой программе поток, удерживающий мьютекс, преждевременно умирает, не разблокируя мьютекс. Затем основной поток успешно получает мьютекс и получает ошибку **EOWNERDEAD**, после чего делает мьютекс согласованным.

Следующий сеанс оболочки показывает, что мы видим при запуске этой программы:

```
$ ./a.out
[original owner] Setting lock...
[original owner] Locked.
Now exiting without unlocking.
[main thread] Attempting to lock the robust mutex.
[main thread] pthread_mutex_lock() returned EOWNERDEAD
[main thread] Now make the mutex consistent [main thread] Mutex is now consistent;
unlocking
```

Исходный код

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <pthread.h>
#include <errno.h>

#define handle_error_en(en, msg) \
               do { errno = en; perror(msg); exit(EXIT_FAILURE); } while (0)

static pthread_mutex_t mtx;

static void *original_owner_thread(void *ptr) {

    printf("[original owner] Устанавливаю блокировку ...\n");
    pthread_mutex_lock(&mtx);
    printf("[original owner] Заблокировано. Теперь выхожу без разблокирования.\n");
    pthread_exit(NULL);
}

int main(int argc, char *argv[]) {

    pthread_t           thr;
    pthread_mutexattr_t attr;

    pthread_mutexattr_init(&attr);                // initialize the attributes object
    pthread_mutexattr_setrobust(&attr,            // set robustness
                                PTHREAD_MUTEX_ROBUST);
    pthread_mutex_init(&mtx, &attr);              // initialize the mutex
```

```
    pthread_create(&thr, NULL, original_owner_thread, NULL);
    sleep(2);

    // "original_owner_thread" к этому моменту должна завершиться

    printf("[main thread] Пытаюсь заблокировать робастный мьютекс.\n");
    int s = pthread_mutex_lock(&mtx);
    if (s == EOWNERDEAD) {
        printf("[main thread] pthread_mutex_lock() возвратила EOWNERDEAD\n");
        printf("[main thread] Теперь сделаю мьютекс согласованным\n");
        s = pthread_mutex_consistent(&mtx);
        if (s != 0) {
            handle_error_en(s, "pthread_mutex_consistent");
        }
        printf("[main thread] Мьютекс в согласованном состоянии; разблокирую\n");
        s = pthread_mutex_unlock(&mtx);
        if (s != 0) {
            handle_error_en(s, "pthread_mutex_unlock");
        }
        exit(EXIT_SUCCESS);
    } else if (s == 0) {
        printf("[main thread] pthread_mutex_lock() неожиданно сработало\n");
        exit(EXIT_FAILURE);
    } else {
        printf("[main thread] pthread_mutex_lock() неожиданный отказ\n");
        handle_error_en(s, "pthread_mutex_lock");
    }
}
```

pthread_mutex_consistent(3) — делает робастный мьютекс согласованным

```
#include <pthread.h>

int pthread_mutex_consistent(pthread_mutex_t *mutex); // _POSIX_C_SOURCE >= 200809L
```

Компилируется и компонуется вместе с опцией **-pthread**.

Данная функция отмечает робастный мьютекс, как согласованный, если тот находился в несогласованном состоянии.

Если владелец робастного мьютекса завершает работу, удерживая мьютекс, мьютекс становится несогласованным, и следующий поток, который получает блокировку мьютекса, будет уведомлен о состоянии с помощью возвращаемого значения **EOWNERDEAD**. В этом случае мьютекс не может нормально использоваться, пока его состояние не будет помечено как согласованное.

Если поток, получивший блокировку мьютекса с возвращаемым значением **EOWNERDEAD**, завершается до вызова **pthread_mutex_consistent()** или **pthread_mutex_unlock()**, следующий поток, который получает блокировку мьютекса, будет уведомлен о состоянии мьютекса с помощью возвращаемого значения **EOWNERDEAD**.

Функция **pthread_mutex_consistent()** отвечает только за уведомление реализации о том, что состояние, защищенное мьютексом, было восстановлено и что нормальные операции с мьютексом могут быть возобновлены.

Ответственность за восстановление общих данных в согласованном состоянии перед вызовом pthread_mutex_consistent() лежит на приложении.

Если приложение не может выполнить восстановление, оно может уведомить реализацию о том, что ситуация не может быть восстановлена, путем вызова **pthread_mutex_unlock()** без предварительного вызова **pthread_mutex_consistent()**. В этом случае последующие потоки, пытающиеся заблокировать мьютекс, потерпят неудачу и им будет возвращено **ENOTRECOVERABLE**.

Если значение, указанное аргументом мьютекса для **pthread_mutex_consistent()**, не относится к инициализированному мьютексу, поведение не определено.

Возвращаемое значение

При успешном выполнении **pthread_mutex_consistent()** возвращает 0.

В противном случае возвращается положительный номер, указывающий на причину ошибки.

Ошибки

EINVAL — мьютекс находится не в робастном состоянии

EINVAL — мьютекс не защищает рассогласованное состояние

pthread_mutex_consistent() просто сообщает реализации, что состояние (общие данные), охраняемое мьютексом, было восстановлено до согласованного состояния и что теперь с мьютексом можно выполнять обычные операции.

pthread_mutexattr_{get|set}type(3p) — получить/установить атрибут типа

#include <pthread.h>

