ОПЕРАЦИОННЫЕ СИСТЕМЫ И СИСТЕМНОЕ ПРОГРАММИРОВАНИЕ

Лекция 17 — Спинлоки и RW-блокировки

Преподаватель: Поденок Леонид Петрович, 505a-5

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

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

Кафедра ЭВМ, 2024

2024.05.23

Оглавление

Дополнительные вызовы pthread

pthread_{get|set}name_np() — получить/установить имя потока

```
#define _GNU_SOURCE             // See feature_test_macros(7)
#include <pthread.h>
int pthread_setname_np(pthread_t thread, const char *name);
int pthread_getname_np(pthread_t thread, char *name, size_t len);
```

По умолчанию все потоки, созданные с помощью `pthread_create( )`, наследуют имя программы.

`pthread_getname_np( )` -- получить имя потока

`thread` – указывает поток, имя которого нужно получить.
`name` – буфер для имени, сюда возвращается имя потока.
`len` – указывает количество байтов, доступных для размещения имени.
Буфер, указанный как **`name`**, должен иметь длину не менее 16 символов.
Возвращаемое имя потока в выходном буфере будет завершено нулем.

`pthread_setname_np( )` -- установить имя потока

Это может быть полезно для отладки многопоточных приложений.
Имя потока представляет собой строку на языке C, длина которой ограничена 16 символами, включая завершающий нулевой байт (**`'\0'`**).
`thread` – указывает поток, имя которого нужно изменить.
`name` – указывает новое имя.

Возвращаемое значение

В случае успеха эти функции возвращают 0.

При ошибке они возвращают ненулевой номер ошибки.

Ошибки

Функция **pthread_setname_np()** может завершиться ошибкой со следующей ошибкой:

ERANGE — Длина указанной строки, на которую указывает имя, превышает допустимый предел.

Функция **pthread_getname_np()** может завершиться ошибкой со следующей ошибкой:

ERANGE — Буфер, указанный с помощью name и len, слишком мал для хранения имени потока.

Если ни одна из этих функций не может открыть файл **/proc/self/task/[tid]/comm**, то вызов может завершиться ошибкой с одной из ошибок, описанных в **open(2)**.

Примечание

pthread_setname_np() выполняет внутреннюю запись в файл связи для конкретного потока в файловой системе **/proc** с именем **/proc/self/task/[tid]/comm**.

Функция **pthread_getname_np()** извлекает имя из этого же места.

Пример

Программа ниже демонстрирует использование **pthread_setname_np()** и **pthread_getname_np()**.

В следующем сеансе оболочки показан пример запуска программы:

```
$ ./name
Создаем поток. Имя по-умолчанию: name
Имя потока после установки имени: THREADFOO.
Готово
```

Программный код

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

#define NAMELEN 16

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

static void *threadfunc(void *parm) {

    sleep(5);           // allow main program to set the thread name
    return NULL;
}

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

    pthread_t thread;
    char      thread_name[NAMELEN];

    int rc = pthread_create(&thread, NULL, threadfunc, NULL);
    if (rc != 0) {
        errExitEN(rc, "pthread_create");
    }
```

```
    rc = pthread_getname_np(thread,                                  // поток
                            thread_name,                             // его имя
                            NAMELEN);                                // длина имени
    if (rc != 0) {
        errExitEN(rc, "pthread_getname_np");
    }

    printf("Создаем поток. Имя по-умолчанию: %s\n", thread_name);
    rc = pthread_setname_np(thread,                                  // поток
                            (argc > 1) ? argv[1] : "THREADFOO");     // его новое имя
    if (rc != 0) {
        errExitEN(rc, "pthread_setname_np");
    }
    sleep(2);
    rc = pthread_getname_np(thread,                                  // поток
                            thread_name,                             // его имя
                            (argc > 2)? atoi(argv[1]) : NAMELEN); // длина имени
    if (rc != 0)
        errExitEN(rc, "pthread_getname_np");

    printf("Имя потока после установки имени: %s.\n", thread_name);

    rc = pthread_join(thread, NULL);
    if (rc != 0)
        errExitEN(rc, "pthread_join");

    printf("Готово\n");
    exit(EXIT_SUCCESS);
}
```

pthread_getcpuclockid() — получить идентификатор часов для потока

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

int pthread_getcpuclockid(pthread_t thread, clockid_t *clock_id);
```

Функция **pthread_getcpuclockid()** возвращает идентификатор часов для потока **thread**.

Возвращаемое значение
В случае успеха эта функция возвращает 0; при ошибке возвращается номер ошибки.

Ошибки
ENOENT — Тактирование частоты процессора для потоков не поддерживается системой.
ESRCH — Не удалось найти поток с идентификатором **thread**.

Примечание
Если задан идентификатор часов **CLOCK_THREAD_CPUTIME_ID**, то когда поток ссылается на вызывающий поток, эта функция возвращает идентификатор, который относится к тем же часам, которыми манипулируют **clock_gettime(2)** и **clock_settime(2)**.

Пример

Приведенная ниже программа создает поток, а затем использует **clock_gettime(2)** для получения общего времени ЦП процесса и времени ЦП для каждого потока, потребляемого двумя потоками. В следующем сеансе оболочки показан пример выполнения:

```
$ ./a.out
Main thread sleeping
Субпоток входит в бесконечный цикл
Главный поток немного поспит
Главный поток проснулся и потребляет время CPU...
Process total CPU time:    2.575
Main thread CPU time:      0.789
Subthread CPU time: 1      1.786
```

Программный код

```
/* Link with "-lrt" */
#include <time.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>
#include <string.h>
#include <errno.h>

#define handle_error(msg) do {perror(msg); exit(EXIT_FAILURE); } while (0)

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

```
static void* thread_start(void *arg) {

    printf("Субпоток входит в бесконечный цикл\n");
    for (;;) continue;
    return NULL;
}

static void pclock(char *msg, clockid_t clock_id) {

    struct timespec ts; // секунды и наносекунды в ts.tv_sec, ts.tv_nsec

    printf("%s", msg);
    if (clock_gettime(clock_id, &ts) == -1)
        handle_error("clock_gettime()");
    printf("%4ld.%03ld\n", ts.tv_sec, ts.tv_nsec / 1000000);
}

