11

Потоки

11.1. Введение

В предыдущих главах мы обсуждали процессы. Мы рассмотрели окружение процессов в UNIX, взаимоотношения между процессами и способы управления ими.

В этой главе мы продолжим изучение внутреннего устройства процессов и узнаем, как можно использовать несколько *потоков выполнения* (или просто *потоков* (*threads*)) для решения нескольких задач в рамках единственного процесса. Все потоки внутри процесса имеют доступ к одним и тем же компонентам процесса, таким как файловые дескрипторы или переменные.

Всякий раз при попытке организовать одновременный доступ нескольких пользователей к одному и тому же ресурсу приходится сталкиваться с проблемой согласования доступа. В конце этой главы мы рассмотрим механизмы синхронизации потоков, которые позволяют предотвратить доступ разных потоков к разделяемым ресурсам, находящимся в несогласованном состоянии.

11.2. Концепция потоков

Типичный процесс в UNIX можно представить как имеющий единственный поток управления – каждый процесс в один момент времени решает только одну задачу. При использовании нескольких потоков управления можно спроектировать приложение так, что оно будет решать одновременно несколько задач в рамках единственного процесса, где каждый поток решает отдельную задачу. Такой подход имеет следующие преимущества.

- Мы можем значительно упростить код, обрабатывающий асинхронные события, привязав каждый тип события к отдельному потоку. В результате каждый поток может обслуживать свое событие, используя для этого синхронную модель программирования, которая намного проще асинхронной.
- Чтобы организовать совместный доступ нескольких процессов к одним и тем же ресурсам, таким как разделяемая память или файловые дескрипторы, необходимо использовать достаточно сложные механизмы синхро-

низации, предоставляемые операционной системой (об этом – в главах 15 и 17). Потоки же, в отличие от процессов, автоматически получают доступ к одному и тому же адресному пространству и файловым дескрипторам.

- Решение некоторых задач можно разбить на более мелкие подзадачи, что может дать прирост производительности программы. Однопоточный процесс, выполняющий решение нескольких задач, неявно вынужден решать их последовательно, поскольку он имеет только один поток управления. При наличии нескольких потоков управления независимые друг от друга задачи могут решаться одновременно отдельными потоками. Две задачи могут решаться одновременно только при условии, что они не зависят друг от друга.
- Аналогично, интерактивные программы могут сократить время отклика на действия пользователя, используя многопоточную модель, чтобы отделить обработку ввода/вывода пользователя от других частей программы.

У многих многопоточное программирование ассоциируется с многопроцессорными системами. Однако преимущества многопоточной модели проявляют себя, даже если программа работает в однопроцессорной системе. Независимо от количества процессоров программа может быть упрощена благодаря многопоточной модели, поскольку количество процессоров не влияет на структуру программы. Кроме того, в то время как однопоточный процесс вынужден периодически простаивать при последовательном решении нескольких задач, многопоточный процесс может повысить производительность и в однопроцессорной системе, так как часть потоков могут продолжать работу, когда другие приостановлены в ожидании наступления некоторых событий.

Поток содержит набор информации, необходимой для представления контекста выполнения внутри процесса. Сюда включаются *идентификатор потока*, который идентифицирует поток внутри процесса, набор значений в регистрах процессора, стек, приоритет, маска сигналов, переменная `errno` (раздел 1.7) и дополнительные данные, специфичные для потока (раздел 12.6). Все компоненты процесса, включая выполняемый код программы, глобальные переменные и динамическую память, стеки и файловые дескрипторы, могут совместно использоваться различными потоками этого процесса.

Интерфейс потоков, о котором мы будем говорить, определяется стандартом POSIX.1-2001. Этот интерфейс, известный также как «pthreads» (от «POSIX threads»), первоначально представлял собой дополнительную функциональную возможность, включенную в стандарт POSIX.1-2001, но в версии SUSv4 был перемещен в раздел базовых спецификаций. Поддержку потоков POSIX можно проверить с помощью макроопределения `_POSIX_THREADS`. Приложения могут выполнять проверку поддержки потоков во время компиляции, используя команду условной компиляции `#ifdef`, или во время выполнения, вызывая функцию `sysconf` с аргументом `_SC_THREADS`. Системы, соответствующие стандарту SUSv4, определяют символ `_POSIX_THREADS` со значением 200809L.

11.3. Идентификация потоков

Как любой процесс обладает идентификатором процесса, так и каждый поток имеет свой идентификатор потока. В отличие от процессов, идентификаторы которых являются уникальными в пределах системы, идентификатор потока имеет смысл только в контексте процесса, которому он принадлежит.

Мы уже говорили, что идентификатор процесса представлен типом `pid_t` и является целым неотрицательным числом. Идентификатор потока представлен типом `pthread_t`. Реализациям разрешается использовать структуру для представления типа `pthread_t`, поэтому, чтобы сохранить переносимость приложений, мы не должны рассматривать этот тип как целое число. Следовательно, сравнение двух идентификаторов потоков должно выполняться с помощью функции.

```
#include <pthread.h>

int pthread_equal(pthread_t tid1, pthread_t tid2);
```

Возвращает ненулевое значение, если идентификаторы равны, 0 – в противном случае

> Для представления типа `pthread_t` ОС Linux 3.2.0 использует тип `long int`, Solaris 10 – `unsigned int`, а FreeBSD 8.0 и Mac OS X 10.6.8 в качестве типа `pthread_t` используют указатель на структуру `pthread`.

Поскольку тип `pthread_t` может быть структурой, не существует достаточно переносимого способа вывести его значение. Иногда в процессе отладки программы бывает удобно выводить идентификаторы потоков, но, как правило, в других случаях в этом нет необходимости. В самом худшем случае это приводит к написанию непереносимого отладочного кода, поэтому данное ограничение можно считать несущественным.

Поток может получить собственный идентификатор, обратившись к функции `pthread_self`.

```
#include <pthread.h>

pthread_t pthread_self(void);
```

Возвращает идентификатор вызывающего потока

Эта функция может использоваться совместно с `pthread_equal`, если внутри потока возникнет необходимость самоидентификации. Например, главный поток может размещать задания в некоторой очереди и сопровождать их идентификаторами потоков, чтобы каждый поток мог выполнять задания, предназначенные конкретно для него. Эта методика показана на рис. 11.1. Главный поток помещает новые задания в очередь, а три рабочих потока извлекают их из очереди. Вместо того чтобы позволить произвольному потоку извлекать очередное задание из начала очереди, главный поток, используя идентификаторы потоков, назначает задания конкретным потокам. В этом случае рабо-

чий поток извлекает из очереди только те задания, которые отмечены его идентификатором.

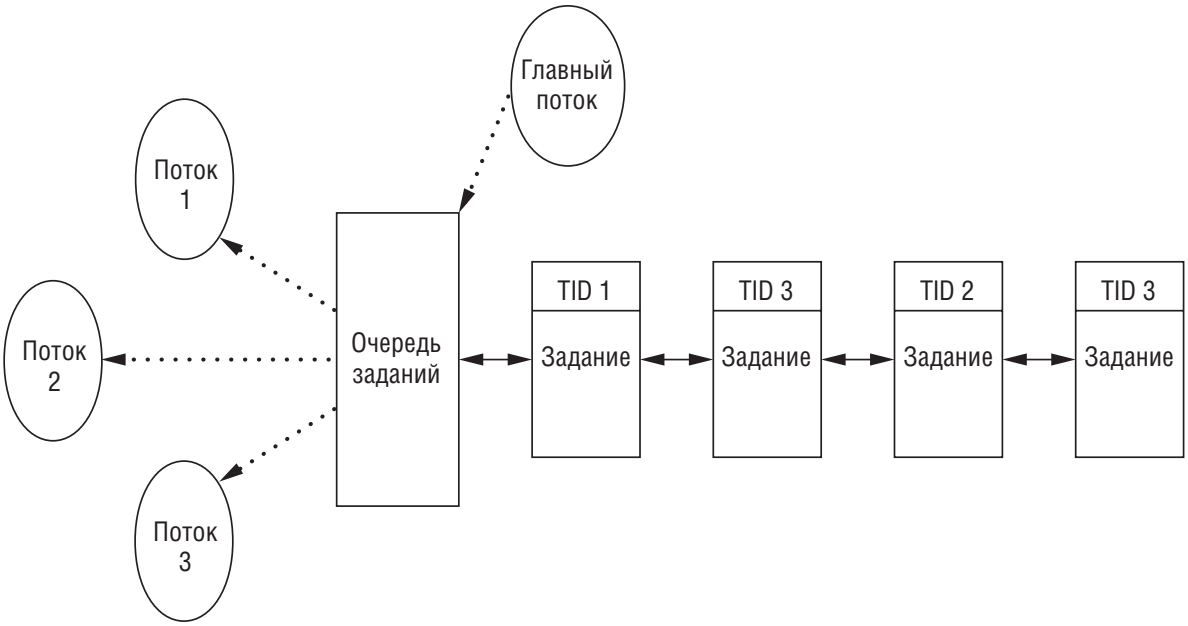

Рис. 11.1. *Пример очереди заданий*

11.4. Создание потока

Традиционная модель процессов в UNIX поддерживает только один поток управления на процесс. Концептуально это то же, что и модель, основанная на потоках, в случае, когда каждый процесс состоит из одного потока. При наличии поддержки pthread программа также запускается как процесс, состоящий из одного потока управления. Поведение такой программы ничем не отличается от поведения традиционного процесса, пока она не создаст дополнительные потоки управления. Создание дополнительных потоков производится с помощью функции `pthread_create`.

```
#include <pthread.h>

int pthread_create(pthread_t *restrict tidp,
                   const pthread_attr_t *restrict attr,
                   void *(*start_rtn)(void *), void *restrict arg);
```

Возвращает 0 в случае успеха, код ошибки – в случае неудачи

Аргумент *tidp* – это указатель на область памяти, где будет размещен идентификатор созданного потока, если вызов функции `pthread_create` завершится успехом. Аргумент *attr* используется для настройки различных атрибутов потока. Об атрибутах потоков мы поговорим в разделе 12.3, а пока будем передавать в этом аргументе пустой указатель (`NULL`), что соответствует созданию потока со значениями атрибутов по умолчанию.

Вновь созданный поток начинает исполнение с функции *start_rtn*. Эта функция принимает единственный аргумент, *arg* – нетипизированный указатель. Если функции *start_rtn* потребуется передать значительный объем информации, ее следует сохранить в виде структуры и передать указатель на структуру в аргументе *arg*.

При создании нового потока нельзя заранее предполагать, кто первым получит управление – вновь созданный поток или поток, вызвавший функцию pthread_create. Новый поток имеет доступ к адресному пространству процесса и наследует от вызывающего потока среду окружения арифметического сопроцессора и маску сигналов, однако набор сигналов, ожидающих обработки, для нового потока очищается.

Обратите внимание, что функции семейства pthread, как правило, возвращают код ошибки в случае неудачи. Они не изменяют значение переменной errno подобно другим функциям POSIX. Экземпляр переменной errno для каждого потока предоставляется только для сохранения совместимости с существующими функциями, которые используют эту переменную. Вообще, при работе с потоками принято возвращать код ошибки из функций, что дает возможность локализовать ошибку, а не полагаться на некоторую глобальную переменную, которая могла быть изменена в результате побочного эффекта.

