ФЕДЕРАЛЬНОЕ АГЕНТСТВО ПО ОБРАЗОВАНИЮ
РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное образовательное учреждение
высшего профессионального образования

«ЮЖНЫЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

Методические указания
на тему:
«Параллельное программирование:
POSIX® thread»

Ростов-на-Дону
2008

Методические указания разработаны кандидатом физико-математических наук, доцентом кафедры информатики и вычислительного эксперимента В.А. Савельевым.

Компьютерный набор и вёрстка автора.

Печатается в соответствии с решением кафедры информатики и вычислительного эксперимента факультета математики, механики и компьютерных наук (мехмата) ЮФУ, протокол №  от «  »  2008 года.

# Введение

В этих методических указаниях рассматривается наиболее часто используемая библиотека для работы с нитями — POSIX® threads, известную также как Pthreads. Описание её впервые вошло в стандарт POSIX 1003.1c-1995. Pthreads является системым API для нитей в операционных системах семейства BSD, откуда оно в основном и существенном позаимствовано, и в POSIX-совместимых системах, в том числе в операционной системе Linux. Сегодня используется расширенная редакция стандарта — POSIX 1003.1-2001.

Для систем, использующих другие API для управления нитями, как правило существует библиотека эмуляции Pthreads. Например, в SunOS и Solaris, где сама операционная система использует значительно более сложный и гибкий API для работы с нитями, библиотека эмуляции Pthreads входит в поставку системы.

Для MS Windows также существует библиотека эмуляции, но в операционную систему она не входит. Установка этой библиотеки рассматривается в приложении.

Как и другие интерфейсы управления нитями универсальных операционных систем, pthreads не предназначались для распараллеливания численных задач. Предполагалось, что Pthreads Будет использоваться, чтобы

1. более эффективно использовать естественную параллельность программы,
2. позволяя продолжать вычисления во время ожидания медленных или блокирующих операций;
3. проектировать более модульные программы, явно выражая независимость тех или иных событий в программе;
4. эффективно использовать параллелизм на многпроцессорных системах.

В областью использования pthreads входят задачи системного программирования, а также программирование разного рода асинхронных взаимодействий.

Однако чрезвычайная распространенность этого API и возможность относительно легко создавать параллельные программы без привлечения дополнительных средств (как технических, так и денежных), привело к широкому его использованию и в научном программировании.

Распространенность Pthreads привела к появлению большого количества руководств начального уровня, при явной нехватке более полных учебников, что отчасти компенсировалось доступностью текста самой спецификации. На русском языке выделяется книга [5], содержащая описание pthreads с точки зрения научного программирования и классическое руководство У. Ричарда Стивенса [4], рассматривающее Pthread в широком контексте взаимодействий процессов и сетевого программирования.

В этих методических указаниях мы тоже рассматриваем спецификацию POSIX® threads не в полном объеме, опустив ряд специальных вопросов, в частности настройку планировщика реального времени и особенности программирования нитей в процессах реального времени, обработку вызова `fork()`, обработчики очистки и полное описание атрибутов объектов.

Для более глубокого изучения pthreads можно порекомендовать саму спецификацию [2], а также книгу [3], написанную Дэвидом Батенхофом — одним из создателей спецификации POSIX 1003.1c-1995.

## 1 Типы данных pthread

POSIX® threads определяет набор типов данных доступных прикладному программисту. Все эти типы данных (включая `pthread_t`) являются «прозрачными», т.е. это дескрипторы (handles), используемые для ссылок на вну-

тренние структуры данных pthreads. Даже если в конкретной реализации они созданы на основе целых типов, не следует считать, что это числа или указатели. Возможны ситуации когда для ссылок на один и тот же объект используются неравные дескрипторы.

*Таблица 1: Типы данных pthreads*

| Тип | Описание |
|---|---|
| `pthread_t` | идентификатор нити |
| `pthread_mutex_t` | мьютекс |
| `pthread_cond_t` | условная переменная |
| `pthread_spinlock_t` | спин-блокировка (2001) |
| `pthread_rwlock_t` | блокировка записи/чтения (2001) |
| `pthread_barrier_t` | барьер (2001) |
| `pthread_key_t` | ключ доступа |
| `pthread_attr_t` | атрибут нити |
| `pthread_mutexattr_t` | атрибут мьютекса |
| `pthread_condattr_t` | атрибут условной переменной |
| `pthread_rwlockattr_t` | атрибуты блокировки записи/чтения (2001) |
| `pthread_barrierattr_t` | атрибуты барьера (2001) |
| `pthread_once_t` | контекст однократной инициализации |

Отметим, что большинство функций Pthreads возвращают целое значение. При успешном выполнении они возвращают 0, при ошибке возвращается код ошибки. В отличии от большинства других функций POSIX функции Pthreads не меняют переменную errno.

Значения собственных типов данных Pthread передаются в библиотеке через параметры.

# 2 Управление нитями

## 2.1 Жизнєнный цикл нити

Выполнение любой программы начинается с создания процесса с единственной нитью управления. Эта нить имеет возможность создавать другие нити, явно указывая *функцию нити* которая и будет выполняться в новой нити. Новые нити начинают вместе со всеми остальными нитями в операционной системе распределяться по процессорам, переходя между состояниями нити (см. Табл. 2) в соответствии с диаграммой переходов на рис.~\ref{fig:pthreads-states}. В большинстве реализаций можно считать, что ограничений на количество создаваемых нитей нет. По крайней мере, все реализации позволяют создавать нитей больше, чем имеется процессоров в системе.

Спецификация pthreads предполагает, что вся память процесса, включая стеки нитей (следовательно локальные переменные), кучу памяти, и, что естественно, глобальные переменные, является разделяемой. Поэтому в pthread включена поддержка *локальной памяти нити*.