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

    pthread_t thread;
    clockid_t cid;
    int j;

    int s = pthread_create(&thread,      // ID потока
                           NULL,         // атрибуты
                           thread_start, // адрес потоковой функции
                           NULL);        // аргументы для потока
    if (s != 0) {
        handle_error_en(s, "pthread_create");
```

```
    printf("Главный поток немного поспит...\n");
    sleep(1);
    printf("Главный поток проснулся и потребляет время CPU...\n");
    for (j = 0; j < 2000000; j++) {
        getppid();
    }
    // CLOCK_PROCESS_CPUTIME_ID -- настраиваемые для каждого процесса часы.
    // Измеряют время ЦП, затраченное всеми нитями процесса.
    pclock("Process total CPU time: ", CLOCK_PROCESS_CPUTIME_ID);

    s = pthread_getcpuclockid(pthread_self(), &cid);
    if (s != 0)
        handle_error_en(s, "pthread_getcpuclockid");
    pclock("Main thread CPU time:   ", cid);
    // pclock("Main thread CPU time:   ", CLOCK_THREAD_CPUTIME_ID); // эквивалентное

    s = pthread_getcpuclockid(thread, &cid);
    if (s != 0)
        handle_error_en(s, "pthread_getcpuclockid");
    pclock("Subthread CPU time: 1    ", cid);

    exit(EXIT_SUCCESS);         // Завершаем оба потока
}
```

pthread_sigqueue() — поставить сигнал и данные в очередь потока

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

int pthread_sigqueue(pthread_t thread,            // _GNU_SOURCE
                     int sig,
                     const union sigval value);
```

Функция **pthread_sigqueue()** выполняет задачу, аналогичную **sigqueue(3)**, но вместо того, чтобы отправлять сигнал процессу, она отправляет сигнал потоку в том же процессе, что и вызывающий поток.

Аргумент **thread** — это идентификатор потока в том же процессе, что и вызывающий.

Аргумент **sig** указывает сигнал, который нужно отправить.

Аргумент **value** указывает данные, сопровождающие сигнал.

Возвращаемое значение

В случае успеха **pthread_sigqueue()** возвращает 0. При ошибке возвращается номер ошибки.

Ошибки

EAGAIN — достигнут предел сигналов, которые могут быть поставлены в очередь.
EINVAL — **sig** недействителен
ENOSYS — **pthread_sigqueue()** не поддерживается в этой системе.
ESRCH — **thread** недействителен

Примечание

Реализация **pthread_sigqueue()** в glibc выдает ошибку **EINVAL** при попытках отправить любой из сигналов реального времени, используемых внутри реализации потоковой передачи NPTL.

pthread_atfork() — зарегистрировать обработчик вызова fork()

```
#include <pthread.h>

int pthread_atfork(void (*prepare)(void), // перед fork()
                   void (*parent)(void),  // в родительском процессе после fork()
                   void (*child)(void));  // в дочернем процессе после fork()
```

Функция **pthread_atfork()** регистрирует обработчики вызова **fork()**, которые должны выполняться, когда поток вызывает **fork()**.

Обработчики выполняются в контексте потока, вызывающего **fork()**.

Можно зарегистрировать три вида обработчиков:

- **prepare** указывает обработчик, который выполняется перед запуском работы **fork()**.
- **parent** указывает обработчик, который выполняется в родительском процессе после возврата **fork()**.
- **child** определяет обработчик, который выполняется в дочернем процессе после **возврата fork()**.

Любой из трех аргументов может иметь значение **NULL**, если на соответствующей фазе работы **fork()** обработчик не требуется.

Если вызов **fork()** в многопоточном процессе приводит к тому, что дочерний обработчик **fork()** вызывает любую функцию, которая не является безопасной для асинхронных сигналов, поведение не определено.

pthread_atfork() может вызываться потоком несколько раз, чтобы зарегистрировать несколько обработчиков для каждой фазы.

Порядок вызовов **pthread_atfork()** имеет значение. Обработчики родительский и дочерний будут вызываться в том порядке, в котором они были установлены вызовами **pthread_atfork()**.

Подготовительные обработчики будут вызываться в обратном порядке.

Возвращаемое значение

В случае успеха **pthread_atfork()** возвращает ноль. В случае ошибки номер ошибки.

Ошибки

ENOMEM — Не удалось выделить память для записи записи обработчика формы.

Примечания

Когда fork() вызывается в многопоточном процессе,
в дочернем процессе дублируется только вызывающий поток.

Первоначальное намерение **pthread_atfork()** состояло в том, чтобы позволить вызывающему потоку вернуться в согласованное состояние. Например, во время вызова **fork(2)** другие потоки могут иметь заблокированные мьютексы, которые видны в памяти пользовательского пространства, дублированной в дочернем процессе. Такие мьютексы никогда не будут разблокированы, поскольку потоки, установившие блокировки, не дублируются в дочернем процессе.

Назначение **pthread_atfork()** состояло в том, чтобы предоставить механизм, с помощью которого приложение (или библиотека) могло бы гарантировать, что мьютексы и другие состояния процесса и потока перед вызовом **fork()** будут восстановлены до согласованного состояния.

На практике эта задача, как правило, слишком сложна,
чтобы ее можно было выполнить,
соответственно, ее лучше не выполнять совсем.

После того, как вызванный в многопоточном процессе **fork(2)** возвращается в дочернем процессе, дочерний процесс должен вызывать только асинхронно-сигнально-безопасные функции.

Применение

Первоначальный шаблон использования, предусмотренный для **pthread_atfork()**, заключался в том, чтобы подготовительный обработчик **fork()** блокировал мьютексы и другие блокировки, а родительский и дочерний обработчики — их разблокировали. Однако, поскольку все соответствующие функции разблокировки, кроме **sem_post()**, не являются безопасными для асинхронных сигналов, это использование приводит к неопределенному поведению дочернего процесса, если только такая функция разблокировки, которую он вызывает, не является **sem_post()**.

Обоснование

В многопоточной программе есть, как минимум, две серьезные проблемы с семантикой **fork()**.