Пример

Несмотря на отсутствие переносимого способа вывода значений идентификаторов потоков, все же можно написать небольшую программу, которая делает это, и тем самым получить представление о некоторых особенностях потоков. Программа в листинге 11.1, выводит идентификатор процесса и идентификаторы начального и вновь созданного потоков.

Листинг 11.1. Вывод идентификаторов потоков

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

pthread_t ntid;

void
printids(const char *s)
{
    pid_t     pid;
    pthread_t tid;

    pid = getpid();
    tid = pthread_self();
    printf("%s pid %u tid %u (0x%x)\n", s, (unsigned int)pid,
           (unsigned int)tid, (unsigned int)tid);
}

void *
thr_fn(void *arg)
{
    printids("новый поток: ");
```

```
    return((void *)0);
}

int
main(void)
{
    int     err;

    err = pthread_create(&ntid, NULL, thr_fn, NULL);
    if (err != 0)
        err_exit(err, "невозможно создать поток");
    printids("главный поток:");
    sleep(1);
    exit(0);
}
```

В этом примере есть два интересных момента, связанных с возможностью гонки за ресурсами между основным и вновь созданным потоками. (Далее в этой же главе мы рассмотрим более правильные способы синхронизации потоков.) В первую очередь необходимо приостановить основной поток. Если этого не сделать, основной поток может завершиться и тем самым завершить весь процесс еще до того, как новый поток получит возможность начать работу. Такое поведение потоков во многом зависит от реализации потоков в операционной системе и алгоритма планирования.

Второй интересный момент заключается в том, что новый поток получает свой идентификатор с помощью функции `pthread_self`, а не берет его из глобальной переменной или из аргумента запускающей функции. При описании функции `pthread_create` мы уже говорили, что она возвращает идентификатор созданного потока через аргумент *tidp*. В нашем примере основной поток сохраняет его в переменной `ntid`, но новый поток не может ее использовать. Если новый поток получит управление первым, еще до того, как функция `pthread_create` вернет управление в основной поток, вместо идентификатора новый поток обнаружит неинициализированное значение переменной `ntid`.

Запустив программу из листинга 11.1 в ОС Solaris, мы получили следующие результаты:

```
$ ./a.out
главный поток: pid 20075 tid 1 (0x1)
новый поток:   pid 20075 tid 2 (0x2)
```

Как мы и ожидали, оба потока обладают одним и тем же идентификатором процесса, но разными идентификаторами потоков. Запуск программы из листинга 11.1 в ОС FreeBSD дал следующие результаты:

```
$ ./a.out
главный поток: pid 37396 tid 673190208 (0x28201140)
новый поток: pid 37396 tid 673280320 (0x28217140)
```

В этом случае потоки также имеют один и тот же идентификатор процесса. Если рассматривать идентификаторы потоков как целые десятичные числа, они могут показаться достаточно странными, но если их рассматривать в шестнад-

цатеричном представлении, они приобретают некоторый смысл. Как уже отмечалось выше, в качестве идентификатора потока FreeBSD использует указатель на структуру с данными потока.

В Mac OS X можно было бы ожидать похожих результатов, однако идентификаторы главного потока и потока, созданного с помощью функции `pthread_create`, принадлежат к разным диапазонам адресов.

```
$ ./a.out
главный поток: pid 31807 tid 140735073889440 (0x7fff70162ca0)
новый поток: pid 31807 tid 4295716864 (0x1000b7000)
```

Запуск программы в ОС Linux дал несколько иные результаты:

```
$ ./a.out
новый поток: pid 17874 tid 140693894424320 (0x7ff5d9996700)
главный поток: pid 17874 tid 140693886129920 (0x7ff5d91ad700)
```

Идентификаторы потоков в Linux больше напоминают указатели, хотя в действительности являются целыми числами без знака.

> Реализация потоков изменилась при переходе от версии Linux 2.4 к Linux 2.6. В Linux 2.4 подсистема LinuxThreads реализовала потоки как отдельные процессы. Это мешало реализации поведения потоков так, чтобы оно соответствовало требованиям стандарта POSIX. В Linux 2.6 ядро и библиотека поддержки потоков были полностью переделаны под новую реализацию потоков под названием Native POSIX Thread Library (NPTL). Она поддерживает модель выполнения множества потоков в рамках единственного процесса и упрощает поддержку семантики потоков в соответствии со стандартом POSIX.

11.5. Завершение потока

Если один из потоков вызовет функцию `exit`, `_exit` или `_Exit`, будет завершен весь процесс. Аналогично, если потоку будет послан сигнал, действие которого заключается в завершении процесса, этот сигнал завершит весь процесс (более подробно о взаимодействиях между сигналами и потоками мы поговорим в разделе 12.8).

Завершить работу единственного потока, то есть без завершения всего процесса, можно тремя способами.

1. Поток может просто вернуть управление из запускающей процедуры. Возвращаемое значение этой процедуры – код завершения потока.
2. Поток может быть принудительно завершен другим потоком того же самого процесса.
3. Поток может вызвать функцию `pthread_exit`.

```
#include <pthread.h>

void pthread_exit(void *rval_ptr);
```

Аргумент *rval_ptr* – это нетипизированный указатель, аналогичный аргументу, передаваемому запускающей процедуре. Этот указатель смогут получить другие потоки процесса, вызвавшие функцию pthread_join.

```
#include <pthread.h>

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

Возвращает 0 в случае успеха, код ошибки – в случае неудачи

Вызывающий поток будет заблокирован, пока указанный поток не вызовет функцию pthread_exit, не вернет управление из запускающей процедуры или не будет принудительно завершен другим потоком. Если поток просто выйдет из запускающей процедуры, *rval_ptr* будет содержать возвращаемое значение. Если поток был принудительно завершен, по адресу *rval_ptr* будет записано значение PTHREAD_CANCELED.

Вызов функции pthread_join автоматически переводит поток в обособленное состояние (вскоре мы обсудим это), которое позволяет вернуть ресурсы потока обратно. Если он уже находится в обособленном состоянии, поток, вызвавший pthread_join, получит код ошибки EINVAL.

Если нас не интересует возвращаемое значение потока, мы можем передать пустой указатель в аргументе *rval_ptr*. В этом случае обращение к функции pthread_join позволит дождаться завершения указанного потока, но не вернет код его завершения.

Пример

В листинге 11.2 показано, как получить код выхода завершившегося потока.

Листинг 11.2. Получение кода выхода потока

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

void *
thr_fn1(void *arg)
{
    printf("поток 1: выход\n");
    return((void *)1);
}

void *
thr_fn2(void *arg)
{
    printf("поток 2: выход\n");
    pthread_exit((void *)2);
}

int
main(void)
{
```

```
    int         err;
    pthread_t   tid1, tid2;
    void        *tret;

    err = pthread_create(&tid1, NULL, thr_fn1, NULL);
    if (err != 0)
        err_exit(err, "невозможно создать поток 1");
    err = pthread_create(&tid2, NULL, thr_fn2, NULL);
    if (err != 0)
        err_exit(err, "невозможно создать поток 2");
    err = pthread_join(tid1, &tret);
    if (err != 0)
        err_exit(err, "невозможно присоединить поток 1");
    printf("код выхода потока 1: %ld\n", (long)tret);
    err = pthread_join(tid2, &tret);
    if (err != 0)
        err_exit(err, "невозможно присоединить поток 2");
    printf("код выхода потока 2: %ld\n", (long)tret);
    exit(0);
}
```

Запустив программу из листинга 11.2, мы получили:

```
$ ./a.out
поток 1: выход
поток 2: выход
код выхода потока 1: 1
код выхода потока 2: 2
```

Как видите, когда поток завершается вызовом функции `pthread_exit` или просто возвращая управление из запускающей процедуры, другой поток может получить код выхода через вызов функции `pthread_join`.

Нетипизированный указатель, передаваемый функциям `pthread_create` и `pthread_exit`, может использоваться для передачи более одного значения. В этом указателе можно передать адрес структуры, содержащей большой объем информации. Помните, что этот адрес должен оставаться действительным после выхода из вызывающей функции. Если, к примеру, структура размещается на стеке вызывающей функции, ее содержимое может оказаться измененным к моменту, когда она будет использована. Если поток размещает структуру на стеке и передает указатель на нее функции `pthread_exit`, стек этого потока может оказаться разрушенным, а память, занимаемая им, может быть использована повторно для других целей к моменту, когда поток, вызвавший `pthread_join`, попытается обратиться к ней.

Пример

Программа в листинге 11.3 демонстрирует проблему, связанную с использованием переменной с автоматическим классом размещения (на стеке) в качестве аргумента функции `pthread_exit`.

Листинг 11.3. Некорректное использование аргумента функции `pthread_exit`

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

struct foo {
    int a, b, c, d;
};

void
printfoo(const char *s, const struct foo *fp)
{
    printf("%s", s);
    printf(" структура по адресу 0x%lx\n", (unsigned long)fp);
    printf(" foo.a = %d\n", fp->a);
    printf(" foo.b = %d\n", fp->b);
    printf(" foo.c = %d\n", fp->c);
    printf(" foo.d = %d\n", fp->d);
}

void *
thr_fn1(void *arg)
{
    struct foo  foo = {1, 2, 3, 4};

    printfoo("поток 1:\n", &foo);
    pthread_exit((void *)&foo);
}

void *
thr_fn2(void *arg)
{
    printf("поток 2: идентификатор - %lu\n", (unsigned long)pthread_self());
    pthread_exit((void *)0);
}

int
main(void)
{
    int         err;
    pthread_t   tid1, tid2;
    struct foo  *fp;

    err = pthread_create(&tid1, NULL, thr_fn1, NULL);
    if (err != 0)
        err_exit(err, "невозможно создать поток 1");
    err = pthread_join(tid1, (void *)&fp);
    if (err != 0)
        err_exit(err, "невозможно присоединить поток 1");
    sleep(1);
    printf("родительский процесс создает второй поток\n");
    err = pthread_create(&tid2, NULL, thr_fn2, NULL);
    if (err != 0)
        err_exit(err, "невозможно создать поток 2");
    sleep(1);
```

```
    printfoo("родительский процесс:\n", fp);
    exit(0);
}
```

Запустив эту программу в ОС Linux, мы получили:

```
$ ./a.out
поток 1:
  структура по адресу 0x7f2c83682ed0
  foo.a = 1
  foo.b = 2
  foo.c = 3
  foo.d = 4
родительский процесс создает второй поток
поток 2: идентификатор - 139829159933696
родительский процесс:
  структура по адресу 0x7f2c83682ed0
  foo.a = -2090321472
  foo.b = 32556
  foo.c = 1
  foo.d = 0
```

Разумеется, результаты зависят от архитектуры памяти, компилятора и реализации библиотеки функций для работы с потоками. В ОС Solaris были получены похожие результаты:

```
$ ./a.out
поток 1:
  структура по адресу 0xffffffff7f0fbf30
  foo.a = 1
  foo.b = 2
  foo.c = 3
  foo.d = 4
родительский процесс создает второй поток
поток 2: идентификатор - 3
родительский процесс:
  структура по адресу 0xffffffff7f0fbf30
  foo.a = -1
  foo.b = 2136969048
  foo.c = -1
  foo.d = 2138049024
```

Как видите, содержимое структуры (размещенной на стеке потоком *tid1*) изменилось к тому моменту, когда главный поток получил к ней доступ. Обратите внимание, как стек второго потока (*tid2*) наложился на стек первого потока. Чтобы решить эту проблему, можно либо использовать глобальную память, либо размещать структуру с помощью функции `malloc`.