При выходе из функции нити или при вызове из нити функции pthread_exit(), или при отмене нити из другой нити, нить завершается (переходит в состояние «Terminated»). При этом происходит вызов деструкторов локальных данных и обработчиков очистки, что частично освобождает ресурсы нити.

Только *отсоединенные* нити (см. раздел 2.5) при переходе в состояние «Terminated» сразу уничтожаются и освобождают все ресурсов.

Остальные функции остаются в состоянии «Terminated» пока какая-либо нить, продолжающая выполняться, не попытается присоединить завершённую нить к себе.

*Таблица 2: Состояния нити*

| Состояние | Описание |
|---|---|
| Ready | Нить готова к выполнению, но ждет доступа к процессору. |
| Running | Нить в настоящий момент выполняется на процессоре. На многопроцессорных или многонитевых (SMT/HyperThreading®) системах может быть более одной выполняющейся нити. |
| Blocked | Нить не может выполняться, так как ждет наступления какого-либо события. Например, захвата мьютекса или завершения операции ввода/вывода. |
| Terminated | Нить завершилась или была отменена, но не была ни ранее отсоединена, ни после этого присоединена. |

### 2.2 Создание нити

Одна нить всегда создается операционной системой при запуске процесса. Для создания дополнительных нитей используется функция `pthread_create`.

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

Ей передается:

`thread` указатель на переменную типа `pthread_t`, в которой, при успешном создании нити, будет размещен дескриптор нити;

`attr` константный указатель на атрибуты нити (могут быть использованы повторно после создания нити), для создания нити со свойствами по умолчанию передается `NULL`;

`start` указатель на функцию нити, тело которой будет выполняться нитью;

`arg` указатель на параметр нити, то есть данные, которые будут переданы в функцию нити, вызванную в данном экземпляре нити.

Поток создается запуском функции `start()` с единственным аргументом `arg`. Если функция `start()` завершается, эффект будет как от неявного вызова функции `pthread_exit()` с использованием возвращаемого значения функции `start()` как кода возврата. Нить в которой выполняется функция `main()` отличается от других нитей — если осуществляется возврат из функции `main()`, происходит неявный вызов функции `exit()` с использованием в качестве кода выхода значения, которое вернула функция `main()`. Заметим, что вызов (явный или неявный) функции `exit()` приводит к завершению всех нитей процесса.

Состояние обработки сигналов в новой нити устанавливается следующим образом:

- маска сигналов наследуется у родительской нити;
- набор ожидающих обработки (pending) сигналов пуст.

Стек не наследуется у родительской нити.

Среда выполнения операций с плавающей запятой наследуется у родительской нити.

Функция возвращает целое значение, равное нулю в случае успешного завершения или, в противном случае, код ошибки:

`EAGAIN` системе не хватает необходимых для создания потока ресурсов, или достигнуто ограничение на количество создаваемых в процессе нитей, заданное константой `PTHREAD_THREADS_MAX`.

`EPERM` вызывающее приложение не имеет прав для установки или изменения политики планирования.

`EINVAL` ошибочные атрибуты `attr`.

Если возникла ошибка, новый поток не создается и значение `*thread` не изменяется.

Вот так выглядит простейший способ создания новой нити:

```
pthread_t thread;

void *thread_func(void *data) {
   /* тело функции выполняющейся в новой нити ... */
}

int main()
{   pthread_t id;
    /* ... */
    pthread_create(&id, NULL, &thread_func, NULL);
    /* ... */
}
```

Программирование с нитями требует наличия вспомогательных функций.

Функция `pthread_self` возвращает дескриптор нити, в которой эта функция была вызвана:

```
pthread_t pthread_self(void);
```

Ошибки для функции `pthread_self()` не определены.

Поскольку в реализациях возможно, чтобы одной нити соотвествовали разные дескрипторы, для проверки совпадения нитей используется функция `pthread_equal`.

```
int pthread_equal(pthread_t t1, pthread_t t2);
```

Функция pthread_equal() возвращает ненулевое значение, если дескрипторы обозначают одну и ту же нить, и нуль в противном случае. В случае некорректных дескрипторов нити поведение функции не определено. Ошибки для этой функции не определены.

### 2.3 Завершение нити

Нить завершается (переходит в состояние «Terminated») как по выходу из функции нити, так и по вызову следующей функции:

```
int pthread_exit(void *value_ptr);
```

Однако уничтожение нити и освобождение занимаемых ею ресурсов происходит только для отсоединенных нитей (см. раздел 2.5). Все остальные

нити для завершения своего существования должны быть присоединены (см. раздел 2.4) к одной из продолжающих выполняться нитей.

### 2.4 Ожидание завершения (присоединение) нити

Следующая функция

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

позволяет текущей нити ожидать завершения нити с дескриптором `thread`. Успешное возвращение управления функцией `pthread_join()` означает, что нить с дескриптором `thread` уже завершилась. Второй параметр позволяет получить код возврата завершившийся нити. Если нить была отменена, всегда в качестве кода возвращается значение `PTHREAD_CANCELLED`.

Функция возвращает нуль в случае успешного присоединения, иначе она возвращает код ошибки:

`EINVAL` дескриптор `thread` указывает на нить, которая не может быть присоединена;

`ESRCH` нет нити с таким дескриптором `thread`;

`EDEADLK` обнаружен тупик (deadlock) или нить пытается завершить себя.

Отметим, что завершения отсоединенных нитей *нельзя* ожидать.