Одна проблема связана с состоянием (например, памятью), покрытым мьютексами.

Рассмотрим случай, когда один поток имеет заблокированный мьютекс и состояние, охватываемое этим мьютексом, несогласовано, в то время как другой поток вызывает **fork()**. В дочернем процессе мьютекс находится в заблокированном состоянии. Он заблокирован несуществующим потоком и, следовательно, никогда не может быть разблокирован. Если потомок просто повторно инициализирует мьютекс, это будет нехорошо, поскольку не решает вопроса о том, как исправить или иным образом справиться с несогласованным состоянием в потомке.

Предлагается, чтобы программы, использующие **fork()**, в дочернем процессе очень скоро после этого вызывали функцию **exec()**, таким образом сбрасывая все состояния.

К сожалению, это решение не удовлетворяет потребности многопоточных библиотек. Прикладные программы могут не знать, что используется многопоточная библиотека, и они могут свободно вызывать любое количество библиотечных подпрограмм между вызовами **fork()** и **exec()**.

В самом деле, они могут быть существующими однопоточными программами, и поэтому нельзя ожидать, что они будут подчиняться новым ограничениям, налагаемым библиотекой потоков.

С другой стороны, многопоточной библиотеке нужен способ защиты своего внутреннего состояния во время **fork()** на случай, если оно будет позже повторно использоваться в дочернем процессе.

Эта проблема возникает особенно в библиотеках многопоточного ввода-вывода, которые почти наверняка будут вызываться между вызовами **fork()** и **exec()** для выполнения перенаправления ввода-вывода. Решение может потребовать блокировки переменных мьютекса во время **fork()** или может повлечь за собой простой сброс состояния в дочернем элементе после завершения обработки **fork()**.

Функция **pthread_atfork()** была предназначена для обеспечения многопоточных библиотек средствами защиты от «*ничего не знающих*» прикладных программ, вызывающих **fork()**, и для предоставления многопоточных прикладных программ стандартным механизмом защиты от вызовов **fork()** в библиотечных процедурах или в самом приложении.

Ожидаемое использование заключалось в том, что обработчик этапа подготовки получит все блокировки мьютексов, а два других обработчика **fork()** освободят их.

Например, приложение могло бы предоставить подпрограмму этапа подготовки, которая получает необходимые мьютексы, поддерживаемые библиотекой, и предоставляет дочерние и родительские подпрограммы, которые освобождают эти мьютексы, тем самым гарантируя, что дочерний элемент получит согласованный моментальный снимок состояния библиотеки (и что мьютексы не остались бы без дела).

Теоретически это хорошо, но на практике — непрактично. Каждый мьютекс и блокировка в процессе должны быть обнаружены и заблокированы. Каждый компонент программы, включая сторонние компоненты, должен участвовать, и они должны согласовать, кто несет ответственность за какой мьютекс или блокировку. Это особенно проблематично для мьютексов и блокировок в динамически выделяемой памяти. Все внутренние мьютексы и блокировки также должны быть заблокированы. Это может задерживать вызов нити **fork()** на долгое время или даже на неопределенное время, поскольку использование этих объектов синхронизации может не контролироваться приложением.

Последняя проблема, о которой следует упомянуть, — это проблема блокировки потоков ввода/вывода.

По крайней мере, потоки, находящиеся под управлением системы (такие как **`stdin`**, **`stdout`**, **`stderr`**), должны быть защищены блокировкой потока с помощью **`flockfile()`**.

Но само приложение может это сделать в том же потоке, который вызывает **`fork()`**.

В этом случае процесс зайдет в тупик.

Спин-блокировки (спинлоки)

pthread_spin_init()/destroy() — инициализировать/уничтожить спинлок

```
#include <pthread.h>                          // _POSIX_C_SOURCE >= 200112L

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

Общее примечание

Большинство программ должны использовать мьютексы вместо спин-блокировок.

Спин-блокировки в первую очередь полезны в сочетании с политиками планирования в реальном времени.

Функция **pthread_spin_init()** выделяет любые ресурсы, необходимые для использования спин-блокировки, на которую ссылается **lock**, и инициализирует спинлок в разблокированном состоянии.

pshared должен иметь одно из следующих значений:

PTHREAD_PROCESS_PRIVATE — спин-блокировка должна использоваться только потоками в том же процессе, что и поток, вызывающий **pthread_spin_init()**.

Попытка совместно использовать спин-блокировку между процессами приводит к неопределенному поведению.

PTHREAD_PROCESS_SHARED — спин-блокировка может использоваться любым потоком в любом процессе, который имеет доступ к памяти, содержащей блокировку, т.е. спин-блокировка может находиться в объекте совместно используемой памяти, который совместно используется несколькими процессами.

¤ Вызов **pthread_spin_init()** для уже инициализированной спин-блокировки приводит к неопределенному поведению.

Функция **pthread_spin_destroy()** уничтожает ранее инициализированную спин-блокировку, освобождая все ресурсы, которые были выделены для нее.

¤ Разрушение спин-блокировки, которая не была ранее инициализирована, или разрушение спин-блокировки, в то время как другой поток удерживает блокировку, приводит к неопределенному поведению.

¤ После того, как спин-блокировка была уничтожена, выполнение любой операции с блокировкой, кроме повторной инициализации ее с помощью **pthread_spin_init()**, приводит к неопределенному поведению.

¤ Результат выполнения таких операций, как **pthread_spin_lock(3)**, **pthread_spin_unlock(3)** и **pthread_spin_destroy(3)** на копиях объекта, на который ссылается блокировка, не определен.

Возвращаемое значение

В случае успеха функции возвращают ноль. В случае неудачи они возвращают номер ошибки. В случае сбоя **pthread_spin_init()** блокировка не инициализируется.

Ошибки

EAGAIN — В системе недостаточно ресурсов для инициализации новой блокировки спина.
ENOMEM — Недостаточно памяти для инициализации спин-блокировки.

Поддержка спин-блокировок с совместным использованием несколькими процессами — это опция POSIX. В реализации glibc эта опция поддерживается

§ Примечание

Спиновые блокировки следует использовать в сочетании с политиками планирования в реальном времени (**SCHED_FIFO** или, возможно, **SCHED_RR**).

Использование спин-блокировок с недетерминированными политиками планирования, такими как **SCHED_OTHER**, вероятно, указывает на ошибку проектирования.

Проблема в том, что если поток, работающий в соответствии с такой политикой, запланирован в состоянии вне ЦП, в то время как он удерживает спин-блокировку, другие потоки будут тратить время на ожидание, пока владелец блокировки не будет еще раз перепланирован и не снимет блокировку.

Если потоки создают ситуацию взаимоблокировки при использовании спин-блокировок, эти потоки будут крутиться бесконечно, потребляя процессорное время.

Спин-блокировки в пользовательском пространстве неприменимы в качестве общего решения для блокировок. Они по определению склонны к инверсии приоритета и неограниченному времени ожидания. Программист, использующий спин-блокировки, должен быть чрезвычайно осторожен не только в коде, но и в отношении конфигурации системы, размещения потоков и назначения приоритетов.

pthread_spin_lock/trylock/unlock — заблокировать/разблокировать спинлок