```
int pthread_mutexattr_gettype(const pthread_mutexattr_t *restrict attr,
                              int                       *restrict type);

int pthread_mutexattr_settype(pthread_mutexattr_t *attr,
                              int                  type);
```

Функции **pthread_mutexattr_gettype()** и **pthread_mutexattr_settype()**, соответственно, должны получать и устанавливать атрибут типа мьютекса.

Значение атрибута **type** по умолчанию — **PTHREAD_MUTEX_DEFAULT**.

Тип мьютекса содержится в атрибуте **type** атрибутов мьютекса. Допустимые типы мьютексов включают:

PTHREAD_MUTEX_NORMAL
PTHREAD_MUTEX_ERRORCHECK
PTHREAD_MUTEX_RECURSIVE
PTHREAD_MUTEX_DEFAULT

Тип мьютекса влияет на поведение вызовов, которые блокируют и разблокируют мьютекс.

Если поток пытается разблокировать мьютекс, который он не заблокировал, или мьютекс, который разблокирован, **`pthread_mutex_unlock()`** будет вести себя, как описано в столбце «Разблокировать, когда не владелец» в таблице.

Тип мьютекса	Робастность	Повторная блокировка	Разблокировка невладельцем
NORMAL	неробастный	deadlock	поведение не определено
NORMAL	робастный	deadlock	возвращается ошибка
ERRORCHECK	любая	возвращается ошибка	возвращается ошибка
RECURSIVE	любая	рекурсивное поведение	возвращается ошибка
DEFAULT	неробастный	поведение не определено†	поведение не определено†
DEFAULT	робастный	поведение не определено†	возвращается ошибка

Если тип мьютекса — **`PTHREAD_MUTEX_DEFAULT`**, поведение вызова **`pthread_mutex_lock()`** может соответствовать одному из трех других стандартных типов мьютекса, как описано в таблице выше. Если он не соответствует ни одному из этих трех, для случаев, отмеченных †, поведение не определено

В случае рекурсивного поведения, мьютекс поддерживает концепцию счетчика блокировок.

Когда поток успешно получает мьютекс в первый раз, счетчик блокировок устанавливается в единицу. Каждый раз, когда поток повторно блокирует этот мьютекс, счетчик блокировок на единицу увеличивается. Каждый раз, когда поток разблокирует мьютекс, счетчик блокировок уменьшается на единицу. Когда счетчик блокировок достигает нуля, мьютекс становится доступным для получения другими потоками.

Реализация может отображать `PTHREAD_MUTEX_DEFAULT` в один из других типов мьютексов.

Если значение, указанное аргументом **attr** для **pthread_mutexattr_gettype()** или **pthread_mutexattr_settype()**, не относится к инициализированному объекту атрибутов мьютекса, поведение не определено.

После успешного завершения функция **pthread_mutexattr_gettype()** возвращает ноль и сохраняет значение атрибута **type** для **attr** в объекте, на который ссылается параметр **type**. В противном случае возвращается ошибка.

В случае успешного завершения функция **pthread_mutexattr_settype()** возвращает ноль.

В противном случае возвращается ошибка.

Возвращаемое значение

После успешного завершения функции **pthread_mutexattr_gettype()** возвращают ноль и сохраняет значение атрибута **type** для **attr** в объекте, на который ссылается параметр типа.

В противном случае возвращается номер ошибки.

После успешного завершения функции **pthread_mutexattr_settype()** возвращают ноль.

В противном случае возвращается номер ошибки.

Ошибки

EINVAL — Тип значения недействителен.

Планирование синхронизации

Управление приоритетом потоков требуется в том случае, если существуют ограничения, особенно ограничения по времени, требующие, чтобы некоторые потоки выполнялись быстрее, чем остальные.

С помощью объектов синхронизации, например, мьютексов, можно приостановить выполнение даже потоков с высоким приоритетом. При этом в некоторых случаях может возникнуть нежелательный эффект, который называется *инверсией приоритетов*. Для его предотвращения в библиотеке pthreads предусмотрено специальное средство — протоколы взаимных блокировок.

Планирование при синхронизации определяет изменение параметров планирования, в частности, приоритета, при взаимной блокировке. Это позволяет программисту управлять параметрами планирования и предотвратить изменение приоритетов, что весьма полезно при работе со сложными схемами блокировки.

В некоторых реализациях библиотеки планирование при синхронизации отсутствует.

Инверсия приоритетов

Инверсия приоритетов возникает тогда, когда поток с низким приоритетом захватывает мьютекс, приостанавливая выполнение потока с высоким приоритетом. Такой поток может неограниченное время сохранять мьютекс, так как его приоритет низкий. Из-за этого не удается обеспечить выполнение потока за заданный срок.

Есть два протокола, которые позволяют избежать инверсии приоритетов.