Приведем пример присоединения созданных нитей из спецификации Pthread:

```
typedef struct {
    int *ar;
    long n;
} subarray;

void *
incer(void *arg)
{
    long i;
    for (i = 0; i < ((subarray *)arg)->n; i++)
        ((subarray *)arg)->ar[i]++;
}
```

```
int main(void)
{
    int        ar[1000000];
    pthread_t  th1, th2;
    subarray   sb1, sb2;

    sb1.ar = &ar[0];
    sb1.n  = 500000;
    (void) pthread_create(&th1, NULL, incer, &sb1);

    sb2.ar = &ar[500000];
    sb2.n  = 500000;
    (void) pthread_create(&th2, NULL, incer, &sb2);

    (void) pthread_join(th1, NULL);
    (void) pthread_join(th2, NULL);
    return 0;
}
```

### 2.5 Отсоединение нити

Отсоединение нити позволяет освободить занятые ею ресурсы сразу после её завершения. Кроме того, другая нить уже не может ожидать завершения отсоединенной нити. Другого влияния на выполнение нити отсоединение не оказывает.

Нить можно также сразу создать отсоединенной, передав функции `pthread_create()` соответствующие атрибуты.

Если же создана обычная нить, её можно отсоединить, вызвав из другой нити следующую функцию:

```
int pthread_detach(pthread_t thread)
```

и передав ей дескриптор отсоединяемой нити.

Функция возвращает нуль, если отсоединение нити прошло успешно, и, в противном случае возвращает код ошибки:

`EINVAL` переданный дескриптор не является дескриптором нити, которую можно присоединить;

`ESRCH` нет нити с таким дескриптором.

Заметим, что каждая созданная в программе нить должна быть либо отсоединена, либо присоединена. Нить, которая была отсоединена, нельзя присоединять.

### 2.6 Перепланирование нити

```
#include <sched.h>
int sched_yield(void);
```

Эта функция, определенная в файле sched.h, вытесняет (снимает) нить с процессора и вызывает планировщик, позволяя системе отдать процессор более приоритетной нити ещё до завершения кванта времени текущей нити.

Функция возвращает ноль, если выполнилась успешно, и -1 в противном случае. Код ошибки доступен через `errno`.

Применение этой функции в системах с вытесняющей многозадачностью вызывает ожесточенные споры между программистами, так как хотя её регулярные вызовы являются *необходимостью* в системах с кооперативной многозадачностью, с точки зрения логики программы вызов этой функции эквивалентен пустому оператору. В системе с вытеснением нитей такой вызов соответствует признанию, что программе больше нечего делать.

Ясно, что в вычислительной задаче использование функции `sched_yield()` не имеет смысла. Осмысленное её использование возможно в серверных приложениях, когда завершение обработки запроса действительно может оставить приложение на какое-то время без работы и оно уступает процессор не дожидаясь конца кванта времени.

### 2.7 Отмена нити

В pthread существует возможность запросить из одной нити отмену (принудительное завершение) другой нити. Для этого вызывают функцию

`pthread_cancel()` передавая ей дескриптор нити, которую нужно отменить.

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

В случае успеха функция возвращает нуль, в противном случае возвращается код ошибки:

ESRCH не удалось найти нить, соответствующую переданному дескриптору.

После этого следует дождаться завершения нити, если она не отсоединена.

Часто случается, что по логике работы программы нить нельзя отменять или можно, но не всегда. Поэтому допускается не выполнять запрос на отмену или выполнять его с задержкой. Ответ нити на попытку отмены зависит от типа нити.

- *Отменяемая* нить может быть отменена другой нитью. Отмена может происходить по разному. Выделяют:
- *Асинхронно отменяемую* нить, которая может быть отменена в любой точке своего выполнения. Следовательно и в любой момент времени.
- *Синхронно отменяемую* нить, которая может быть отменена только в определенных точках своего выполнения. Запрос на отмену ставиться в очередь, ожидать достижения отменяемой нитью ближайшей из таких точек.
- *Неотменяемая* нить неуязвима к попыткам других нитей отменить её. Все попытки отмены игнорируются.

«Отношение» нити к попыткам отмены можно динамически изменить во время выполнения нити с помощью следующих функций:

```
int pthread_setcancelstate(int state,
    int *oldstate);
int pthread_setcanceltype();(int type,
    int *oldtype);
```

Здесь первый параметр содержит новое состояние, а второй возвращает предыдущее состояние. В качестве значений первого параметра, следует использовать константы из табл..

Значения параметров state/type

| Параметр | Имя константы |
|---|---|
| state | PTHREAD_CANCEL_ENABLE |
| | PTHREAD_CANCEL_DISABLE |
| type | PTHREAD_CANCEL_DEFERRED |
| | PTHREAD_CANCEL_ASYNCHRONOUS |

Для нитей, допускающих синхронную отмену, необходимо указать точки, в которых эту нить можно отменить. В таких точках необходимо сделать вызовы следующей функции:

```
void pthread_testcancel(void);
```

Если была выполнена отмена для данной нити, функция `pthread_testcancel()` извлечет из очереди сообщение об отмене и завершит нить.

Не рекомендуется отменять нить, если её можно просто завершить. Отмена нити — аварийное средство. Нормальный способ завершить другую нить — каким-либо образом (флагом, условной переменной, ...) просигнализировать ей о необходимости завершения и дождаться штатного завершения.

### 2.8 Однократная инициализация

Для однократной оптимизации используется функция

```
int pthread_once(pthread_once_t *once_control,
    void (*init_routine) (void));
```

Эта функция исполняет `init_routine()` лишь при первом вызове для данного `once_control`, вне зависимости от того из какой нити процесса `pthread_once()` вызвана. Для возврата из первого вызова `pthread_once()` сначала должна завершиться функция `init_routine()`. Повторные вызовы с тем же `once_control` ничего не делают.

Сама по себе функция `pthread_once()` не является точкой отмены нити, но такой точкой может (но не обязана) быть функция `init_routine()`. Если функция `init_routine()` является точкой отмены и отмена произошла при её выполнении, эффект выполнения функции `pthread_once()` не наступает, то есть считается, что в этой нити `pthread_once()` не вызывалась.