```
#include <pthread.h>                        // _POSIX_C_SOURCE >= 200112L

int pthread_spin_lock(pthread_spinlock_t *lock);
int pthread_spin_trylock(pthread_spinlock_t *lock);
int pthread_spin_unlock(pthread_spinlock_t *lock);
```

Функция **pthread_spin_lock()** блокирует спин-блокировку, на которую ссылается **lock**.

Если спин-блокировка в настоящее время разблокирована, вызывающий поток получает блокировку немедленно.

Если спин-блокировка в настоящее время заблокирована другим потоком, вызывающий поток «вращается», проверяя блокировку, пока она не станет доступной, после чего вызывающий поток получает блокировку.

- Вызов **pthread_spin_lock()** для блокировки, которая уже удерживается вызывающей стороной, или блокировки, которая не была инициализирована с помощью **pthread_spin_init(3)**, приводит к неопределенному поведению.

Функция **pthread_spin_trylock()** похожа на **pthread_spin_lock()**, за исключением того, что если спин-блокировка, на которую ссылается **lock**, в настоящее время заблокирована, то вместо «вращения» в цикле ожидания вызов немедленно возвращается с ошибкой **EBUSY**.

Функция **pthread_spin_unlock()** разблокирует спин-блокировку **lock**.

Если какие-либо потоки «вращаются» на блокировке, один из этих потоков получит разблокированную блокировку.

- Вызов **pthread_spin_unlock()** для блокировки, которая не удерживается вызывающей стороной, приводит к неопределенному поведению.

Возвращаемое значение

В случае успеха эти функции возвращают ноль. В случае неудачи они возвращают номер ошибки.

Ошибки

`pthread_spin_lock()` может завершиться ошибкой со следующими ошибками:
EDEADLOCK — Система обнаружила состояние взаимоблокировки.

`pthread_spin_trylock()` выдает следующие ошибки:
EBUSY — Спин-блокировка в настоящее время заблокирована другим потоком.

Примечание

¤ Применение любой из функций, описанных выше, к неинициализированной спин-блокировке приводит к неопределенному поведению.

RW-блокировки

Во многих ситуациях данные читаются чаще, чем изменяются или записываются.

В этих случаях можно, удерживая блокировку, разрешить потокам читать одновременно, и разрешить только одному потоку удерживать блокировку для изменения данных.

Таким поведением обладает блокировка с несколькими «читателями» и одним «писателем» (или блокировка чтения/записи).

Блокировка чтения-записи устанавливается либо для чтения, либо для записи, а затем снимается.

Поток, который получает блокировку чтения-записи, должен быть тем, который ее снимает.

Объект атрибутов блокировки чтения/записи

Вызов **pthread_rwlockattr_init()** инициализирует объект атрибутов блокировки чтения-записи (**attr**). Значение по умолчанию для всех атрибутов определяется реализацией.

Если подпрограмма **pthread_rwlockattr_init()** указывает уже инициализированный объект атрибутов блокировки чтения-записи, могут возникнуть неожиданные результаты.

В следующих примерах показано, как вызвать **pthread_rwlockattr_init()** с объектом **attr**:

```
pthread_rwlockattr_t attr;

pthread_rwlockattr_init(&attr);
```

После того, как объект атрибутов блокировки чтения/записи был использован для инициализации одной или нескольких блокировок чтения/записи, любая функция, влияющая на объект атрибутов (включая разрушение), не влияет ни на какие ранее инициализированные блокировки чтения/записи.

Вызов **pthread_rwlockattr_destroy()** уничтожает объект атрибутов блокировки чтения/записи.

¤ Если объект атрибутов используется до того, как он будет повторно инициализирован другим вызовом подпрограммы **pthread_rwlockattr_init()**, результаты не определены.

Создание и разрушение блокировок чтения-записи

Прежде чем можно пользоваться блокировкой **rwlock**, она должна быть проинициализирована.

Для инициализации используется вызов **rwlock_init()**, который инициализирует блокировку чтения-записи, на которую ссылается объект **rwlock**, с атрибутами, на которые ссылается объект **attr**.

Как всегда, если объект **attr** имеет значение **NULL**, используются атрибуты блокировки чтения-записи по умолчанию.

После успешной инициализации состояние блокировки чтения-записи становится инициализированной и разблокированной.

RW-блокировку можно использовать любое количество раз без повторной инициализации.

Вызов **rwlock_destroy()** уничтожает (делает недоступным) объект блокировки чтения-записи, на который ссылается объект **rwlock**, и освобождает все ресурсы, используемые блокировкой.

Уничтоженный объект блокировки чтения-записи может быть повторно инициализирован с помощью вызова **pthread_rwlock_init()**.

Блокировка объекта RWLOCK для чтения (RDLOCK)

Вызов `rwlock_rdlock()` выполняет блокировку чтения объекта, на который ссылается **`rwlock`**.

1) Вызывающий поток получает блокировку чтения, если эту блокировку не удерживает «писатель» и нет ни одного «писателя», заблокированного в ее ожидании.

2) Получит ли вызывающий поток блокировку, когда «писатель» не удерживает блокировку, но есть «писатели», ее ожидающие, не указано — это прерогатива реализации;

3) Если блокировку удерживает «писатель», вызывающий поток блокировку чтения не получит.

Если блокировка чтения не установлена сразу, вызывающий поток блокируется (не возвращается из вызова **`rwlock_rdlock()`**, пока не получит блокировку).

Поток может удерживать несколько одновременных блокировок чтения на объекте **`rwlock`**, то есть успешно вызывать **`rwlock_rdlock()`** несколько раз.

Если это так, поток должен выполнить соответствующие разблокировки, то есть он должен вызвать **`rwlock_unlock()`** столько раз, сколько раз он вызывал **`rwlock_rdlock()`**.

Вызов **`rwlock_tryrdlock()`** выполняет блокировку чтения аналогично вызову **`rwlock_rdlock()`**, за исключением того, что вызов сразу возвращается с ошибкой, если какой-либо поток удерживает блокировку записи на объекте **`rwlock`** или есть «писатели», заблокированные на объекте **`rwlock`**.

Блокировка объекта RW-блокировки для записи

Вызов **`rwlock_wrlock()`** выполняет блокировку записи объекта, на который ссылается **`rwlock`**.

Вызывающий поток получает блокировку записи, если никакой другой поток («читатель» или «писатель») эту блокировку не удерживает, в том числе и вызывающий. В противном случае поток будет заблокирован (не вернется из вызова **`rwlock_wrlock()`**, пока не получит блокировку).

Вызов **`rwlock_trywrlock()`** выполняет блокировку чтения аналогично вызову **`rwlock_wrlock()`**, за исключением того, что вызов сразу возвращается с ошибкой, если какой-либо поток в настоящее удерживает **`rwlock`** для чтения или записи.

rwlock_init()/destroy() — инициализировать и уничтожить объект RW-блокировки