- протокол наследования приоритета;
- протокол защиты приоритета.

Протокол наследования приоритета

Иногда этот протокол называется базовым протоколом наследования приоритета.

В соответствии с этим протоколом владелец мьютекса наследует приоритет, который является *максимальным среди приоритетов заблокированных потоков*.

Если поток блокируется при попытке захватить мьютекс, владелец этого мьютекса временно получает приоритет блокированного потока, если он больше его собственного приоритета. Это позволяет потоку, владеющему мьютексом, его освободить.

При освобождении мьютекса приоритет бывшего владельца восстанавливается.

Протокол защиты приоритета

Иногда этот протокол называется эмуляцией протокола максимального приоритета.

В протоколе защиты приоритетов с каждым мьютексом связан максимальный приоритет.

Это некоторое допустимое значение приоритета. Когда поток захватывает мьютекс, ему временно присваивается *максимальный приоритет этого мьютекса*, если он превышает ее собственный приоритет.

При освобождении взаимной блокировки приоритет владельца восстанавливается.

Максимальный приоритет должен равняться наибольшему значению среди приоритетов потоков, которые могут захватить мьютекс. Если его значение будет меньше, то протокол не будет защищать от инверсии приоритетов и тупиков.

Оба протокола увеличивают приоритет потока, захватившего мьютекс, что позволяет гарантировать выполнение потока за указанный срок.

Кроме того, протоколы взаимных блокировок предотвращают появление тупиков.

С каждым мьютексом связывается свой протокол взаимных блокировок.

Выбор протокола взаимных блокировок

Протокол взаимных блокировок настраивается при создании мьютекса с помощью специального атрибута протокола. Этот атрибут можно настроить в объекте атрибутов мьютекса с помощью функций **pthread_mutexattr_getprotocol()** и **pthread_mutexattr_setprotocol()**.

Допустимы следующие значения атрибута протокола:

PTHREAD_PRIO_DEFAULT — (**PTHREAD_PRIO_INHERIT**);
PTHREAD_PRIO_NONE — Протокол не выбран;
PTHREAD_PRIO_INHERIT — Протокол наследования приоритетов;
PTHREAD_PRIO_PROTECT — Протокол защиты приоритетов.

Поведение **PTHREAD_PRIO_DEFAULT** такое же, как у атрибута **PTHREAD_PRIO_INHERIT**.

С помощью ссылки на мьютекс, потоки работают с атрибутом по умолчанию, который временно повышает приоритет владельца мьютекса, когда пользователь заблокирован и имеет более высокий приоритет, чем владелец.

В протоколе защиты приоритетов требуется настроить дополнительный атрибут, который задает максимальный приоритет мьютекса. Этот атрибут можно настроить в объекте атрибутов мьютекса с помощью **pthread_mutexattr_getprioceiling()** и **pthread_mutexattr_setprioceiling()**.

При увеличении приоритета потока требуется динамическое изменение максимального приоритета мьютекса. Для динамического изменения максимального приоритета мьютекса служат функции **pthread_mutex_getprioceiling()** и **pthread_mutex_setprioceiling()**.

Для предотвращения тупика мьютекс не должен принадлежать потоку, вызывающему функцию pthread_mutex_setprioceiling().

Реализации протоколов взаимных блокировок относятся к дополнительным функциям системы. Эти протоколы являются компонентами POSIX.

Выбор протокола наследования или защиты

Оба протокола выполняют одинаковые функции и основаны на увеличении приоритета потока, захватывающего мьютекс. При наличии обоих протоколов нужно выбрать один из них.

Выбор протокола зависит от того, известны ли приоритеты потоков, захватывающих мьютекс, тому программисту, который этот мьютекс создает.

Обычно мьютексы, которые определены в библиотеке и применяются прикладными потоками, работают с протоколом наследования, а мьютексы, созданные в прикладных программах, работают с протоколом защиты.

На выбор могут повлиять и повышенные требования к производительности программы – в большинстве реализаций, для изменения приоритета потока необходимо выполнить системный вызов.

Протоколы мьютексов различаются по числу генерируемых системных вызовов:

- В *протоколе наследования* системный вызов выполняется каждый раз, когда поток блокируется при попытке захватить мьютекс.
- В *протоколе защиты* системный вызов выполняется при каждом захвате мьютекса потоком.

В общем случае для повышения производительности программы рекомендуется выбрать протокол наследования, так как нити конкурируют за мьютекс достаточно редко.

Мьютексы захватываются на небольшие периоды времени, поэтому вероятность блокировки потока при попытке захвата мьютекса достаточно низкая.

pthread_mutexattr_{get|set}protocol(3p) - получить и установить атрибут протокола

#include <pthread.h>