Переменная `once_control` должна быть статической и инициализирована перед использованием с помощью статического инициализатора `PTHREAD_ONCE_INIT`. Если эта переменная размещена в автоматической памяти или не инициализирована, поведение функции не определено.

При успешном выполнении функция возвращает нуль, в противном случае возвращается код ошибки:

EINVAL параметры некорректны.

Вот фрагмент с простым примером использования этой функции:

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

int random_function()
{
  (void)pthread_once(&random_is_initialized,
initialize_random) ;
  ... /* Operations performed after initialization. */
}
```

### 2.9 Атрибуты нити

Операционные системы позволяют менять свойства нитей управления в достаточно широких пределах. Pthreads для изменения свойств создаваемой нити использует механизм атрибутов.

Следующая функция создает атрибут нити и инициализирует его по умолчанию:

```
int pthread_attr_init(pthread_attr_t *attr);
```

Созданный атрибут может быть использован при создании любого количества нитей. После создания нити атрибут никак с ней не связан и может быть без опасения изменен и использован для создания других нитей.

После создания нитей атрибут больше не нужен и может быть уничтожен функцией:

```
int pthread_attr_destroy(pthread_attr_t *attr);
```

Наиболее часто используемый атрибут определяет, создается ли поток отсоединенным. С каждым атрибутом работают две функции, позволяющие, соответственно, установить и проверить значение атрибута:

```
int pthread_attr_setdetachstate(pthread_attr_t *attr,
    int detachstate);
int pthread_attr_getdetachstate(
    const pthread_attr_t *attr,
    int *detachstate);
```

Атрибут может принимать значения:

| | |
|---|---|
| PTHREAD_CREATE_JOINABLE | нить создавать присоединяемой |
| PTHREAD_CREATE_DETACHED | нить создавать отсоединенной |

Привидем здесь шаблон программы, создающей отсоединенный поток:

```
#include <pthread.h>

void* thread_func(void *arg)
{
    /* Выполняемый код новой нити ... */
}

int main()
{   pthread_t       tid;
    pthread_attr_t  attr;

    pthread_attr_init(&attr);
    pthread_attr_setdetachstate(&attr,
        PTHREAD_CREATE_DETACHED);
    pthread_create(&tid, &attr, &thread_func, NULL);
    pthread_attr_destroy(&attr);

    /* Тело главной программы ... */

    return 0;
```

```
}
```

Следующий атрибут определяет политику планирования нитей:

```
int pthread_attr_setschedpolicy(pthread_attr_t *attr,
    int policy);
int pthread_attr_getschedpolicy(
    const pthread_attr_t *attr,
    int *policy);
```

Возможные значения атрибута есть:

| | |
|---|---|
| SCHED_OTHER | обычный планировщик, не реального времени |
| SCHED_RR | карусельный планировщик реального времени |
| SCHED_FIFO | FIFO-планировщик реального времени |

Также предусмотрен атрибут, содержащий параметры планирования:

```
int pthread_attr_setschedparam(pthread_attr_t *attr,
    const struct sched_param *param);
int pthread_attr_getschedparam(
    const pthread_attr_t *attr,
    struct sched_param *param);
```

Параметры планирования являются системно-зависимым атрибутом. Использование его определяется реализацией и для переносимых программ не рекомендуется.

Следующий атрибут управляет наследованием атрибутов нитей:

```
int pthread_attr_setinheritsched(pthread_attr_t *attr,
    int inherit);
int pthread_attr_getinheritsched(
    const pthread_attr_t *attr,
    int *inherit);
```

Его возможные значения:

| | |
|---|---|
| PTHREAD_EXPLICIT_SCHED | планировщик явно управляется политикой и параметрами |
| PTHREAD_INHERIT_SCHED | планировщик наследует параметры у родительской нити |

Область обзора планировшика определяется атрибутом:

```
int pthread_attr_setscope(pthread_attr_t *attr,
```

```
    int scope);
```

Возможные значения атрибута:

| | |
|---|---|
| PTHREAD_SCOPE_SYSTEM | планировщик учитывает нити всех процессов в системе |
| PTHREAD_SCOPE_PROCESS | планировщик планирует нити только одного текущего процесса |

Установка атрибута может игнорироваться реализацией.

Атрибут управляет размером стека нити в байтах:

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

Атрибут управляет расположением стека нити в памяти. Необязателен, то есть может игнорироваться реализацией.

```
int pthread_attr_getstackaddr(
    const pthread_attr_t *attr,
    void **stackaddr);
int pthread_attr_setstackaddr(pthread_attr_t *attr,
    void *stackaddr);
```

## 3 Примеры

Рассмотрим простой пример. В нем создаются две дополнительные нити.

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

void* print_chars(void* ch)
{   char c = *(char*) ch;

    while(1)
    {
       fputc(c, stdout);
    }
```

```
    return NULL;
}

int main()
{   pthread_t thread1_id, thread2_id;
    char c1 = 'x', c2 = 'v';

    pthread_create(&thread1_id,
       NULL, &print_chars, (void*)&c1);
    pthread_create(&thread2_id,
       NULL, &print_chars, (void*)&c2);

    while(1)
    {
       fputc('m', stdout);
    }

    pthread_join(thread1_id, NULL);
    pthread_join(thread2_id, NULL);

    return 0;
}
```