В Mac OS X мы получили иные результаты:

```
$ ./a.out
поток 1:
  структура по адресу 0x1000b6f00
  foo.a = 1
```

```
  foo.b = 2
  foo.c = 3
  foo.d = 4
родительский процесс создает второй поток
поток 2: идентификатор - 4295716864
родительский процесс:
  структура по адресу 0x1000b6f00
Segmentation fault (core dumped)
```

В данном случае, когда родитель попытался получить доступ к структуре, переданной ему при выходе из первого потока, адрес структуры оказался недействительным и родителю был послан сигнал SIGSEGV.

В ОС FreeBSD память не была затерта к моменту, когда родитель обратился к ней, и мы получили:

```
поток 1:
  структура по адресу 0xbf9fef88
  foo.a = 1
  foo.b = 2
  foo.c = 3
  foo.d = 4
родительский процесс создает второй поток
поток 2: идентификатор - 673279680
родительский процесс:
  структура по адресу 0xbf9fef88
  foo.a = 1
  foo.b = 2
  foo.c = 3
  foo.d = 4
```

Но, даже при том что после выхода из потока память оказалась не повреждена, мы не должны полагать, что так будет всегда. Кроме того, на других платформах мы наблюдаем как раз противоположное поведение.

Один поток может передать запрос на принудительное завершение другого потока того же самого процесса, обратившись к функции pthread_cancel.

```
#include <pthread.h>

int pthread_cancel(pthread_t tid);
```

Возвращает 0 в случае успеха, код ошибки – в случае неудачи

По умолчанию вызов функции pthread_cancel заставляет указанный поток вести себя, как будто бы он вызвал функцию pthread_exit с аргументом PTHREAD_CANCELED. Однако поток может отвергнуть запрос или как-то иначе отреагировать на него. Более подробно мы обсудим эту тему в разделе 12.7. Обратите внимание, что функция pthread_cancel не ждет завершения потока. Она просто посылает запрос.

Поток может назначить некоторую функцию для вызова в момент его завершения примерно так же, как это делается для процессов с помощью функции

atexit (раздел 7.3), которая регистрирует функции, запускаемые при завершении процесса. Эти функции называют *функциями обработки выхода из потока*. Поток может зарегистрировать несколько таких функций обработки выхода. Обработчики заносятся в стек – это означает, что они будут вызываться в порядке, обратном порядку их регистрации.

```
#include <pthread.h>

void pthread_cleanup_push(void (*rtn)(void *), void *arg);

void pthread_cleanup_pop(int execute);
```

Функция pthread_cleanup_push регистрирует функцию *rtn*, которая будет вызвана с аргументом *arg*, когда поток выполнит одно из следующих действий:

- Вызовет функцию pthread_exit
- Ответит на запрос о принудительном завершении
- Вызовет функцию pthread_cleanup_pop с ненулевым аргументом *execute*

Если аргумент *execute* имеет значение 0, функция обработки выхода из потока вызываться не будет. В любом случае функция pthread_cleanup_pop удаляет функцию-обработчик, зарегистрированную последним обращением к функции pthread_cleanup_push.

Ограничение, связанное с этими функциями, заключается в том, что они могут быть реализованы в виде макроопределений и тогда они должны использоваться в паре, в пределах одной и той же области видимости в потоке. Макроопределение функции pthread_cleanup_push может включать в себя символ {, и тогда парная ей скобка } будет находиться в макроопределении pthread_cleanup_pop.

Пример

В листинге 11.4 показан порядок использования функций обработки выхода из потока. Хотя это достаточно искусственный пример, тем не менее он прозрачно иллюстрирует описываемую методику. Обратите внимание: хотя ненулевой аргумент и не передается в функцию pthread_cleanup_pop, тем не менее мы по-прежнему вынуждены вызывать функции pthread_cleanup_push и pthread_cleanup_pop в паре, в противном случае программа может не скомпилироваться.

Листинг 11.4. Обработчик выхода из потока

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

void
cleanup(void *arg)
{
    printf("выход: %s\n", (char *)arg);
}

void *
thr_fn1(void *arg)
{
```

```
    printf("запуск потока 1\n");
    pthread_cleanup_push(cleanup, "поток 1, первый обработчик");
    pthread_cleanup_push(cleanup, "поток 1, второй обработчик");
    printf("поток 1, регистрация обработчиков закончена\n");
    if (arg)
        return((void *)1);
    pthread_cleanup_pop(0);
    pthread_cleanup_pop(0);
    return((void *)1);
}

void *
thr_fn2(void *arg)
{
    printf("запуск потока 2\n");
    pthread_cleanup_push(cleanup, "поток 2, первый обработчик");
    pthread_cleanup_push(cleanup, "поток 2, второй обработчик");
    printf("поток 1, регистрация обработчиков закончена\n");
    if (arg)
        pthread_exit((void *)2);
    pthread_cleanup_pop(0);
    pthread_cleanup_pop(0);
    pthread_exit((void *)2);
}

int
main(void)
{
    int         err;
    pthread_t   tid1, tid2;
    void        *tret;

    err = pthread_create(&tid1, NULL, thr_fn1, (void *)1);
    if (err != 0)
        err_exit(err, "невозможно создать поток 1");
    err = pthread_create(&tid2, NULL, thr_fn2, (void *)1);
    if (err != 0)
        err_exit(err, "невозможно создать поток 2");
    err = pthread_join(tid1, &tret);
    if (err != 0)
        err_exit(err, "невозможно присоединить поток 1");
    printf("код выхода потока 1: %ld\n", (long)tret);
    err = pthread_join(tid2, &tret);
    if (err != 0)
        err_exit(err, "невозможно присоединить поток 2");
    printf("код выхода потока 2: %ld\n", (long)tret);
    exit(0);
}
```

Запуск программы из листинга 11.4 дал нам следующие результаты:

```
$ ./a.out
запуск потока 1
поток 1, регистрация обработчиков закончена
```

```
запуск потока 2
поток 2, регистрация обработчиков закончена
выход: поток 2, второй обработчик
выход: поток 2, первый обработчик
код выхода потока 1: 1
код выхода потока 2: 2
```

Из полученных результатов видно, что оба потока нормально запустились и корректно завершились, но функции обработки выхода были вызваны только для второго потока. Таким образом, можно сделать вывод, что функции обработки выхода из потока не вызываются, если поток завершается простым возвратом из его процедуры запуска. Кроме того, обратите внимание, что функции обработки выхода запускаются в порядке, обратном порядку их регистрации.

Если запустить ту же программу в ОС FreeBSD или Mac OS X, программа завершится аварийно с сохранением файла core. Проблема в том, что в этих системах функция `pthread_cleanup_push` реализована как макрос, сохраняющий некоторые контекстные данные на стеке. Когда поток 1 возвращает управление между вызовами `pthread_cleanup_push` и `pthread_cleanup_pop`, содержимое стека затирается и программа в этих системах использует уже поврежденный к этому моменту контекст, когда пытается вызвать обработчики выхода из потока. Поведение системы при возврате между соответствующими парными вызовами `pthread_cleanup_push` и `pthread_cleanup_pop` не регламентируется стандартом Single UNIX Specification. Единственный переносимый способ выполнить возврат между вызовами этих двух функций – обратиться к функции `pthread_exit`.

Сейчас вы уже должны обнаружить некоторые черты сходства между функциями управления процессами и функциями управления потоками. В табл. 11.1 приводится список аналогичных функций.

Таблица 11.1. Функции управления процессами и потоками

Процессы	Потоки	Описание
`fork`	`pthread_create`	Создает новый поток управления
`exit`	`pthread_exit`	Завершает существующий поток управления
`waitpid`	`pthread_join`	Возвращает код выхода из потока управления
`atexit`	`pthread_cleanup_push`	Регистрирует функцию обработки выхода из потока управления
`getpid`	`pthread_self`	Возвращает идентификатор потока управления
`abort`	`pthread_cancel`	Запрашивает аварийное завершение потока управления

По умолчанию код завершения потока сохраняется, пока для этого потока не будет вызвана функция `pthread_join`. Основная память потока может быть немедленно освобождена по его завершении, если поток был *обособлен*. Когда поток обособлен, функция `pthread_join` не может использоваться для получения его кода завершения, потому что в этом случае ее поведение не определено. Обособить поток можно с помощью функции `pthread_detach`.