```
int pthread_mutexattr_getprotocol(const pthread_mutexattr_t *restrict attr,
                                  int                       *restrict protocol);

int pthread_mutexattr_setprotocol(pthread_mutexattr_t *attr,
                                  int                  protocol);
```

Функции **pthread_mutexattr_getprotocol()** и **pthread_mutexattr_setprotocol()** соответственно получают и устанавливают атрибут протокола объекта атрибутов мьютекса, на который указывает **attr**, ранее созданного функцией **pthread_mutexattr_init()**.

Атрибут протокола определяет протокол, которому необходимо следовать при использовании мьютексов. Значение протокола может быть одним из:

PTHREAD_PRIO_INHERIT — наследовать
PTHREAD_PRIO_NONE — значение по умолчанию.
PTHREAD_PRIO_PROTECT — защитить

Значения атрибута определены в заголовке **<pthread.h>** и по умолчанию устанавливается в **PTHREAD_PRIO_NONE**.

Когда поток владеет мьютексом с атрибутом протокола **PTHREAD_PRIO_NONE**, его приоритет и планирование доступа к мьютексу не зависят от факта владения мьютексом.

Если поток блокирует потоки с более высоким приоритетом из-за того, что владеет одним или несколькими робастными мьютексами с атрибутом протокола **PTHREAD_PRIO_INHERIT**, он будет выполняться с более высоким приоритетом или приоритетом потока с наивысшим приоритетом из потоков, которые ожидают на любом из робастных мьютексов, принадлежащих данному потоку, и инициализированных этим протоколом.

Если поток блокирует потоки с более высоким приоритетом из-за того, что владеет одним или несколькими неробастными мьютексами с атрибутом протокола **PTHREAD_PRIO_INHERIT**, он будет выполняться с приоритетом, который будет наивысшим из своего приоритета или приоритета потока с наивысшим приоритетом из потоков, которые ожидают на любом из неробастных мьютексов, принадлежащих этому потоку и инициализированых данным протоколом.

Если поток владеет одним или несколькими неробастными мьютексами, инициализированными протоколом **PTHREAD_PRIO_PROTECT**, он должен выполняться с наивысшим из своего приоритета или наивысшим из максимальных значений приоритета всех неробастных мьютексов, принадлежащих этому потоку и инициализированных этим атрибутом независимо от того, заблокированы ли другие потоки на каком-либо из этих неробастных мьютексов или нет.

Пока поток удерживает мьютекс, который был инициализирован атрибутами протокола **PTHREAD_PRIO_INHERIT** или **PTHREAD_PRIO_PROTECT**, он не должен перемещаться в конец очереди планирования его приоритета (доступа к мьютексу) в случае изменения своего исходного приоритета, например, с помощью вызова **sched_setparam()**.

Аналогичным образом, когда поток разблокирует мьютекс, который был инициализирован с помощью атрибутов протокола **PTHREAD_PRIO_INHERIT** или **PTHREAD_PRIO_PROTECT**, он не должен перемещаться в конец очереди планирования его приоритета в случае изменения своего исходного приоритета.

Если поток одновременно владеет несколькими мьютексами, инициализированными разными протоколами, он должен выполняться с наивысшим из приоритетов, которые он получил бы от каждого из этих протоколов.