Следующий, более сложный пример, выполняет численное интегрирование, используя модель вычислений «master-slave».

```
#include <pthread.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#define NUMTHR10

int numthr = NUMTHR;
int *offsets;
long n = 100000;
double a = 0.0, b = 1.0, h, *res;

external double f(double);

void *
worker(void *arg)
{
    int offset;
    double x, s;
```

```
    offset = *((int*)arg);
    s = 0.0;
    for (x = a + offset * h; x < b; x += numthr * h)
        s += 0.5 * (f(x) + f(x + h));
    res[offset] = s;
    pthread_exit(NULL);
}

int
main(int argc, char *argv[])
{
  double S = 0.0;
  pthread_t *threads;
  int i, rc;
  threads = (pthread_t*)malloc(NUMTHR * sizeof(phtread_t));
  offsets = (int*)malloc(NUMTHR * sizeof(int));
  res = (double*)malloc(NUMTHR * sizeof(double));
  h = (b - a) / n;
  for (i = 0; i < NUMTHR; ++i)
  {
    offsets[i] = i;
    rc = pthread_create(threads + i, NULL, worker, offsets + i);
    if (rc != 0)
    {
      fprintf(stderr, "pthread_create: error code %d: %s\n",
          rc, strerror(rc));
      exit(-1);
    }
  }

  for (i = 0; i < NUMTHR; ++i)
  {
    rc = pthread_join(threads[i], NULL);
    if (rc != 0)
    {
      fprintf(stderr, "pthread_join: error code %d: %s\n",
          rc, strerror(rc));
      exit(-1);
    }
  }

  for (i = 0; i < NUMTHR; ++i)
      S += res[i];
  print("result = %lf\n", S * h);
}
```

# 4 Синхронизация

## 4.1 Мьютексы

*Мьютексом* (от **MUTual EXclusion** — взаимное исключение) называется специальный примитив синхронизации, которую в конкретный момент времени может устанавливать только одна нить. Если мьютекс захвачен одной нитью, то любая нить обратившаяся к мьютексу будет заблокирована. По освобождению мьютекса владельцем (и только владельцем) одна заблокированная нить пробуждается (и становится новым владельцем мьютекса). Освобождение мьютекса не владельцем игнорируется. То есть мьютекс, в сущности, является специальной разновидностью булевского семафора.

Чтобы избежать типичных для семафоров ошибок программирования, были введены ещё две разновидности мьютексов:

- проверяющий (error check) мьютекс обнаруживает повторный захват мьютекса владельцем, и не выполняя других действий возвращает код ошибки.
- рекурсивный мьютекс считает захваты и освобождения мьютекса владельцем и осуществляет освобождение только при обращении счётчика в ноль.

Для создания мьютекса определяется соотвествующая переменная типа pthread_mutex_t и для нее вызывается инициализатор:

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

Вторым аргументом является набор атрибутов мьютекса.

При успешном выполнении возвращается ноль, в противном случае возвращается код ошибки:

EAGAIN нехватка системных ресурсов (не памяти) для инициализации мьютекса;

ENOMEM нехватка памяти для инициализации мьютекса;

EPERM не хватает привилегий для выполнения операции;

EBUSY попытка повторной инициализации мьютекса без его предварительного разрушения;

EINVAL неправильно сформированы атрибуты мьютекса `mutexattr`.

При успешной инициализации мьютекс становится инициализированным и незаблокированным.

Для создания глобальных (доступных в разных функциях и/или файлах) мьютексов необходимо использовать статические инициализаторы:

```
pthread_mutex_t
    fastmutex = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t
    recmutex = PTHREAD_RECURSIVE_MUTEX_INITIALIZER;
pthread_mutex_t
    errchkmutex = PTHREAD_ERRORCHECK_MUTEX_INITIALIZER;
```

Если мьютекс больше не нужен, его можно уничтожить:

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

При успешном выполнении функция возвращает ноль, в противном случае код ошибки:

EBYSY попытка уничтожить захваченный (заблокированный) мьютекс, или мьютекс который используется условной переменной.

EINVAL некорректное (скорее всего неинициализированное) значение мьютекса.

После уничтожения мьютекса его переменную можно повторно инициализировать, получая новый инициализированный и незаблокированный мьютекс.

Следующая функция захватывает мьютекс, если он никому не принадлежит.

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

Во многих случаях оказывается удобней неблокирующий захват:

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

Если мьютекс ничейный, эта функция захватывает его. Если мьютекс уже захвачен, функция не блокирует вызвавшую её нить, а возвращает код ошибки.

Ещё одна удобная функция. Она пытается захватить мьютекс до наступления момента времени abs_timeout. Если это не удается, возвращатеся ошибка `ETIMEDOUT`.

```
int pthread_mutex_timedlock(pthread_mutex_t *mutex,
    const struct timespec *abs_timeout);
```

И наконец, функция освобождающая мьютекс:

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

Дополняет API набор функций настраивающих атрибуты мьютекса. В него входят создание атрибутов мьютекса:

```
int pthread_mutexattr_init(pthread_mutexattr_t *attr);
```

Функция уничтожающая атрибуты мьютекса

```
int pthread_mutexattr_destroy(pthread_mutexattr_t *attr);
```

Рассмотрим наиболее часто используемые атрибут, определяющие тип мьютекса.

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

Параметр может принимать следующие значения:

| | |
|---|---|
| PTHREAD_MUTEX_NORMAL | Обычный (быстрый) мьютекс |
| PTHREAD_MUTEX_RECURSIVE | Рекурсивный мьютекс |
| PTHREAD_MUTEX_ERRORCHECK | Мьютекс с проверками |

По «историческим» причинам есть ещё один атрибут, делающий в точности то же самое:

```
int pthread_mutexattr_setkind_np(
    pthread_mutexattr_t *attr,
    int type);
int pthread_mutexattr_getkind_np(
    const pthread_mutexattr_t *attr,
    int *type);
```

По тем же «историческим» причинам изменены и имена констант:

PTHREAD_MUTEX_FAST_NP
PTHREAD_MUTEX_RECURSIVE_NP
PTHREAD_MUTEX_ERRORCHECK_NP

Для определения, является ли мьютекс разделяемым, используется следующий атрибут. Реализация может игнорировать изменения атрибута.