```
#include <pthread.h>

int pthread_detach(pthread_t tid);
```

Возвращает 0 в случае успеха, код ошибки – в случае неудачи

Как мы увидим в следующей главе, существует возможность создания потока изначально в обособленном состоянии, через изменение атрибутов потока, передаваемых функции `pthread_create`.

11.6. Синхронизация потоков

При наличии нескольких потоков управления, совместно использующих одни и те же данные, необходимо гарантировать, что все потоки будут видеть стабильное представление этих данных. Если каждый из потоков использует переменные, которые не модифицируются в других потоках, проблем не возникает. Аналогично, если переменная доступна одновременно нескольким потокам только для чтения, здесь также отсутствует проблема сохранения непротиворечивости. Однако, если один поток изменяет значение переменной, читать или изменять которое могут также другие потоки, необходимо синхронизировать доступ к переменной, чтобы гарантировать, что потоки не будут получать неверное значение переменной при одновременном доступе к ней.

Когда поток изменяет значение переменной, существует потенциальная опасность, что другой поток может прочитать еще не до конца записанное значение. На аппаратных платформах, где запись в память осуществляется более чем за один цикл, может произойти так, что между двумя циклами записи вклинится цикл чтения. Разумеется, такое поведение во многом зависит от аппаратной архитектуры, но при написании переносимых программ мы не можем полагаться на то, что они будут выполняться только на определенной платформе.

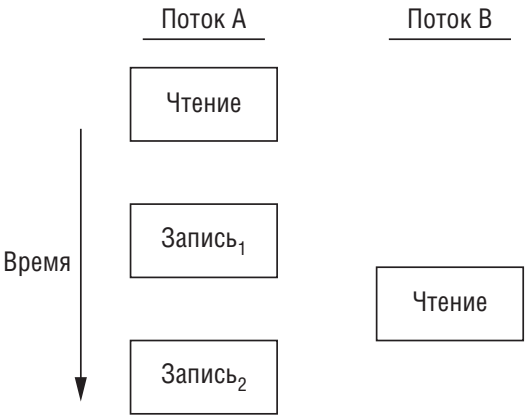

Рис. 11.2. *Перемежение циклов доступа к памяти из двух потоков*

На рис. 11.2 приводится пример гипотетической ситуации, когда два потока одновременно выполняют запись и чтение значения одной и той же переменной. В данном примере поток A считывает значение переменной и затем запи-

сывает в нее новое значение, но операция записи производится за два цикла. Если поток B прочитает значение этой же переменной между двумя циклами записи, он обнаружит переменную в нестабильном состоянии.

Для решения этой проблемы потоки должны использовать блокировки, которые позволят только одному потоку работать с переменной в один момент времени. На рис. 11.3 показана подобная синхронизация. Если поток B должен прочитать значение переменной, он устанавливает блокировку. Аналогично, когда поток A изменяет значение переменной, он также устанавливает блокировку. Благодаря такой организации, поток B не сможет прочитать значение переменной, пока поток A не снимет блокировку.

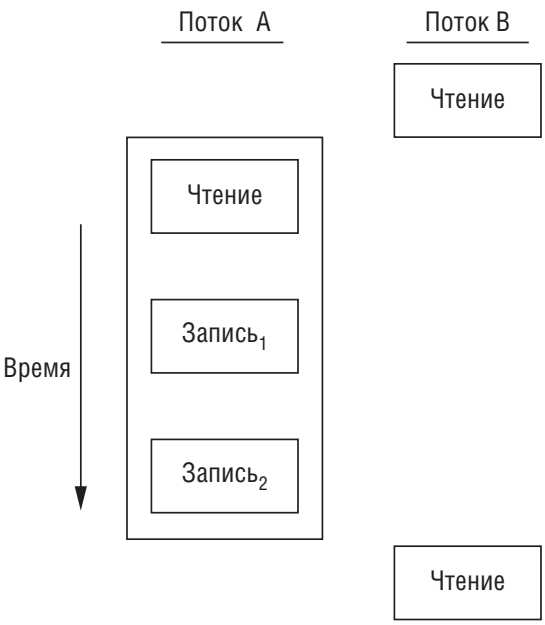

Рис. 11.3. *Синхронизированный доступ к памяти из двух потоков*

Точно так же следует синхронизировать два или более потоков, которые могут попытаться одновременно изменить значение переменной. Рассмотрим случай, когда выполняется увеличение значения переменной на 1 (рис. 11.4). Операцию увеличения (инкремента) обычно можно разбить на три шага.

1. Прочитать значение переменной из памяти в регистр процессора.
2. Увеличить значение в регистре.
3. Записать новое значение из регистра процессора в память.

Если два потока попытаются одновременно увеличить значение одной и той же переменной, не согласуя свои действия между собой, то результаты могут быть получены самые разные. В конечном итоге полученное значение может оказаться на 1 или на 2 больше предыдущего в зависимости от того, какое значение получил второй поток перед началом операции. Если второй поток выполнил шаг 1 до того, как первый выполнил шаг 3, второй поток прочитает то же самое значение, что и первый поток, увеличит его на 1 и запишет обратно в память, фактически не оказав никакого влияния на значение переменной.

Рис. 11.4. *Два несинхронизированных потока пытаются увеличить значение одной и той же переменной*

Если изменение переменной производится атомарно, подобная гонка между потоками отсутствует. В предыдущем примере, если увеличение производится за одно обращение к памяти, состояние гонки между потоками не возникает. Если данные *постоянно находятся в непротиворечивом состоянии*, нет необходимости предусматривать дополнительную синхронизацию. Операции являются последовательно непротиворечивыми, если различные потоки не могут получить доступ к данным, когда они находятся в противоречивом состоянии. В современных компьютерных системах доступ к памяти выполняется за несколько тактов шины, а в многопроцессорных системах доступ к шине вообще чередуется между несколькими процессорами, поэтому невозможно гарантировать непротиворечивое состояние данных в любой произвольный момент времени.

В непротиворечивой среде изменения данных можно описать как последовательность операций, выполняемых потоками. Мы можем сказать: «Поток A увеличил значение переменной, затем поток B увеличил значение переменной, в результате значение переменной было увеличено на 2» или: «Поток B увеличил значение переменной, затем поток A увеличил значение переменной, в результате значение переменной было увеличено на 2». Конечный результат не зависит от того или иного порядка выполнения потоков.

Помимо особенностей аппаратной архитектуры, состояние гонки может быть вызвано алгоритмом использования переменных в программах. Например,

мы можем увеличить значение переменной и затем, основываясь на полученном значении, принять решение о дальнейшем порядке выполнения операций. Комбинация операций, состоящая из увеличения переменной и проверки полученного значения, не является атомарной, и таким образом появляется вероятность принятия неверного решения.

11.6.1. Мьютексы

Мы можем защитить данные и ограничить доступ к ним одним потоком в один момент времени с помощью взаимоисключений (mutual-exclusion) интерфейса pthreads. *Мьютекс* (*mutex*) – это фактически блокировка, которая устанавливается (запирается) перед обращением к разделяемому ресурсу и снимается (отпирается) после выполнения требуемой последовательности операций. Если мьютекс заперт, любой другой поток, который попытается запереть его, будет заблокирован, пока мьютекс не будет отперт. Если в момент отпирания мьютекса сразу несколько потоков будут находиться в заблокированном состоянии, все они будут запущены, и первый, кто успеет запереть мьютекс, продолжит работу. Все остальные потоки обнаружат запертый мьютекс и опять перейдут в режим ожидания. Таким образом, доступ к ресурсу сможет получить одновременно только один поток.

Такой механизм взаимоисключений будет корректно работать только при условии, что все потоки приложения будут соблюдать одни и те же правила доступа к данным. Операционная система никак не упорядочивает доступ к данным. Если мы позволим одному потоку производить действия с разделяемыми данными, предварительно не ограничив доступ к ним, остальные потоки могут обнаружить эти данные в противоречивом состоянии, даже если перед обращением к ним будут устанавливать блокировку.

Переменные-мьютексы определяются с типом `pthread_mutex_t`. Прежде чем использовать переменную-мьютекс, следует сначала инициализировать ее, записав значение константы `PTHREAD_MUTEX_INITIALIZER` (только для статически размещаемых мьютексов) или вызвав функцию `pthread_mutex_init`. Если мьютекс размещается в динамической памяти (например, с помощью функции `malloc`), прежде чем освободить занимаемую память, необходимо вызвать функцию `pthread_mutex_destroy`.

```
#include <pthread.h>

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

int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

Обе возвращают 0 в случае успеха, код ошибки – в случае неудачи

Чтобы инициализировать мьютекс со значениями атрибутов по умолчанию, мы должны передать значение `NULL` в аргументе *attr*. Конкретные значения атрибутов мьютексов мы рассмотрим в разделе 12.4.

Запирается мьютекс вызовом функции pthread_mutex_lock. Если мьютекс уже заперт, вызывающий поток будет заблокирован, пока мьютекс не будет отперт. Мьютекс отпирается вызовом функции pthread_mutex_unlock.

Если поток не должен блокироваться при попытке запереть мьютекс, он может воспользоваться функцией pthread_mutex_trylock. Если к моменту вызова этой функции мьютекс будет отперт, функция запрет мьютекс и вернет значение 0. В противном случае pthread_mutex_trylock вернет код ошибки EBUSY.

```
#include <pthread.h>

int pthread_mutex_lock(pthread_mutex_t *mutex);

int pthread_mutex_trylock(pthread_mutex_t *mutex);

int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

Все три возвращают 0 в случае успеха, код ошибки – в случае неудачи

Пример

Листинг 11.5 иллюстрирует использование мьютексов для защиты структуры данных. Если более чем один поток работает с данными, размещаемыми динамически, мы можем предусмотреть в структуре данных счетчик ссылок на объект, чтобы освобождать память, только когда все потоки завершат работу с объектом.

Листинг 11.5. Использование мьютексов для защиты структур данных

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

struct foo {
    int             f_count;
    pthread_mutex_t f_lock;
    int             f_id
    /* ... другие поля структуры ... */
};

struct foo *
foo_alloc(int id) /* размещает объект в динамической памяти */
{
    struct foo *fp;

    if ((fp = malloc(sizeof(struct foo))) != NULL) {
        fp->f_count = 1;
        fp->f_id = id;
        if (pthread_mutex_init(&fp->f_lock, NULL) != 0) {
            free(fp);
            return(NULL);
        }
        /* ... продолжение инициализации ... */
    }
    return(fp);
```

```
}

void
foo_hold(struct foo *fp) /* наращивает счетчик ссылок на объект */
{
    pthread_mutex_lock(&fp->f_lock);
    fp->f_count++;
    pthread_mutex_unlock(&fp->f_lock);
}

void
foo_rele(struct foo *fp) /* освобождает ссылку на объект */
{
    pthread_mutex_lock(&fp->f_lock);
    if (--fp->f_count == 0) { /* последняя ссылка */
        pthread_mutex_unlock(&fp->f_lock);
        pthread_mutex_destroy(&fp->f_lock);
        free(fp);
    } else {
        pthread_mutex_unlock(&fp->f_lock);
    }
}
```

Перед увеличением или уменьшением счетчика ссылок и перед его проверкой на равенство нулю мьютекс запирается. При инициализации счетчика ссылок значением 1 в функции `foo_alloc` запирать мьютекс нет необходимости, поскольку пока только поток, который размещает структуру, имеет к ней доступ. Если бы в этой точке структура включалась в некий список, она могла бы быть обнаружена другими потоками, и тогда пришлось бы сначала запереть мьютекс.

Прежде чем приступить к работе с объектом, поток должен увеличить счетчик ссылок на него. По окончании работы с объектом поток должен удалить ссылку. Когда удаляется последняя ссылка, память, занимаемая объектом, освобождается.

Данный пример игнорирует возможность обнаружения объектов другими потоками перед вызовом `foo_hold`. Даже если счетчик ссылок равен нулю, для `foo_rele` было бы ошибкой пытаться освободить память, занимаемую объектом, так как другой поток мог запереть мьютекс вызовом `foo_hold`. Этой проблемы можно избежать, если перед освобождением памяти убедиться, что объект не может быть найден. Как это сделать, будет показано в примерах, следующих ниже.

11.6.2. Предотвращение тупиковых ситуаций

Поток может попасть в тупиковую ситуацию (deadlock), если попытается дважды захватить один и тот же мьютекс, но есть и менее очевидные способы. Например, тупиковая ситуация может возникнуть, когда в программе используется более одного мьютекса и мы позволим одному потоку удерживать первый мьютекс и пытаться запереть второй мьютекс, в то время как другой поток аналогичным образом может удерживать второй мьютекс и пытается

запереть первый. В результате ни один из потоков не сможет продолжить работу, поскольку каждый из них будет ждать освобождения ресурса, захваченного другим потоком, и возникнет тупиковая ситуация.

Тупиковых ситуаций можно избежать, жестко определив порядок, в котором производится захват ресурсов. Приведем пример. Предположим, что есть два мьютекса, A и B, которые необходимо запереть одновременно. Если все потоки сначала будут запирать мьютекс A, а потом B, тупиковой ситуации с этими мьютексами никогда не возникнет. Аналогично, если все потоки сначала будут запирать мьютекс B, а потом A, тупиковой ситуации с этими мьютексами также никогда не возникнет. Опасность попадания в тупиковую ситуацию возникает, только когда разные потоки могут попытаться запереть мьютексы в разном порядке.