```
 #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);

pthread_rwlock_t rwlock = PTHREAD_RWLOCK_INITIALIZER; // если атрибуты по умолчанию
```

Вызов **pthread_rwlock_init()** выделяет ресурсы, необходимые для использования блокировки чтения-записи, на которую ссылается **rwlock**, и инициализирует блокировку в разблокированном состоянии с атрибутами, на которые ссылается **attr**.

Если **attr** равен **NULL**, используются атрибуты блокировки чтения-записи по умолчанию.

После инициализации блокировку можно использовать любое количество раз без повторной инициализации.

В случаях, когда подходят атрибуты блокировки чтения-записи по умолчанию, для инициализации блокировок чтения-записи можно использовать макрос **PTHREAD_RWLOCK_INITIALIZER**.

Эффект эквивалентен динамической инициализации вызовом **pthread_rwlock_init()** с параметром **attr**, заданным как **NULL**, за исключением того, что не выполняются проверки на ошибки.

¤ Если блокировка чтения-записи используется без предварительной инициализации, результаты не определены.

Если вызов **pthread_rwlock_init()** завершается неудачно, **rwlock** не инициализируется, а содержимое **rwlock** будет не определено.

¤ Если вызывается **pthread_rwlock_init()** с указанием уже инициализированной блокировки чтения-записи, результаты не определены.

¤ Если значение, указанное аргументом **attr** для **pthread_rwlock_init()**, не относится к инициализированному объекту атрибутов блокировки чтения-записи, поведение не определено.

Вызов **pthread_rwlock_destroy()** уничтожает объект блокировки чтения-записи, на который ссылается **rwlock**, и освобождает все ресурсы, используемые этой блокировкой.

¤ Если **pthread_rwlock_destroy()** вызывается, когда какой-либо поток удерживает **rwlock**, результаты не определены.

¤ Попытка уничтожить неинициализированную блокировку чтения-записи приводит к неопределенному поведению.

Возращаемое значение

В случае успеха функции **pthread_rwlock_destroy()** и **pthread_rwlock_init()** возвращают ноль; в противном случае номер ошибки.

Ошибки

Вызов **pthread_rwlock_init()** завершается ошибкой:

EAGAIN — в системе не хватает необходимых ресурсов (кроме памяти) для инициализации еще одной блокировки чтения-записи.

ENOMEM — недостаточно памяти для инициализации блокировки чтения-записи.

EPERM — вызывающий не имеет права выполнять операцию.

Функции **pthread_rwlock_destroy()** и **pthread_rwlock_init()** *могут завершиться* ошибкой:

EINVAL — значение, указанное аргументом **rwlock**, не относится к инициализированному объекту блокировки чтения-записи.

EBUSY — значение, указанное аргументом **rwlock**, относится к заблокированному объекту блокировки чтения-записи

EBUSY — значение, указанное аргументом **rwlock** для **pthread_rwlock_init()**, относится к уже инициализированному объекту блокировки чтения-записи.

rwlock_rdlock()/tryrdlock() — заблокировать объект RW-блокировки для чтения