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

Возможные значения атрибута:

| | |
|---|---|
| PTHREAD_PROCESS_PRIVATE | мьютекс используется только для нитей одного процесса |
| PTHREAD_PROCESS_SHARED | мьютекс может использоваться для синхронизации нитей разных процессов |

### 4.2 Условные переменные

Для синхронизации нитей используют и *условные (сигнальные) переменные*. Фактически это сигналы, для которых временное освобождения мьютекса и ожидание сигнала объединены в одну атомарную операцию. По получению сигнала производится новый захват мьютекса. Такое решение существенно снижает вероятность ошибок при совместном использовании мьютексов и сигналов.

Для создания условной переменной заводим переменную, которую инициализируем либо вызовом функции-инициализатора:

```
int pthread_cond_init(pthread_cond_t *cond,
    pthread_condattr_t *cond_attr);
```

либо статическим инициализатором:

```
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
```

Ненужная условная переменная уничтожается вызовом:

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

Следующие функции атомарно освобождают мьютекс и ждут получения сигнала от условной переменной. После получения сигнала, они вновь (но уже не атомарно) захватывают мьютекс. Вторая функция ждет сигнала ограниченное время, лишь до момента времени abstime, после чего перестает ждать сигнала и вновь захватывает мьютекс.

```
int pthread_cond_wait(pthread_cond_t *cond,
    pthread_mutex_t *mutex);
int pthread_cond_timedwait(pthread_cond_t *cond,
    pthread_mutex_t *mutex,
    const struct timespec *abstime);
```

У нас есть две функции предназначенные для посылки сигналов. Первая функция пробуждает одну из нитей, ждущих на данной условной переменной, вторая функция пробуждает все нити, ждущие на данной условной переменной. Если ждущих нитей нет, обе функции ничего не делают[1].

```
int pthread_cond_signal(pthread_cond_t *cond);
int pthread_cond_broadcast(pthread_cond_t *cond);
```

Управление атрибутами для условных переменных ничем не отличается от других объектов Pthreads.

```
int pthread_condattr_init(pthread_condattr_t *attr);
int pthread_condattr_destroy(pthread_condattr_t *attr);
int pthread_condattr_getpshared(
    const pthread_condattr_t *attr,
```

1 Именно этим отличаются сигналы от семафоров. Семафоры запоминают посланные сигналы.

```
    int *pshared);
int pthread_condattr_setpshared(
    pthread_condattr_t *attr,
    int pshared);
```

### 4.3 Блокировка чтения/записи

Часто бывают нужны блокровки на чтение/запись разделяемой информации. Обычно для этого используют комбинацию двух семафоров или мьютексов и довольно сложную логику их взаимодействия. Для снижения вероятности ошибок в Pthread в 2001 году был предложен необязательный API блокировок чтения/записи. Обратите внимание, что использование блокировок чтения/записи может вызвать инверсию приоритетов[2].

Поведение такой блокировки основано на следующих правилах:

- любое количество нитей может получить блокировку на чтение, если ни одна нить не получила блокировку на запись;
- блокировка на запись может быть получена, если ни одна нить не получила блокировку на чтение или на запись.

Иными словами, читать можно, если никто не изменяет данные. Изменять данные можно, если никто больше не обращается (не важно, на чтение или на запись) к этим данным.

Для использования такое блокировки создаем и инициализируем соотвествующую переменную, с помощью функции:

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

или статического инициализатора:

```
pthread_rwlock_t rwlock = PTHREAD_RWLOCK_INITIALIZER;
```

Как и для всех объектов, присутствует деструктор:

```
int pthread_rwlock_destroy(pthread_rwlock_t *rwlock);
```

2 То есть явление, когда увеличение приоритета нити приводит к уменьшению достающегося ей времени процессора.

Следующие три функции отвечают за блокировку на чтение, соответственно обычную блокировку, временный запрос блокировки и неблокирующую блокировку.

```
int pthread_rwlock_rdlock(pthread_rwlock_t *rwlock);
int pthread_rwlock_timedrdlock(pthread_rwlock_t *rwlock,
    const struct timespec *abs_timeout);
int pthread_rwlock_tryrdlock(pthread_rwlock_t *rwlock);
```

Аналогично за блокировку на запись отвечают следующие функции:

```
int pthread_rwlock_wrlock(pthread_rwlock_t *rwlock);
int pthread_rwlock_timedwrlock(pthread_rwlock_t *rwlock,
    const struct timespec *abs_timeout);
int pthread_rwlock_trywrlock(pthread_rwlock_t *rwlock);
```

И функция отвечающая за снятие блокировки:

```
int pthread_rwlock_unlock(pthread_rwlock_t *rwlock);
```

### 4.4 Спин-блокировка

С точки зрения прикладного программиста это просто ещё один мьютекс. Нет ни одной разумной причины использовать их на однопроцессорной машине[3].

Областью применения этого примитива является поддержка тонкого (fine-grained) параллелизма на многопроцессорных системах, в предположении, что скорость счета для нас существенно важнее, чем полная загрузка аппаратуры, экономия энергии, и, тем более, чем справедливое распределение ресурсов между процессами.