Иногда архитектура приложения не позволяет заранее предопределить порядок захвата мьютексов. Если программа использует достаточно много мьютексов и структур данных, а доступные функции, которые работают с ними, не укладываются в достаточно простую иерархию, придется попробовать иной подход. Например, при невозможности запереть мьютекс можно отпереть захваченные мьютексы и повторить попытку немного позже. В этом случае во избежание блокировки потока можно использовать функцию `pthread_mutex_trylock`. Если мьютекс удалось запереть с помощью `pthread_mutex_trylock`, можно продолжить работу. Однако, если мьютекс запереть не удалось, можно отпереть уже захваченные мьютексы, освободить занятые ресурсы и повторить попытку немного позже.

Пример

В этом примере приводится версия из листинга 11.5, дополненная с целью продемонстрировать работу с двумя мьютексами. Во избежание тупиковой ситуации, которая может возникнуть при попытке одновременного захвата обоих ресурсов, во всех потоках используется один и тот порядок запирания мьютексов. Второй мьютекс защищает хеш-список структур `foo`. То есть мьютекс `hashlock` защищает хеш-таблицу `fh` и поле связи `f_next` в структуре `foo`. Доступ к остальным полям структуры `foo` производится под защитой мьютекса `f_lock`.

Листинг 11.6. Использование двух мьютексов

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

#define NHASH 29
#define HASH(id) (((unsigned long)id)%NHASH)

struct foo *fh[NHASH];

pthread_mutex_t hashlock = PTHREAD_MUTEX_INITIALIZER;

struct foo {
    int             f_count;
    pthread_mutex_t f_lock;
    int             f_id;
    struct foo     *f_next; /* защищается мьютексом hashlock */
```

```
    /* ... другие поля структуры ... */
};

struct foo *
foo_alloc(void) /* размещает объект в динамической памяти */
{
    struct foo *fp;
    int        idx;

    if ((fp = malloc(sizeof(struct foo))) != NULL) {
        fp->f_count = 1;
        fp->f_id = id;
        if (pthread_mutex_init(&fp->f_lock, NULL) != 0) {
            free(fp);
            return(NULL);
        }
        idx = HASH(fp);
        pthread_mutex_lock(&hashlock);
        fp->f_next = fh[idx];
        fh[idx] = fp->f_next;
        pthread_mutex_lock(&fp->f_lock);
        pthread_mutex_unlock(&hashlock);
        /* ... продолжение инициализации ... */
        pthread_mutex_unlock(&fp->f_lock);
    }
    return(fp);
}

void
foo_hold(struct foo *fp) /* добавить ссылку на объект */
{
    pthread_mutex_lock(&fp->f_lock);
    fp->f_count++;
    pthread_mutex_unlock(&fp->f_lock);
}

struct foo *
foo_find(int id) /* найти существующий объект */
{
    struct foo *fp;

    pthread_mutex_lock(&hashlock);
    for (fp = fh[HASH(id)]; fp != NULL; fp = fp->f_next) {
        if (fp->f_id == id) {
            foo_hold(fp);
            break;
        }
    }
    pthread_mutex_unlock(&hashlock);
    return(fp);
}

void
foo_rele(struct foo *fp) /* освободить ссылку на объект */
```

```
{
    struct foo *tfp;
    int        idx;

    pthread_mutex_lock(&fp->f_lock);
    if (fp->f_count == 1) { /* последняя ссылка */
        pthread_mutex_unlock(&fp->f_lock);
        pthread_mutex_lock(&hashlock);
        pthread_mutex_lock(&fp->f_lock);
        /* необходима повторная проверка условия */
        if (fp->f_count != 1) {
            fp->f_count--;
            pthread_mutex_unlock(&fp->f_lock);
            pthread_mutex_unlock(&hashlock);
            return;
        }
        /* удалить из списка */
        idx = HASH(fp->f_id);
        tfp = fh[idx];
        if (tfp == fp) {
            fh[idx] = fp->f_next;
        } else {
            while (tfp->f_next != fp)
                tfp = tfp->f_next;
            tfp->f_next = fp->f_next;
        }
        pthread_mutex_unlock(&hashlock);
        pthread_mutex_unlock(&fp->f_lock);
        pthread_mutex_destroy(&fp->f_lock);
        free(fp);
    } else {
        fp->f_count;
        pthread_mutex_unlock(&fp->f_lock);
    }
}
```

Сравнив листинги 11.6 и 11.5, можно заметить, что теперь функция размещения объекта в динамической памяти блокирует доступ к хеш-таблице, добавляет в нее новую структуру, а перед снятием блокировки с хеш-таблицы запирает новую структуру. Поскольку новая структура размещается в глобальном списке, ее может обнаружить любой другой поток, и поэтому мы вынуждены запирать ее, пока не будет закончена инициализация структуры.

Функция `foo_find` запирает хеш-таблицу и производит поиск запрошенной структуры. Если таковая будет найдена, мы увеличиваем в ней счетчик ссылок и возвращаем указатель на структуру. Обратите внимание, что здесь мы соблюдаем порядок захвата мьютексов, запирая мьютекс `hashlock` до того, как функция `foo_hold` запрет мьютекс `f_lock`.

Теперь перейдем к функции `foo_rele`, алгоритм работы которой несколько сложнее. Если освобождается последняя ссылка на объект, необходимо отпереть мьютекс `f_lock`, чтобы запереть `hashlock`, поскольку нам необходимо уда-

лить структуру из списка. После этого необходимо запереть мьютекс f_lock. Учитывая, что поток мог быть заблокирован во время повторной попытки захватить мьютексы, мы вынуждены повторить проверку необходимости удаления структуры. Если какой-либо другой поток нашел структуру и нарастил счетчик ссылок в ней, в то время как данный поток был заблокирован в ожидании освобождения мьютекса, мы просто уменьшаем счетчик ссылок, отпираем оба мьютекса и возвращаем управление.

Такой алгоритм работы с мьютексами достаточно сложен, поэтому нужно пересмотреть его. Алгоритм заметно упростится, если мьютекс hashlock будет защищать еще и счетчик ссылок. Мьютекс f_lock будет защищать все остальные поля структуры foo. Эти изменения отражены в листинге 11.7.

Листинг 11.7. Упрощенный вариант использования мьютексов

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

#define NHASH 29
#define HASH(id) (((unsigned long)id)%NHASH)

struct foo *fh[NHASH];
pthread_mutex_t hashlock = PTHREAD_MUTEX_INITIALIZER;

struct foo {
    int             f_count; /* защищается мьютексом hashlock */
    pthread_mutex_t f_lock;
    int             f_id;
    struct foo     *f_next;  /* защищается мьютексом hashlock */
    /* ... другие поля структуры ... */
};

struct foo *
foo_alloc(int id) /* размещает объект в динамической памяти */
{
    struct foo  *fp;
    int         idx;

    if ((fp = malloc(sizeof(struct foo))) != NULL) {
        fp->f_count = 1;
        fp->f_id = id;
        if (pthread_mutex_init(&fp->f_lock, NULL) != 0) {
            free(fp);
            return(NULL);
        }
        idx = HASH(id);
        pthread_mutex_lock(&hashlock);
        fp->f_next = fh[idx];
        fh[idx] = fp;
        pthread_mutex_lock(&fp->f_lock);
        pthread_mutex_unlock(&hashlock);
        /* ... продолжение инициализации ... */
        pthread_mutex_unlock(&fp->f_lock);
    }
```

```
    return(fp);
}

void
foo_hold(struct foo *fp) /* добавляет ссылку на объект */
{
    pthread_mutex_lock(&hashlock);
    fp->f_count++;
    pthread_mutex_unlock(&hashlock);
}

struct foo *
foo_find(int id) /* найти существующий объект */
{
    struct foo *fp;

    pthread_mutex_lock(&hashlock);
    for (fp = fh[HASH(id)]; fp != NULL; fp = fp->f_next) {
        if (fp->f_id == id) {
            fp->f_count++;
            break;
        }
    }
    pthread_mutex_unlock(&hashlock);
    return(fp);
}

void
foo_rele(struct foo *fp) /* освобождает ссылку на объект */
{
    struct foo  *tfp;
    int         idx;

    pthread_mutex_lock(&hashlock);
    if (--fp->f_count == 0) { /* последняя ссылка, удалить из списка */
        idx = HASH(fp->f_id);
        tfp = fh[idx];
        if (tfp == fp) {
            fh[idx] = fp->f_next;
        } else {
            while (tfp->f_next != fp)
                tfp = tfp->f_next;
            tfp->f_next = fp->f_next;
        }
        pthread_mutex_unlock(&hashlock);
        pthread_mutex_destroy(&fp->f_lock);
        free(fp);
    } else {
        pthread_mutex_unlock(&hashlock);
    }
}
```

Обратите внимание, насколько проще стала программа по сравнению с листингом 11.6. Когда мы стали использовать один и тот же мьютекс для защиты

хеш-списка и счетчика ссылок, отпала проблема соблюдения порядка захвата мьютексов. При разработке многопоточных приложений достаточно часто приходится идти на подобные компромиссы. Слишком грубая детализация блокировок в конечном итоге приведет к тому, что большинство потоков будут простаивать при попытках запереть один и тот же мьютекс, а преимущества многопоточной архитектуры приложения будут сведены к минимуму. Если детализация блокировок будет слишком мелкой, это существенно усложнит код, а производительность приложения снизится из-за избыточного количества мьютексов. Программист должен отыскать правильный баланс между производительностью и сложностью алгоритма и при этом выполнить все требования, связанные с захватом ресурсов.

11.6.3. Функция pthread_mutex_timedlock

Один из дополнительных примитивов управления мьютексами позволяет ограничить время блокировки потока при попытке захватить мьютекс, запертый другим потоком. Функция `pthread_mutex_timedlock` эквивалентна функции `pthread_mutex_lock`, но по истечении указанного тайм-аута `pthread_mutex_timedlock` вернет код ошибки `ETIMEDOUT`, не запирая мьютекс.

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

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

Возвращает 0 в случае успеха, код ошибки – в случае неудачи

Под тайм-аутом здесь понимается абсолютный момент времени (в противоположность относительному времени; то есть мы должны указать момент времени X, когда следует прекратить попытки приобрести блокировку, а не количество секунд Y, в течение которых следует ждать освобождения мьютекса, если он занят). Значение тайм-аута определяется в виде структуры `timespec`, хранящей время в секундах и наносекундах.

Пример

В листинге 11.8 демонстрируется, как использовать функцию `pthread_mutex_timedlock`, чтобы избежать блокировки потока навечно.

Листинг 11.8. Использование функции `pthread_mutex_timedlock`

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

int
main(void)
{
    int             err;
    struct timespec tout;
    struct tm       *tmp;
    char            buf[64];
```

```
    pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;

    pthread_mutex_lock(&lock);
    printf("мьютекс заперт\n");
    clock_gettime(CLOCK_REALTIME, &tout);
    tmp = localtime(&tout.tv_sec);
    strftime(buf, sizeof(buf), "%r", tmp);
    printf("текущее время: %s\n", buf);
    tout.tv_sec += 10;   /* 10 секунд, начиная от текущего времени */
    /* внимание: это может привести к тупиковой ситуации */
    err = pthread_mutex_timedlock(&lock, &tout);
    clock_gettime(CLOCK_REALTIME, &tout);
    tmp = localtime(&tout.tv_sec);
    strftime(buf, sizeof(buf), "%r", tmp);
    printf("текущее время: %s\n", buf);
    if (err == 0)
        printf("мьютекс снова заперт!\n");
    else
        printf("не получилось повторно запереть мьютекс: %s\n", strerror(err));
    exit(0);
}
```

Ниже приводится вывод программы из листинга 11.8.