Когда поток вызывает **pthread_mutex_lock()**, а мьютекс был инициализирован атрибутом протокола, имеющим значение **PTHREAD_PRIO_INHERIT**, когда вызывающий поток блокируется из-за того, что мьютекс принадлежит другому потоку, этот поток-владелец наследует уровень приоритета вызывающего потока, пока тот продолжает владеть мьютексом.

Если значение, указанное аргументом **attr** для **pthread_mutexattr_getprotocol()** или **pthread_mutexattr_setprotocol()**, не относится к инициализированному объекту атрибутов мьютекса, поведение не определено.

Возвращаемое значение

После успешного завершения функции **pthread_mutexattr_getprotocol()** и **pthread_mutexattr_setprotocol()** возвращают ноль.

В противном случае возвращается номер ошибки.

Ошибки

Функция **pthread_mutexattr_setprotocol()** завершится ошибкой:

ENOTSUP — значение, указанное протоколом, не поддерживается.

Функции **pthread_mutexattr_getprotocol()** и **pthread_mutexattr_setprotocol()** *могут* завершиться ошибкой:

EINVAL — значение, указанное протоколом, недействительно.

EPERM — вызывающий не имеет права выполнять операцию.

pthread_mutexattr_{get|set}prioceiling(3p) — получить/установить атрибут потолка приоритетов

```
#include <pthread.h>

int pthread_mutexattr_getprioceiling(const pthread_mutexattr_t *restrict attr,
                                     int                       *restrict prioceiling);

int pthread_mutexattr_setprioceiling(pthread_mutexattr_t *attr,
                                     int                  prioceiling);
```

Функции **pthread_mutexattr_getprioceiling()** и **pthread_mutexattr_setprioceiling()**, соответственно, получают и устанавливают *атрибут верхнего предела приоритета* в объекте атрибутов мьютекса, на который указывает **attr**, который ранее был создан функцией **pthread_mutexattr_init()**.

Атрибут **prioceiling** содержит потолок приоритета инициализированных мьютексов.

Значения **prioceiling** находятся в пределах максимального диапазона приоритетов, определенного **SCHED_FIFO**.

Атрибут **prioceiling** определяет потолок приоритета инициализированных мьютексов, который является минимальным уровнем приоритета, на котором выполняется критическая секция, охраняемая мьютексом.

Чтобы избежать инверсии приоритета, верхний предел приоритета мьютекса должен быть установлен на приоритет выше или равный наивысшему приоритету всех потоков, которые могут блокировать этот мьютекс.

Если значение, указанное аргументом **attr** для **pthread_mutexattr_getprioceiling()** или **pthread_mutexattr_setprioceiling()**, не относится к инициализированному объекту атрибутов мьютекса, поведение не определено.

Возвращаемое значение

При успешном завершении функции **pthread_mutexattr_getprioceiling()** и **pthread_mutexattr_setprioceiling()** возвращают ноль.

В противном случае возвращается номер ошибки.

Ошибки

EINVAL — значение, указанное **prioceiling**, недопустимо.

EPERM — вызывающий не имеет права выполнять операцию.