Инициализация спин-блокировки осуществляется следующей функцией.

```
int pthread_spin_init(pthread_spinlock_t *lock,
    int pshared);
```

Значением параметра pshared должна быть одна из констант:

PTHREAD_PROCESS_PRIVATE синхронизирует только нити одного про-

3 На машине с одним процессором, поддерживающим технологии SMT/HyperThreading®, использование спин-блокировок может быть опасным, приводя практически к зависанию системы.

| | |
|---|---|
| | цесса. Допустима всегда. |
| PTHREAD_PROCESS_SHARED | может синхронизировать нити разных процессов. Допустимо в зависимости от реализации. |

Для инициализации можно также использовать статический инициализатор:

```
pthread_spinlock_t lock = PTHREAD_SPINLOCK_INITIALIZER;
```

Освобождение ресурсов обеспечивает функция:

```
int pthread_spin_destroy(pthread_spinlock_t *lock);
```

Семантика функций установки/снятия блокировки полностью совпадает с семантикой аналогичных функций для мьютексов:

```
int pthread_spin_lock(pthread_spinlock_t *lock);

int pthread_spin_trylock(pthread_spinlock_t *lock);

int pthread_spin_unlock(pthread_spinlock_t *lock);
```

Отметим, что изучение литературы, совпадающее с нашим небольшим опытом использования спин-блокировки в Pthread, позволяет сделать вывод, что спин-блокировка является средством тонкой настройки приложения на конкретную вычислительную установку.

Изменение числа процессоров, скорости процессоров, количества кэш-памяти, скорости оперативной памяти может потребовать новой настройки приложения с выбором между обычными (быстрыми) мьютексами и спин-блокировкой в каждом случае их использования. Целенаправленный отбор вариантов предполагает глубокое понимание архитектуры и принципов функционирования компонентов современных компьютеров и, безусловно, определенной инженерной интуиции.

### 4.4 Барьер

Барьер обеспечивает синхронизацию группы нитей. Достигшие барьера нити ждут, пока остальные нити группы не достигнут барьера. Когда последняя нить из группы достигает барьера, она не переходит к ожиданию, а остальные нити группы пробуждаются. Отметим, что при использовании барьеров может возникать инверсия приоритетов.

Барьеры естественно возникают при попытках распараллеливания циклов[4] и гнёзд циклов, что делает их, возможно, важнейшим видом синхронизации в счётных задачах. Поддержка барьеров со стороны библиотеки и операционной системы тем более важна, что реализация их через другие высокоуровневые примитивы синхронизации обычно отличается неэффективностью.

В Pthreads барьеры были включены в 2001 году как необязательный API.

Создавая барьер мы при его инициализации обязаны указать количество нитей count в группе:

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

Наличие обязательного параметра делает невозможным статический инициализатор для барьера.

Ненужный более барьер может быть уничтожен:

```
int pthread_barrier_destroy(pthread_barrier_t *barrier);
```

Нити синхронизируются на барьере вызывая функцию:

```
int pthread_barrier_wait(pthread_barrier_t *barrier);
```

Нити, вызвавшие эту функцию, приостанавливаются и ждут, пока эта функция не будет вызвана count~раз. Когда будет сделан вызов с номером count, все вызывавшие нити пробуждаются, а сам барьер переходит в состояние, кото-

4 Что позволяет игнорировать инверсию приоритетов, так как в этом случае приоритеты всех нитей полагают равными.

рое имел после последнего для него по времени вызова функции pthread_barrier_init(). Таким образом один барьер может использоваться многократно для групп нитей одинаковой численности.

Работа с атрибутами барьера ничем не отличается от атрибутов других объектов, и прототипы соответствующих функций приводятся здесь для полноты.

```
int pthread_barrierattr_init(pthread_barrierattr_t *attr);

int pthread_barrierattr_destroy(pthread_barrierattr_t *attr);

int pthread_barrierattr_getpshared(
    const pthread_barrierattr_t * attr,
    int *pshared);
int pthread_barrierattr_setpshared(
    pthread_barrierattr_t *attr,
    int pshared);
```

## 5 Частные данные нити

Модель вычислений Pthreads считает всю память процесса разделяемой между всеми нитями этого процесса. Хотя нить (точнее, функция нити) видит далеко не все переменные, и автоматические локальные переменные функции нити в каком-то смысле можно считать собственной памятью нити, во многих ситуациях требуется память, к которой гарантировано имеет доступ только одна нить, а другая нить, обращаясь к тем же переменным получает другие, свои собственные переменные.

Такая память называется *локальной памятью нити* (TLS—Thread Local Storage или TSS—Thread Specific Storage).

Создание такой памяти требует в Pthreads начинается с регистрации ключа.

Для этого создается глобальная переменная типа `pthread_key_t`

которая регистрируется как ключ с помощью следующей функции:

```
int pthread_key_create(pthread_key_t *key,
    void   (*destructor)(void *));
```

Параметр destructor является ссылкой на функцию, которая может корректно освободить память занятую под локальную память нити. Например, если в качестве локальной памяти нити используется блок памяти выделенный с помощью `malloc()`, в качестве параметра `destructor` можно просто передать указатель на функцию `free()`.

Более ненужный ключ можно удалить:

```
int pthread_key_delete(pthread_key_t key);
```

Однако ключи лучше вообще не удалять. Регистрация ключа все равно будет удалена по завершению процесса.

По своей природе ключи — это глобальные переменные, существующие все ремя жизни процесса. Удаление ключа допустимо, когда точно никто и никогда к нему больше не обратиться, поскольку несвоевременное удаление ключа небезопасно для приложения, а в некоторых ситуациях и для операционной системы.