```
#include <pthread.h>

int pthread_rwlock_rdlock(pthread_rwlock_t *rwlock);
int pthread_rwlock_tryrdlock(pthread_rwlock_t *rwlock);
```

Вызов **pthread_rwlock_rdlock()** пытается применить блокировку чтения к блокировке, на которую ссылается **rwlock**.

Вызывающий поток получает блокировку чтения, если «писатель» не удерживает блокировку и нет ни одного «писателя», заблокированного на **rwlock**.

Если поддерживается опция планирования выполнения потоков, и потоки, задействованные в блокировке, выполняются с политиками планирования **SCHED_FIFO** или **SCHED_RR**, вызывающий поток не получит блокировку, если блокировку удерживает «писатель» или если «писатели» с более высоким или равным приоритетом заблокированы в ожидании захвата блокировки.

В противном случае вызывающий поток получает блокировку.

Аналогичное имеет место и для политики **SCHED_SPORADIC.**

Если параметр планирования выполнения потоков не поддерживается, получает ли вызывающий поток блокировку, когда нет «писателей», удерживающих блокировку, но есть «писатели», заблокированные в ожидании, определяется реализацией.

Если же «писатель» удерживает блокировку, вызывающий поток блокировку чтения не получит.

Если блокировка чтения не получена сразу, вызывающий поток блокируется, пока не получит блокировку.

Вызывающий поток может заблокироваться, если во время вызова он сам удерживает блокировку записи.

Поток может удерживать несколько одновременных блокировок чтения на **rwlock**, то есть успешно вызывать функцию **pthread_rwlock_rdlock()** несколько раз. Если это так, приложение должно гарантировать, что поток выполняет соответствующее количество разблокировок, то есть он вызывает функцию **pthread_rwlock_unlock()** соответствующее количество раз.

Максимальное количество одновременных блокировок чтения, которое, как гарантирует реализация, может быть применено к блокировке чтения-записи, определяется реализацией.

Вызов **pthread_rwlock_rdlock()** может завершиться неудачей, если этот максимум будет превышен.

Вызов **pthread_rwlock_tryrdlock()** ведет себя аналогично вызову **pthread_rwlock_rdlock()**, за исключением того, что **pthread_rwlock_tryrdlock()** завершится ошибкой в том случае, когда эквивалентный вызов **pthread_rwlock_rdlock()** заблокировал бы вызывающий поток.

Вызов **pthread_rwlock_tryrdlock()** никогда не блокируется — он всегда либо получает блокировку, либо завершается неудачно и немедленно возвращается.

Если какая-либо из этих функций вызывается с неинициализированной блокировкой чтения-записи, результат не определен.

Если потоку, ожидающему блокировки чтения-записи для чтения, доставляется сигнал, после возврата из обработчика сигнала поток возобновляет ожидание блокировки, как если бы оно не было прервано.

Возращаемое значение

В случае успеха вызов **pthread_rwlock_rdlock()** вернет ноль, в противном случае номер ошибки.

Вызов **pthread_rwlock_tryrdlock()** возвращает ноль, если установлена блокировка чтения для объекта, на который ссылается **rwlock**.

В противном случае возвращается номер ошибки, указывающий на ошибку.

Ошибки

Вызов **pthread_rwlock_tryrdlock()** завершается с ошибкой:

EBUSY — Блокировка чтения-записи не может быть получена для чтения, потому что ее удерживает «писатель» или «писатель» с соответствующим приоритетом на ней заблокирован.

Функции **pthread_rwlock_rdlock()** и **pthread_rwlock_tryrdlock()** могут завершиться с ошибкой, если:

EAGAIN — Невозможно получить блокировку чтения, поскольку превышено максимальное количество блокировок чтения для **rwlock**.

EINVAL — Значение, указанное аргументом **rwlock**, не относится к инициализированному объекту блокировки чтения-записи.

Функция **pthread_rwlock_rdlock()** может завершиться с ошибкой, если:

EDEADLK — Обнаружено состояние взаимоблокировки или текущий поток уже владеет блокировкой чтения-записи для записи.

rwlock_timedrdlock() — заблокировать объект RW-блокировки для чтения

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

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

Вызов **pthread_rwlock_timedrdlock()** пытается заблокировать по чтению rw-блокировку, на которую ссылается **rwlock**, аналогично тому, как это делает вызов **pthread_rwlock_rdlock()**.

Однако, если блокировка не может быть получена в течение некоторого времени, из-за того, что другие потоки ее блокируют, ожидание прекращается по истечении указанного тайм-аута.

Тайм-аут истекает, когда проходит абсолютное время, указанное в **abstime**, то есть, когда значение часов, с помощью которых измеряется таймаут, равно или превышает **abstime**, или если абсолютное время, указанное в **abstime**, на момент вызова уже было прошло.

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

Разрешение тайм-аута должно быть разрешением часов **CLOCK_REALTIME**.

Тип данных **timepec** определяется в заголовке **<time.h>**.

struct timespec – содержит интервал, указанный в секундах и наносекундах, который может представлять календарное время, основанное на конкретной эпохе.

Структура **timespec** содержит, как минимум, следующие члены в любом порядке.

```
time_t tv_sec;  // целое число секунд – ≥ 0
long   tv_nsec; // наносекунды – [0, 999999999]
```

Элемент **tv_sec** является линейным счетчиком секунд. и может не иметь обычной семантики **time_t**. Семантика членов и их нормальные диапазоны выражены в комментариях.

В Linux структура **timespec** определена так:

```
#ifndef _STRUCT_TIMESPEC
#define _STRUCT_TIMESPEC
struct timespec {
    __kernel_old_time_t tv_sec;  // seconds
    long                tv_nsec; // nanoseconds
};
#endif
```

Если блокировка может быть получена немедленно, вызов **pthread_rwlock_timedrdlock()** никогда не завершается с ошибкой по тайм-ауту.

Если блокировку можно получить немедленно, достоверность параметра **abstime** не проверяется.

Если потоку, ожидающему блокировку чтения-записи для чтения, доставляется сигнал, после возврата из обработчика сигнала поток возобновляет ожидание блокировки, как если бы оно не было прервано.

- Вызывающий поток может зайти в тупик, если во время вызова он удерживает блокировку записи на **rwlock**.
- Если эта вызов вызывается с неинициализированной блокировкой чтения-записи, результат не определен.

Возращаемое значение

Вызов **pthread_rwlock_timedrdlock()** возвращает ноль, если на объекте блокировки чтения-записи, на который ссылается **rwlock**, блокировка для чтения получена.

В противном случае возвращется номер ошибки.

Ошибки

Вызов **pthread_rwlock_timedrdlock()** завершается ошибкой:

ETIMEDOUT — если не удалось получить блокировку до истечения указанного тайм-аута.

Вызов **pthread_rwlock_timedrdlock()** может завершиться ошибкой:

EAGAIN — если невозможно получить блокировку чтения, поскольку будет превышено максимальное количество блокировок чтения для блокировки.

EDEADLK — если обнаружено состояние взаимоблокировки или вызывающий поток уже удерживает на **rwlock** блокировку записи.

EINVAL — если значение наносекундной части **abstime** меньше нуля или больше или равно 1000 миллионов.

EINVAL — если значение, указанное аргументом **rwlock** для **pthread_rwlock_timedrdlock()**, не относится к инициализированному объекту блокировки.

rwlock_wrlock()/trywrlock() — заблокировать объект RW-блокировки для записи