```
$ ./a.out
мьютекс заперт
текущее время: 11:41:58 AM
текущее время: 11:42:08 AM
не получилось повторно запереть мьютекс: Connection timed out
```

Эта программа преднамеренно пытается повторно запереть уже запертый мьютекс, чтобы продемонстрировать работу функции `pthread_mutex_timedlock`. Данную стратегию не рекомендуется использовать на практике, потому что она может служить источником тупиковых ситуаций.

Обратите внимание, что протяженность интервала блокировки может варьироваться в некоторых пределах по следующим причинам: начальный момент времени может быть определен в середине текущей секунды, разрешение системных часов может быть недостаточно точным для поддержки желаемой точности тайм-аута, задержки в планировщике задач могут вызвать увеличение времени ожидания.

> Mac OS X 10.6.8 не поддерживает функцию `pthread_mutex_timedlock`, но FreeBSD 8.0, Linux 3.2.0 и Solaris 10 поддерживают ее, однако в Solaris эта функция до сих пор находится в библиотеке реального времени, `librt`. В Solaris 10 имеется также альтернативная функция, которой можно передать относительный тайм-аут.

11.6.4. Блокировки чтения–записи

Блокировки чтения-записи похожи на мьютексы, за исключением того, что они допускают более высокую степень параллелизма. Мьютексы могут иметь всего два состояния, закрытое и открытое, и только один поток может вла-

деть мьютексом в каждый момент времени. Блокировки чтения-записи могут иметь три состояния: режим блокировки для чтения, режим блокировки для записи и отсутствие блокировки. Режим блокировки для записи может установить только один поток, но установка режима блокировки для чтения доступна нескольким потокам одновременно.

Если блокировка чтения-записи установлена в режиме для записи, все потоки, которые попытаются захватить ее, будут приостановлены, пока блокировка не будет снята. Если блокировка чтения-записи установлена в режиме для чтения, все потоки, которые попытаются захватить ее для чтения, получат доступ к ресурсу, но если какой-либо поток попытается установить режим блокировки для записи, он будет приостановлен, пока не будет снята последняя блокировка для чтения. Различные реализации блокировок чтения-записи могут значительно различаться, но обычно, если блокировка для чтения уже установлена и имеется поток, который пытается установить блокировку для записи, остальные потоки, которые пытаются получить блокировку для чтения, будут приостановлены. Это предотвращает возможность блокирования пишущих потоков непрекращающимися запросами на получение блокировки для чтения.

Блокировки чтения-записи прекрасно подходят для ситуаций, когда чтение данных производится намного чаще, чем запись. Когда блокировка чтения-записи установлена в режиме для записи, можно безопасно выполнять модификацию защищаемых ею данных, поскольку только один поток может владеть блокировкой для записи. Когда блокировка чтения-записи установлена в режиме для чтения, защищаемые ею данные могут быть безопасно прочитаны несколькими потоками, если эти потоки смогли получить блокировку для чтения.

Блокировки чтения-записи еще называют совместно-исключающими блокировками (shared-exclusive locks). Когда блокировка чтения-записи установлена в режиме для чтения, говорят, что блокировка находится в режиме совместного использования. Когда блокировка чтения-записи установлена в режиме для записи, говорят, что блокировка находится в режиме исключительного использования.

Как и в случае с мьютексами, блокировки чтения-записи должны быть инициализированы перед их использованием и разрушены перед освобождением занимаемой ими памяти.

```
#include <pthread.h>

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

int pthread_rwlock_destroy(pthread_rwlock_t *rwlock);
```

Обе возвращают 0 в случае успеха, код ошибки – в случае неудачи

Функция `pthread_rwlock_init` инициализирует блокировку чтения-записи. Если в аргументе *attr* передается пустой указатель, блокировка инициализиру-

ется с атрибутами по умолчанию. Атрибуты блокировок чтения-записи мы рассмотрим в разделе 12.4.2.

Стандарт Single UNIX Specification определяет константу PTHREAD_RWLOCK_INITIALIZER, как расширение XSI, которую можно использовать для инициализации статических блокировок чтения-записи, когда значений по умолчанию для атрибутов вполне достаточно.

Перед освобождением памяти, занимаемой блокировкой чтения-записи, нужно вызвать функцию pthread_rwlock_destroy, чтобы освободить все занимаемые блокировкой ресурсы. Функция pthread_rwlock_init размещает все необходимые для блокировки ресурсы, а pthread_rwlock_destroy освобождает их. Если освободить память, занимаемую блокировкой чтения-записи без предварительного обращения к функции pthread_rwlock_destroy, все ресурсы, занимаемые блокировкой, будут потеряны для системы.

Чтобы установить блокировку в режиме для чтения, необходимо вызвать функцию pthread_rwlock_rdlock. Чтобы установить блокировку в режиме для записи, необходимо вызвать функцию pthread_rwlock_wrlock. Независимо от режима блокировки чтения-записи, ее снятие выполняется функцией pthread_rwlock_unlock.

```
#include <pthread.h>

int pthread_rwlock_rdlock(pthread_rwlock_t *rwlock);

int pthread_rwlock_wrlock(pthread_rwlock_t *rwlock);

int pthread_rwlock_unlock(pthread_rwlock_t *rwlock);
```

Все три возвращают 0 в случае успеха, код ошибки – в случае неудачи

Реализации могут ограничивать количество блокировок, установленных в режиме совместного использования, поэтому обязательно нужно проверять значение, возвращаемое функцией pthread_rwlock_rdlock. Даже когда функции pthread_rwlock_wrlock и pthread_rwlock_unlock возвращают код ошибки, нет необходимости проверять возвращаемые значения этих функций, если схема наложения блокировок разработана надлежащим образом. Эти функции могут вернуть код ошибки, только когда блокировка не была инициализирована или когда может возникнуть тупиковая ситуация при попытке повторно установить уже установленную блокировку. Однако вы должны помнить, что некоторые реализации могут определять дополнительные коды ошибок.

Стандарт Single UNIX Specification определяет дополнительные версии примитивов для работы с блокировками, которые могут использоваться для проверки состояния блокировки.

```
#include <pthread.h>

int pthread_rwlock_tryrdlock(pthread_rwlock_t *rwlock);

int pthread_rwlock_trywrlock(pthread_rwlock_t *rwlock);
```

Обе возвращают 0 в случае успеха, код ошибки – в случае неудачи

Если блокировка была успешно установлена, эти функции возвращают значение 0. В противном случае возвращается код ошибки EBUSY. Эти функции могут использоваться, когда невозможно заранее предопределить порядок установки блокировок, чтобы избежать тупиковых ситуаций, которые мы обсуждали ранее.

Пример

Программа в листинге 11.9 иллюстрирует применение блокировок чтения-записи. Очередь запросов на выполнение заданий защищается единственной блокировкой чтения-записи. Этот пример является одной из возможных реализаций приложения, представленного на рис. 11.1, где множество потоков получают задания, назначаемые им главным потоком.

Листинг 11.9. Использование блокировки чтения-записи

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

struct job {
    struct job *j_next;
    struct job *j_prev;
    pthread_t  j_id; /* сообщает, какой поток выполняет это задание */
    /* ... другие поля структуры ... */
};

struct queue {
    struct job       *q_head;
    struct job       *q_tail;
    pthread_rwlock_t  q_lock;
};

/*
 * Инициализация очереди.
 */
int
queue_init(struct queue *qp)
{
    int err;

    qp->q_head = NULL;
    qp->q_tail = NULL;
    err = pthread_rwlock_init(&qp->q_lock, NULL);
    if (err != 0)
        return(err);
    /* ... продолжение инициализации ... */
    return(0);
}

/*
 * Добавить задание в начало очереди.
 */
void
job_insert(struct queue *qp, struct job *jp)
```

```
{
    pthread_rwlock_wrlock(&qp->q_lock);
    jp->j_next = qp->q_head;
    jp->j_prev = NULL;
    if (qp->q_head != NULL)
        qp->q_head->j_prev = jp;
    else
        qp->q_tail = jp; /* список был пуст */
    qp->q_head = jp;
    pthread_rwlock_unlock(&qp->q_lock);
}

/*
 * Добавить задание в конец очереди.
 */
void
job_append(struct queue *qp, struct job *jp)
{
    pthread_rwlock_wrlock(&qp->q_lock);
    jp->j_next = NULL;
    jp->j_prev = qp->q_tail;
    if (qp->q_tail != NULL)
        qp->q_tail->j_next = jp;
    else
        qp->q_head = jp; /* список был пуст */
    qp->q_tail = jp;
    pthread_rwlock_unlock(&qp->q_lock);
}

/*
 * Удалить задание из очереди.
 */
void
job_remove(struct queue *qp, struct job *jp)
{
    pthread_rwlock_wrlock(&qp->q_lock);
    if (jp == qp->q_head) {
        qp->q_head = jp->j_next;
        if (qp->q_tail == jp)
            qp->q_tail = NULL;
        else
             jp->j_next->j_prev = jp->j_prev;
    } else if (jp == qp->q_tail) {
        qp->q_tail = jp->j_prev;
        jp->j_prev->j_next = jp->j_next;
    } else {
        jp->j_prev->j_next = jp->j_next;
        jp->j_next->j_prev = jp->j_prev;
    }
    pthread_rwlock_unlock(&qp->q_lock);
}
```

```
/* Найти задание для потока с заданным идентификатором. */
struct job *
job_find(struct queue *qp, pthread_t id)
{
    struct job *jp;

    if (pthread_rwlock_rdlock(&qp->q_lock) != 0)
        return(NULL);

    for (jp = qp->q_head; jp != NULL; jp = jp->j_next)
        if (pthread_equal(jp->j_id, id))
            break;

    pthread_rwlock_unlock(&qp->q_lock);
    return(jp);
}
```

В этом примере блокировка чтения-записи очереди устанавливается в режиме для записи, только когда необходимо добавить новое задание в очередь или удалить задание из очереди. Когда нужно выполнить поиск задания в очереди, мы устанавливаем блокировку в режиме для чтения, допуская возможность поиска заданий несколькими рабочими потоками одновременно. В данном случае использование блокировки чтения-записи дает прирост производительности, только если поиск заданий в очереди выполняется чаще, чем добавление или удаление.

Рабочие потоки извлекают из очереди только те задания, которые соответствуют их идентификаторам. Поскольку сама структура с заданием используется только одним потоком, для организации доступа к ней не требуется дополнительных блокировок.

11.6.5. Блокировки чтения–записи с тайм-аутом

Как и в случае с мьютексами, стандарт Single UNIX Specification определяет функции для приобретения блокировок чтения-записи с тайм-аутом, не позволяющие приложениям заблокироваться навечно при попытке приобрести блокировку чтения-записи. Это – функции `pthread_rwlock_timedrdlock` и `pthread_rwlock_timedwrlock`.