Зафиксировать значение локальной памяти нити:

```
int pthread_setspecific(pthread_key_t key,
    const void *value);
```

Получить текущее значение локальной памяти нити:

```
void* pthread_getspecific(pthread_key_t key);
```

Рассмотрим пример использования локальной памяти нити и однократной инициализации:

```
#include <errno.h>
#include <stdlib.h>

#include <pthread.h>
#include <stdio.h>

/* Создадим обработчики ошибок */
```

```
#define err_abort(code,text)  do { \
   fprintf(stderr, "%s at \"%s\":%d: %s\n",  \
       text, __FILE__, __LINE__, strerror(code)); \
   abort(); \
} while(0);

#define errno_abort(text)  do { \
   fprintf(stderr, "%s at \"%s\":%d: %s\n",  \
       text, __FILE__, __LINE__, strerror(errno)); \
   abort(); \
} while(0);


/* Структура, которую будем помещать в TLS */
typedef struct tld_tag
{   pthread_t  thread_id;
    char       *string;
}   tld_t;

pthread_key_t  tld_key;  /* Thread Local Data Key */
pthread_once_t key_once = PTHREAD_ONCE_INIT;

/* Функция для одноразовой инициализации */
void once_func(void)
{   int status;
    printf("Инициализируем ключ\n");
    status = pthread_key_create(&tld_key, free);
    if (status != 0)
       err_abort(status, "Create key");
}

/* Функция нити */
void* thread_func(void *data)
{   tld_t *value;
    int   status;

    status = pthread_once(&key_once, once_func);
    if (status != 0)
       err_abort(status, "Once init");
    value = (tld_t*) malloc(sizeof(tld_t));
    if (value == NULL)
       errno_abort("Allocate key value");
    status = pthread_setspecific(tld_key, value);
    if (status != 0)
       err_abort(status, "Set tld");
```

```
    printf("%s set tld value %p\n", data, value);
    value->thread_id = pthread_self();
    value->string    = (char*) data;
    value = (tld_t*) pthread_getspecific(tld_key);
    printf("%s starting...", value->string);
    sleep(2);
    value = (tld_t*) pthread_getspecific(tld_key);
    printf("%s done...", value->string);
    return NULL;
}

int main(int argc, char *argv[], char *emvp[])
{   pthread_t thread1, thread2, thread3;
    int       status;
    status = pthread_create(&thread1,
        NULL, thread_func, "Thread 1");
    if (status != 0) err_abort(status, "Create thread 1");
    status = pthread_create(&thread2,
        NULL, thread_func, "Thread 2");
    if (status != 0) err_abort(status, "Create thread 2");
    status = pthread_create(&thread3,
        NULL, thread_func, "Thread 3");
    if (status != 0)2,5
        err_abort(status, "Create thread 3");
    pthread_exit(NULL);
}
```

# Приложение.
# Использование Pthreads-Win32

Пакет Pthreads-Win32 является реализацией спецификации нитей управления из стандарта POSIX 1003c-1995 с дополнениями и исправлениями. В пакет кроме обязательной функциональности нитей включены несколько необязательных средств из стандарта POSIX 1003.1-2001, а также семафоры POSIX и реализации некоторых часто используемых функций POSIX и X/Open OGSU/2, отсутствующих в стандартной библиотеке C.

Предкомпилированный бинарный пакет можно использовать с компиляторами Microsoft VisualC или GNU C (MinGW/MSys). В связи с различием способов формирования фреймов функций и организации обработки исключений в этих компиляторах, установочный пакет содержит несколько версий библиотеки:

Пакет также можно скомпилировать используя другие компиляторы.

Как двоичный пакет, так и его исходные тексты можно получить на его домашней странице `http://sources.redhat.com/pthreads-win32/` или на сервере `ftp://sources.redhat.com/pub/pthreads-win32/` и его зеркалах.

Для использования пакета необходимо получить файлы из каталога `ftp://sources.redhat.com/pub/pthreads-win32/dll-latest/`.

Сначала устанавливается драйвер QueueUserAPCEx. Без него не будут функционировать асинхронные компоненты библиотеки. Для этого файл AlertDrv.sys копируется в `%windir%\system32\drivers`, файл quserex.dll копируется в `%windir%\system32`, и делаем двойной шелчок на файле alertdrv.reg. После перезагрузки драйвер будет установлен в систему и появится новая служба, которую можно запустить командой net start alertdrv, и остановить командой net stop alertdrv.

Таблица 3: Динамические библиотеки в Pthreads-Win32

| Динамическая библиотека | Библиотека импорта | Компилятор и режим |
|---|---|---|
| pthreadGC2.dll | libpthreadGC2.a | GNU C без исключений |
| pthreadGCE2.dll | libpthreadGCE2.a | GNU C++ Exception Handler |
| pthreadVC2.dll | pthreadVC2.lib | MS Visual C без исключений |
| pthreadVCE2.dll | pthreadVCE2.lib | MSVisual C++ Exception Handler |
| pthreadVSE2.dll | pthreadVSE2.lib | MS Visual C со структурными исключениями |

После этого надо скопировать необходимые файлы динамических библиотек в какой-либо каталог в пути поиска, соотвествующие им библиотеки в каталог библиотек, а заголовочные файлы в каталог включаемых файлов компилятора.

Для использования программ, скомпилированных с Pthreads-Win32, нужен драйвер QueueUserAPCEx и динамическая библиотека, соотвествующая используемому компилятору.

Некоторые версии пакета Pthreads-Win32 доступны в виде инсталятора, который сам автоматически поставит все необходимые компоненты в систему.

## Литература


1. Воеводин В. В., Воеводин В. В. Параллельные вычисления. — СПб: БХВ-Петербург, 2002. — 608 с.; ил.
2. ISO/IEC 9945-1:1996 (IEEE/ANSI Std 1003.1 1996 Edition) Information Technology-Portable Operating System Interface (POSIX)-Part 1: System Application: Program Interface (API) [C Language]. — IEEE Standard Press, 1996.
3. Butenhof D. R. Programming with POSIX Threads. — Reading: Addison-Wesley, 1997. — 398 pp.
4. Стивенс У. Р. UNIX: взаимодействие процессов. / Пер. с англ. — СПб: Питер, 2002. — 576 с.; ил.
5. Богачёв К. Ю. Основы параллельного программирования. — М.: БИНОМ. Лаборатория знаний, 2003. — 342 с.; ил.