```
#include <pthread.h>

int pthread_rwlock_trywrlock(pthread_rwlock_t *rwlock);
int pthread_rwlock_wrlock(pthread_rwlock_t    *rwlock);
```

Вызов **pthread_rwlock_wrlock()** применяет блокировку записи к блокировке чтения-записи, на которую ссылается **rwlock**.

Вызывающий поток получает блокировку записи в том случае, если ни один поток, не имеет значения, «читатель» это или «писатель», не удерживает блокировку чтения-записи на **rwlock**.

В противном случае, если другой поток удерживает блокировку чтения-записи на **rwlock**, вызывающий поток блокируется до тех пор, пока не получит блокировку.

Если возникает состояние взаимоблокировки или вызывающий поток уже владеет блокировкой чтения-записи для записи или чтения, вызов может быть заблокирован, либо может быть возвращен код ошибки **EDEADLK**.

Вызов **pthread_rwlock_trywrlock()** применяет блокировку записи так же, как и вызов **pthread_rwlock_wrlock()**, за исключением того, что вызов завершится ошибкой, если какой-либо поток в настоящее время удерживает **rwlock** для чтения или для записи.

- Если вызов применяется к неинициализированной блокировке, результаты не определены.

Если потоку, ожидающему блокировку чтения-записи для чтения, доставляется сигнал, после возврата из обработчика сигнала поток возобновляет ожидание блокировки, как если бы оно не было прервано.

Возвращаемое значение

В случае успешного завершения вызов **pthread_rwlock_wrlock()** вернет ноль, в противном случае возвращается номер ошибки.

Вызов **pthread_rwlock_trywrlock()** возвращает ноль, если блокировка для записи в объекте, на который ссылается **rwlock**, установлена. В противном случае возвращается номер ошибки.

Ошибки

Вызов **pthread_rwlock_trywrlock()** завершится ошибкой:

EBUSY — если блокировка чтения-записи для записи не может быть получена из-за того, что она уже заблокирована для чтения или записи.

Вызов **pthread_rwlock_wrlock()** может завершиться ошибкой:

EDEADLK — если обнаружено состояние взаимоблокировки, или текущий поток уже владеет блокировкой чтения-записи для записи или чтения.

Функции могут завершиться с ошибкой:

EINVAL — если значение, указанное аргументом **rwlock для pthread_rwlock_trywrlock()** или **pthread_rwlock_wrlock()**, не относится к инициализированному объекту блокировки чтения-записи.

rwlock_timedwrlock() — заблокировать объект RW-блокировки для записи

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

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

Вызов **pthread_rwlock_timedwrlock()** пытается получить блокировку записи к rw-блокировке, на которую ссылается **rwlock**, аналогично тому, как это делает вызов **pthread_rwlock_wrlock()**. Однако, если блокировка не может быть получена из-за того, что другие потоки ее удерживают, ожидание прекращается по истечении указанного тайм-аута.

Тайм-аут истекает, когда проходит абсолютное время, указанное в **abstime**, то есть, когда значение часов, с помощью которых измеряется таймаут, равно или превышает **abstime**, или если абсолютное время, указанное в **abstime**, на момент вызова уже было прошло.

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

Разрешение тайм-аута должно быть разрешением часов **CLOCK_REALTIME**.

Тип данных **timepec** определяется в заголовке **<time.h>**.

Если блокировка может быть получена немедленно, вызов ни при каких обстоятельствах не завершается с ошибкой по таймауту.

Достоверность параметра **abstime** не проверяется, если блокировку можно получить немедленно.

Если потоку, ожидающему блокировку чтения-записи для записи, доставляется сигнал, после возврата из обработчика сигнала поток возобновляет ожидание блокировки, как если бы оно не было прервано.

¤ Вызывающий поток может зайти в тупик, если во время вызова он удерживает блокировку записи на **rwlock**.

¤ Если эта вызов вызывается с неинициализированной блокировкой чтения-записи, результат не определен.

Возвращаемое значение

Вызов **pthread_rwlock_timedwrlock()** возвращает ноль, если блокировка для записи в объекте, на который ссылается **rwlock**, получена.

В противном случае должен возвращается номер ошибки.

Ошибки

Вызов **pthread_rwlock_timedwrlock()** завершится ошибкой:

ETIMEDOUT — если не удалось получить блокировку до истечения указанного тайм-аута.

Вызов **pthread_rwlock_timedwrlock()** может завершиться ошибкой:

EAGAIN — если невозможно получить блокировку чтения, поскольку будет превышено максимальное количество блокировок чтения для блокировки.

EDEADLK — если обнаружено состояние взаимоблокировки или вызывающий поток уже удерживает на **rwlock** блокировку.

EINVAL — если значение наносекундной части **abstime** меньше нуля или больше или равно 1000 миллионов.

EINVAL — если значение, указанное аргументом **rwlock** для **pthread_rwlock_timedwrlock()**, не относится к инициализированному объекту блокировки.

rwlockattr_init()/destroy() — инициализировать и уничтожить объект атрибутов RW-блокировки

```
#include <pthread.h>

int pthread_rwlockattr_destroy(pthread_rwlockattr_t *attr);
int pthread_rwlockattr_init(pthread_rwlockattr_t *attr);
```

Вызов **pthread_rwlockattr_init()** инициализирует объект атрибутов блокировки чтения-записи **attr** значением по умолчанию для всех атрибутов, определенных реализацией.

- Если **pthread_rwlockattr_init()** вызывается с указанием уже инициализированного объекта атрибутов **attr**, результаты не определены.

Вызов **pthread_rwlockattr_destroy()** уничтожает объект атрибутов блокировки чтения-записи. Уничтоженный объект атрибутов **attr** можно повторно инициализировать с помощью **pthread_rwlockattr_init()**.

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

Реализация может вместо разрушения объекта атрибутов установить для объекта, на который ссылается **attr**, недопустимое значение.

После того, как объект атрибутов rw-блокировки был использован для инициализации одной или нескольких блокировок чтения-записи, любая функция, влияющая на объект атрибутов (включая разрушение), не влияет на какие-либо ранее инициализированные блокировки чтения-записи.

¤ Если значение, указанное аргументом **attr** для **pthread_rwlockattr_destroy()**, не относится к инициализированному объекту атрибутов блокировки чтения-записи, поведение не определено.

Возвращаемое значение

В случае успеха функции **pthread_rwlockattr_destroy()** и **pthread_rwlockattr_init()** возвращают ноль, в противном случае номер ошибки.

Ошибки

Вызов **pthread_rwlockattr_init()** завершается ошибкой:

ENOMEM — если недостаточно памяти для инициализации объекта атрибутов блокировки чтения-записи.

Вызов **pthread_rwlockattr_init()** может завершиться ошибкой:

EINVAL — если значение, указанное аргументом **attr**, не относится к инициализированному объекту атрибутов блокировки.

rwlockattr_{get|set}pshared — получить и установить атрибут «общий для процессов»