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

int pthread_rwlock_timedrdlock(pthread_rwlock_t *restrict rwlock,
                               const struct timespec *restrict tsptr);

int pthread_rwlock_timedwrlock(pthread_rwlock_t *restrict rwlock,
                               const struct timespec *restrict tsptr);
```

Обе возвращают 0 в случае успеха, код ошибки – в случае неудачи

Эти функции действуют подобно своим «неограниченным» эквивалентам. В аргументе *tsptr* они принимают указатель на структуру `timespec`, определяющую момент времени, когда следует прекратить попытки приобрести бло-

кировку. В случае невозможности приобрести блокировку по истечении тайм-аута эти функции возвращают код ошибки ETIMEDOUT. Как и в функции pthread_mutex_timedlock, тайм-аут определяет абсолютный момент времени, а не интервал ожидания.

11.6.6. Переменные состояния

Переменные состояния (condition variables) – еще один механизм синхронизации потоков. Переменные состояния предоставляют потокам своеобразное место встречи. При использовании вместе с мьютексами переменные состояния позволяют потокам ожидать наступления некоторого события, избегая состояния гонки.

Сами переменные состояния защищаются мьютексами. Прежде чем изменить значение такой переменной, поток должен захватить мьютекс. Другие потоки не будут замечать изменений переменной, пока не попытаются захватить этот мьютекс, потому что для оценки переменной состояния необходимо запереть мьютекс.

Переменная состояния, представленная типом pthread_cond_t, должна быть инициализирована перед использованием. При статическом размещении переменной можно присвоить ей значение константы PTHREAD_COND_INITIALIZER, но если переменная состояния размещается динамически, ее следует инициализировать вызовом функции pthread_cond_init.

Для уничтожения переменной состояния перед освобождением занимаемой ею памяти используется функция pthread_cond_destroy.

```
#include <pthread.h>

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

int pthread_cond_destroy(pthread_cond_t *cond);
```

Обе возвращают 0 в случае успеха, код ошибки – в случае неудачи

Если в аргументе *attr* передается пустой указатель, переменная состояния будет инициализирована со значениями атрибутов по умолчанию. Атрибуты переменных состояния мы рассмотрим в разделе 12.4.3.

Функция pthread_cond_wait ожидает, пока переменная не перейдет в истинное состояние. Если нужно ограничить время ожидания заданным интервалом, используется функция pthread_cond_timedwait.

Мьютекс, передаваемый функции pthread_cond_wait, защищает доступ к переменной состояния. Вызывающий поток передает его функции в запертом состоянии, а функция атомарно помещает вызывающий поток в список потоков, ожидающих изменения состояния переменной, и отпирает мьютекс. Это исключает вероятность, что переменная изменит состояние между моментом ее проверки и моментом приостановки потока, благодаря чему поток не про-

пустит наступление ожидаемого события. Когда функция pthread_cond_wait возвращает управление, мьютекс снова запирается.

```
#include <pthread.h>

int pthread_cond_wait(pthread_cond_t *restrict cond,
                      pthread_mutex_t *restrict mutex);

int pthread_cond_timedwait(pthread_cond_t *restrict cond,
                           pthread_mutex_t *restrict mutex,
                           const struct timespec *restrict tsptr);
```

Обе возвращают 0 в случае успеха, код ошибки – в случае неудачи

Функция pthread_cond_timedwait работает аналогично, но дополнительно предоставляет возможность ограничить время ожидания. Значение аргумента *tpstr* определяет, как долго поток будет ожидать наступления события. Время тайм-аута задается структурой timespec.

Как было показано в листинге 11.8, в этой структуре следует указывать абсолютное время, а не относительное. Например, если потребуется ограничить время ожидания 3 минутами, мы должны сохранить в этой структуре не 3 минуты, а текущее время + 3 минуты.

Для этого можно воспользоваться функцией clock_gettime (раздел 6.10), возвращающей текущее время в виде структуры timespec. Однако эта функция поддерживается пока не всеми платформами. Вместо нее можно использовать функцию gettimeofday, чтобы получить текущее время в виде структуры timeval, и затем преобразовать ее в структуру timespec. Чтобы получить абсолютное время для аргумента *tsptr*, можно использовать следующую функцию (предполагается, что продолжительность интервала времени измеряется в минутах):

```
#include <sys/time.h>
#include <stdlib.h>

void
maketimeout(struct timespec *tsp, long minutes)
{
    struct timeval now;

    /* получить текущее время */
    gettimeofday(&now);
    tsp->tv_sec = now.tv_sec;
    tsp->tv_nsec = now.tv_usec * 1000; /* микросекунды в наносекунды */
    /* добавить величину тайм-аута */
    tsp->tv_sec += minutes * 60;
}
```

Если тайм-аут истечет до появления ожидаемого события, функция pthread_cont_timedwait запрет мьютекс и вернет код ошибки ETIMEDOUT. Когда функция pthread_cont_wait или pthread_cont_timedwait завершится успехом, поток должен оценить значение переменной, поскольку к этому моменту другой поток мог изменить его.

Для передачи сообщения о наступлении события существуют две функции. Функция pthread_cond_signal возобновит работу одного потока, ожидающего наступления события, а pthread_cond_broadcast – всех потоков, ожидающих наступления события.

> Для упрощения реализации стандарт POSIX допускает, чтобы функция pthread_cond_signal возобновляла работу нескольких потоков.

```
#include <pthread.h>

int pthread_cond_signal(pthread_cond_t *cond);

int pthread_cond_broadcast(pthread_cond_t *cond);
```

Обе возвращают 0 в случае успеха, код ошибки – в случае неудачи

Когда вызывается функция pthread_cond_signal, говорят, что посылается сигнал о наступлении события. Мы должны сделать все возможное, чтобы сигнал о наступлении события посылался только после изменения состояния переменной.

Пример

В листинге 11.10 приводится пример синхронизации потоков с помощью переменных состояния и мьютексов.

Листинг 11.10. Пример использования переменных состояния

```
#include <pthread.h>

struct msg {
    struct msg *m_next;
    /* ... другие поля структуры ... */
};

struct msg *workq;

pthread_cond_t qready = PTHREAD_COND_INITIALIZER;

pthread_mutex_t qlock = PTHREAD_MUTEX_INITIALIZER;

void
process_msg(void)
{
    struct msg *mp;

    for (;;) {
        pthread_mutex_lock(&qlock);
        while (workq == NULL)
            pthread_cond_wait(&qready, &qlock);
        mp = workq;
        workq = mp->m_next;
        pthread_mutex_unlock(&qlock);
        /* обработка сообщения mp */
```

```
        }
    }

    void
    enqueue_msg(struct msg *mp)
    {
        pthread_mutex_lock(&qlock);
        mp->m_next = workq;
        workq = mp;
        pthread_mutex_unlock(&qlock);
        pthread_cond_signal(&qready);
    }
```

В данном случае отслеживается состояние очереди сообщений. Переменная состояния защищена мьютексом, а определение изменения состояния производится в цикле `while`. Чтобы поместить очередное сообщение в очередь, необходимо запереть мьютекс, но чтобы послать сигнал ожидающим потокам, запирать мьютекс не нужно. Такой вариант, когда сигнал посылается после отпирания мьютекса, будет прекрасно работать, даже если какой-либо поток успеет возобновить работу до передачи сигнала. Поскольку наступление события проверяется в цикле, это не представляет проблемы: поток просто возобновит работу, убедится, что очередь пуста, и опять перейдет в режим ожидания. Если логика программы не допускает подобной гонки, тогда необходимо сначала вызвать `pthread_cond_signal`, а затем отпереть мьютекс.

11.6.7. Циклические блокировки

Циклическая блокировка (spin lock) подобна мьютексу, но блокируемый процесс не приостанавливается, а вращается (spinning) в цикле ожидания, пока не приобретет блокировку. Циклическую блокировку можно использовать в ситуациях, когда блокировка нужна на очень короткий промежуток времени, а накладные расходы на перепланирование потока выполнения выглядят непомерно большими.

Циклические блокировки часто используются как низкоуровневые примитивы для реализации блокировок других типов. В зависимости от архитектуры системы они могут быть реализованы с использованием инструкций «проверил и установил». Несмотря на высокую эффективность, они могут приводить к напрасной трате вычислительных ресурсов: пока поток вращается в цикле, ожидая освобождения блокировки, процессор не может заняться чем-то другим. Именно поэтому циклические блокировки должны приобретаться на очень короткие промежутки времени.

Циклические блокировки особенно полезны при использовании невытесняемого ядра (nonpreemptive kernel): помимо поддержки механизма взаимоисключения (mutual exclusion) они блокируют прерывания, поэтому исключается вероятность тупиковой ситуации, когда обработчик прерывания может попытаться приобрести уже запертую циклическую блокировку (прерывания в данном контексте можно рассматривать как одну из разновидностей вытеснения). В ядрах этого типа обработчики прерываний не могут приоста-

навливаться, поэтому единственные примитивы синхронизации, которые они могут использовать, – это циклические блокировки.

Однако на уровне пользовательского приложения циклические блокировки не так полезны, если только приложение не выполняется с классом планирования в режиме реального времени, не допускающим вытеснение. Пользовательские потоки, выполняющиеся с классом планирования в режиме разделения времени, могут вытесняться после исчерпания выделенного кванта времени или при появлении готового к выполнению потока с более высоким приоритетом. В этих случаях поток, приобретший циклическую блокировку, будет приостановлен, и другие потоки, заблокированные на блокировке, будут продолжать вращаться в цикле ожидания дольше, чем предполагалось.

Многие реализации мьютексов настолько эффективны, что производительность приложений с использованием мьютексов не уступает производительности тех же приложений с циклическими блокировками. На практике некоторые реализации мьютексов вращаются ограниченное время в цикле ожидания, пытаясь приобрести мьютекс, и только когда счетчик циклов ожидания превысит пороговое значение, приостанавливают поток. Такой подход в сочетании с возможностями современных процессоров, позволяющими им переключать контекст выполнения все быстрее и быстрее, делают циклические блокировки пригодными лишь в редких ситуациях.

Интерфейс циклических блокировок похож на интерфейс мьютексов, что позволяет легко заменять одни другими. Инициализация циклической блокировки выполняется вызовом функции `pthread_spin_init`. Чтобы уничтожить циклическую блокировку, следует вызвать функцию `pthread_spin_destroy`.

```
#include <pthread.h>

int pthread_spin_init(pthread_spinlock_t *lock, int pshared);

int pthread_spin_destroy(pthread_spinlock_t *lock);
```

Обе возвращают 0 в случае успеха, код ошибки – в случае неудачи

При инициализации циклической блокировки можно указать только один атрибут, который имеет смысл, только если платформа поддерживает расширение Thread Process-Shared Synchronization (Синхронизация потоков между процессами) (в настоящее время это расширение перенесено в разряд базовых спецификаций Single UNIX Specification; табл. 2.5). В аргументе *pshared* передается признак разделения блокировки *между-процессами* (*process-shared*), определяющий ее доступность. Если в нем передать значение `PTHREAD_PROCESS_SHARED`, циклическая блокировка будет доступна потокам выполнения, имеющим доступ к памяти, где хранится блокировка, – даже потокам в других процессах. В противном случае аргумент *pshared* следует устанавливать в значение `PTHREAD_PROCESS_PRIVATE`, и в этом случае циклическая блокировка будет доступна только потокам процесса, инициализировавшего ее.

Запереть циклическую блокировку можно с помощью `pthread_spin_lock`, которая будет крутиться в цикле ожидания, пока блокировка не будет приобрете-

на, или pthread_spin_trylock, которая вернет код ошибки EBUSY, если блокировка не может быть приобретена немедленно. Обратите внимание, что pthread_spin_trylock не выполняет цикл ожидания. Независимо от того, каким способом блокировка была заперта, ее можно освободить вызовом pthread_spin_unlock.

```
#include <pthread.h>

int pthread_spin_lock(pthread_spinlock_t *lock);

int pthread_spin_trylock(pthread_spinlock_t *lock);

int pthread_spin_unlock(pthread_spinlock_t *lock);
```

Все возвращают 0 в случае успеха, код ошибки – в случае неудачи

Обратите внимание, что если циклическая блокировка свободна, функция pthread_spin_lock может запереть ее, не выполняя цикл ожидания. Стандарты не определяют, что должна делать реализация, если поток попытается приобрести блокировку, которой уже владеет. В этом случае вызов pthread_spin_lock может вернуть код ошибки EDEADLK (или какой-то другой) или попасть в бесконечный цикл ожидания. Поведение зависит от реализации. Также стандарты не определяют, что должна делать реализация, если поток попытается освободить незапертую блокировку.

Если pthread_spin_lock или pthread_spin_trylock вернула 0, следовательно, циклическая блокировка была успешно заперта. Необходимо проявлять особую осторожность, чтобы не вызвать какую-нибудь функцию, которая может приостановить поток, пока он удерживает циклическую блокировку. Иначе мы впустую будем тратить процессорное время, увеличивая продолжительность времени, которое другие потоки потратят при попытке приобретения этой циклической блокировки.

11.6.8. Барьеры

Барьеры (barriers) – это механизм синхронизации, который можно использовать для координации действий нескольких потоков, выполняющихся одновременно. Барьер позволяет каждому потоку дождаться момента, когда все сотрудничающие с ним потоки достигнут той же точки, и продолжить работу. Мы уже познакомились с одной из разновидностей барьеров – функцией pthread_join, действующей как барьер, позволяя одному потоку дождаться завершения другого.

Однако объекты барьеров более универсальны, чем эта функция. Они дают возможность любому количеству потоков дождаться, пока все потоки завершат обработку, но при этом потоки не обязаны завершаться. Они могут продолжить работу, когда все потоки достигнут барьера.

Инициализировать барьер можно с помощью функции pthread_barrier_init, а уничтожить – с помощью функции pthread_barrier_destroy.

При инициализации барьера в аргументе *count* передается количество потоков, которые должны достигнуть барьера, прежде чем всем потокам будет по-

зволено продолжить работу. В аргументе *attr* передаются атрибуты объекта барьера, с которыми мы познакомимся в следующей главе. А пока достаточно знать, что если передать в аргументе *attr* пустой указатель (NULL), барьер будет инициализирован со значениями атрибутов по умолчанию. Если функция pthread_barrier_init выделяет какие-либо ресурсы для барьера, эти ресурсы будут освобождены функцией pthread_barrier_destroy.

```
#include <pthread.h>

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

int pthread_barrier_destroy(pthread_barrier_t *barrier);
```

Обе возвращают 0 в случае успеха, код ошибки – в случае неудачи

Чтобы показать, что поток выполнил свое задание и готов ждать, когда другие потоки достигнут барьера, он должен вызвать функцию pthread_barrier_wait.

```
#include <pthread.h>

int pthread_barrier_wait(pthread_barrier_t *barrier);
```

Возвращает 0 или PTHREAD_BARRIER_SERIAL_THREAD в случае успеха, код ошибки– в случае неудачи

Поток, вызвавший pthread_barrier_wait, приостанавливается, если количество ожидающих потоков не сравнялось со счетчиком барьера (устанавливается вызовом функции pthread_barrier_init). Если поток оказался последним, достигшим барьера, вызов pthread_barrier_wait возобновит работу всех ожидающих потоков.

В одном потоке (выбранном произвольно) pthread_barrier_wait вернет значение PTHREAD_BARRIER_SERIAL_THREAD. В остальных потоках она вернет 0. Это дает возможность одному из потоков взять на себя руководящие функции и заняться обработкой результатов, произведенных всеми другими потоками.

Как только количество потоков достигнет счетчика в объекте барьера и все ожидающие потоки будут разблокированы, барьер можно использовать повторно. Однако изменить счетчик барьера нельзя иначе, как вызвав функцию pthread_barrier_destroy и за ней функцию pthread_barrier_init с другим значением счетчика.

Пример

В листинге 11.11 демонстрируется, как можно использовать барьер для синхронизации потоков, сотрудничающих над решением общей задачи.

Листинг 11.11. Использование барьера

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

```
#include <limits.h>
#include <sys/time.h>

#define NTHR 8              /* количество потоков */
#define NUMNUM 8000000L     /* количество чисел для сортировки */
#define TNUM (NUMNUM/NTHR) /* количество чисел для одного потока */

long nums[NUMNUM];
long snums[NUMNUM];

pthread_barrier_t b;

#ifdef SOLARIS
#define heapsort qsort
#else
extern int heapsort(void *, size_t, size_t,
                    int (*)(const void *, const void *));
#endif

/*
 * Сравнивает два длинных целых (вспомогательная функция для heapsort)
 */
int
complong(const void *arg1, const void *arg2)
{
    long l1 = *(long *)arg1;
    long l2 = *(long *)arg2;
    if (l1 == l2)
        return 0;
    else if (l1 < l2)
        return -1;
    else
        return 1;
}

/*
 * Рабочий поток, сортирующий фрагмент массива чисел.
 */
void *
thr_fn(void *arg)
{
    long idx = (long)arg;

    heapsort(&nums[idx], TNUM, sizeof(long), complong);
    pthread_barrier_wait(&b);

    /*
     * Выполнить дополнительные операции при необходимости ...
     */
    return((void *)0);
}

/*
 * Выполняет слияние результатов сортировки фрагментов.
 */
```

```
void
merge()
{
    long     idx[NTHR];
    long     i, minidx, sidx, num;

    for (i = 0; i < NTHR; i++)
        idx[i] = i * TNUM;
    for (sidx = 0; sidx < NUMNUM; sidx++) {
        num = LONG_MAX;
        for (i = 0; i < NTHR; i++) {
            if ((idx[i] < (i+1)*TNUM) && (nums[idx[i]] < num)) {
                num = nums[idx[i]];
                minidx = i;
            }
        }
        snums[sidx] = nums[idx[minidx]];
        idx[minidx]++;
    }
}

int
main()
{
    unsigned long   i;
    struct timeval  start, end;
    long long       startusec, endusec;
    double          elapsed;
    int             err;
    pthread_t       tid;

    /*
     * Создать начальный массив чисел для сортировки.
     */
    srandom(1);
    for (i = 0; i < NUMNUM; i++)
        nums[i] = random();

    /*
     * Запустить 8 потоков для сортировки массива.
     */
    gettimeofday(&start, NULL);
    pthread_barrier_init(&b, NULL, NTHR+1);
    for (i = 0; i < NTHR; i++) {
        err = pthread_create(&tid, NULL, thr_fn, (void *)(i * TNUM));
        if (err != 0)
            err_exit(err, "невозможно создать поток");
    }
    pthread_barrier_wait(&b);
    merge();
    gettimeofday(&end, NULL);
```

```
        /*
         * Вывести отсортированный массив.
         */
        startusec = start.tv_sec * 1000000 + start.tv_usec;
        endusec = end.tv_sec * 1000000 + end.tv_usec;
        elapsed = (double)(endusec - startusec) / 1000000.0;
        printf("продолжительность сортировки (сек.): %.4f\n", elapsed);
        for (i = 0; i < NUMNUM; i++)
            printf("%ld\n", snums[i]);
        exit(0);
    }
```

Этот пример демонстрирует использование барьера в простой ситуации, когда все потоки решают общую задачу. В более сложных ситуациях, после возврата из функции `pthread_barrier_wait`, рабочие потоки могут приступать к решению других задач.

В данном примере мы использовали восемь потоков, чтобы распределить между ними работу по сортировке восьми миллионов чисел. Каждый поток сортирует один миллион чисел, применяя алгоритм пирамидальной сортировки (heapsort) [Knuth 1998]. Когда все потоки завершат работу, главный поток выполняет объединение результатов.

Нам не потребовалось использовать возвращаемое значение `PTHREAD_BARRIER_SERIAL_THREAD` функции `pthread_barrier_wait`, чтобы решить, какой поток будет объединять результаты, потому что эту работу выполняет главный поток. Именно поэтому мы указали счетчик потоков на единицу больше, чем количество рабочих потоков, – главный поток учитывается как один из ожидающих на барьере.

Если написать программу, выполняющую сортировку 8 миллионов чисел с помощью алгоритма пирамидальной сортировки в единственном потоке, можно заметить, насколько быстрее выполняется программа в листинге 11.11. В системе с 8 ядрами однопоточная программа сортирует 8 миллионов чисел за 12,14 секунды. В той же системе программа с 8 потоками, выполняющимися одновременно, и одним потоком, объединяющим результаты, сортирует тот же массив из 8 миллионов чисел за 1,91 секунды, в 6 раз быстрее.

11.7. Подведение итогов

В этой главе мы обсуждали концепцию потоков и примитивы POSIX.1 для работы с ними. Мы также коснулись проблемы синхронизации потоков. Были рассмотрены пять фундаментальных механизмов синхронизации – мьютексы, блокировки чтения-записи и переменные состояния, циклические блокировки и барьеры – и их применение для организации доступа к совместно используемым ресурсам.

Упражнения

11.1. Измените программу из листинга 11.3 таким образом, чтобы она корректно передавала структуру данных между потоками.

11.2. Изучите листинг 11.9 и скажите, какая дополнительная синхронизация должна быть предусмотрена (если она необходима), чтобы позволить главному потоку изменять идентификатор потока в задании. Как это повлияет на функцию `job_remove`?

11.3. Примените технику, показанную в листинге 11.10, к программе (рис. 11.1 и листинг 11.9) для реализации функции рабочего потока. Не забудьте дополнить функцию `queue_init` инициализацией переменной состояния и измените функции `job_insert` и `job_append` так, чтобы они посылали сигналы рабочим потокам. Какие сложности при этом возникнут?

11.4. Какую последовательность действий можно считать правильной?

1. Запереть мьютекс (`pthread_mutex_lock`).
2. Изменить переменную состояния, защищаемую мьютексом.
3. Послать сигнал ожидающим потокам (`pthread_cond_broadcast`).
4. Отпереть мьютекс (`pthread_mutex_unlock`).

или

1. Запереть мьютекс (`pthread_mutex_lock`).
2. Изменить переменную состояния, защищаемую мьютексом.
3. Отпереть мьютекс (`pthread_mutex_unlock`).
4. Послать сигнал ожидающим потокам (`pthread_cond_broadcast`).

11.5. Какие примитивы синхронизации можно было бы использовать для реализации барьера? Реализуйте функцию `pthread_barrier_wait`.