```
#include <pthread.h>

int pthread_rwlockattr_getpshared(const pthread_rwlockattr_t *restrict attr,
                                  int *restrict pshared);

int pthread_rwlockattr_setpshared(pthread_rwlockattr_t *attr,
                                  int pshared);
```

Вызов **pthread_rwlockattr_getpshared()** получает значение атрибута общего доступа к блокировке из инициализированного объекта атрибутов, на который ссылается **attr**.

Вызов **pthread_rwlockattr_setpshared()** установливает атрибут общего доступа в инициализированном объекте атрибутов, на который ссылается **attr**.

Атрибут имеет два значения:

PTHREAD_PROCESS_PRIVATE — совместное использование разными процессами запрещено.

PTHREAD_PROCESS_SHARED — совместное использование разными процессами разрешено.

Значение по умолчанию атрибута совместного использования разными процессами равно **PTHREAD_PROCESS_PRIVATE**.

Чтобы разрешить блокировке чтения-записи работать с любым потоком, имеющим доступ к памяти, где блокировка выделена, даже если она выделена в памяти, которая совместно используется несколькими процессами, атрибут общего доступа к процессу должен быть установлен в **PTHREAD_PROCESS_SHARED**.

Дополнительные атрибуты, их значения по умолчанию и имена связанных с ними функций получения и установки значений этих атрибутов определяются реализацией.

¤ Если значение, указанное аргументом **attr** для **pthread_rwlockattr_getpshared()** или **pthread_rwlockattr_setpshared()**, не относится к инициализированному объекту атрибутов, поведение не определено.

Возращаемое значение

После успешного завершения вызов **pthread_rwlockattr_getpshared()** возвращает ноль и сохраняет значение атрибута **attr** в объекте, на который ссылается параметр **pshared**. В противном случае должен возвращается номер ошибки.

В случае успеха вызов **pthread_rwlockattr_setpshared()** возвращает ноль, в противном случае возвращается номер ошибки.

Ошибки

Вызов **pthread_rwlockattr_setpshared()** может завершиться ошибкой:

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

Планирование процессов (Process Scheduling) (самостоятельно)

Политики планирования (Scheduling Policies)

Семантика планирования POSIX имеет отношение к реальному времени и оперирует наборами списков потоков. Никаких структур и способов реализации стандарт не определяет.

За реализацию в соответствии со стандартом отвечает ОС и делает это в соответствии с возможностями, которые предоставляет аппаратная платформа.

В модели планирования процессов POSIX обсуждается только планирование процессора для готовых к выполнению потоков (runnable). При этом отмечается, что если политика планирования процессора учитывается при упорядочении других ресурсов, предсказуемость приложений в реальном времени значительно улучшается.

Концептуально для каждого приоритета существует один список потоков.

Выполняемый поток будет в списке потоков, соответствующем приоритету данного потока.

Предусмотрено несколько политик планирования.

Каждый непустой список потоков упорядочен, содержит голову (head) на одном конце и хвост (tail) на другом.

Цель политики планирования — определить допустимые операции с этим набором списков, например, такие, как перемещение потоков между списками и внутри них.

Модель POSIX рассматривает «процесс» как совокупность системных ресурсов, включая один или несколько потоков, которые могут быть запланированы операционной системой для выполнения на процессоре(ах), которым(и) она управляет.

При этом, хотя процесс имеет свой собственный набор атрибутов планирования, на поведение планирования отдельных потоков, они имеют косвенное влияние, если вообще это имеет место.

Каждый поток управляется соответствующей политикой планирования и приоритетом.

Эти параметры могут быть указаны путем явного выполнения приложением функции **`pthread_setschedparam(3)`** — установить параметры политики планирования — алгоритм и параметры планирования для потоков.

Параметры планирования потока (но не его политика планирования) могут быть изменены путем выполнения приложением функции **`pthread_setschedprio(3)`** — установить параметры приоритезации для потока.

Параметры, имеющие отношение к политике планирования, могут быть указаны путем явного выполнения приложением функций

`sched_setscheduler()` — установить алгоритм и параметры планирования для процесса;

`sched_setparam()` — установить параметры планирования для процесса.

Существует две области конкуренции планирования — системная и область планирования процесса. Влияние атрибутов планирования процесса на отдельные потоки в процессе зависит от области конкуренции планирования потоков.

С каждой политикой связан некоторый диапазон приоритетов. Определение некоторой политики определяет для нее минимальный диапазон приоритетов. Диапазоны приоритетов для каждой политики могут, но не обязательно, перекрывать диапазоны приоритетов других политик.

Чтобы поток стал выполняющимся, реализация выбирает поток, который находится в голове непустого списка потоков с наивысшим приоритетом. После этого выбранный поток удаляется из своего списка потоков.

В частности, существует четыре стандартных политики планирования.
Могут быть определены другие политики планирования, определяемые реализацией.
В заголовке **<sched.h>** определены следующие символы:

SCHED_FIFO	политика планирования в порядке очереди (FIFO);
SCHED_RR	политика планирования циклического перебора (Round robin);
SCHED_SPORADIC	политика планирования спорадического сервера;
SCHED_OTHER	другая политика планирования.

SCHED_FIFO

Потоки, запланированные в соответствии с этой политикой, выбираются из списка потоков, который упорядочен по времени того, сколько они находятся в списке, но не выполняются.

Как правило, в голове списка поток, который был в списке дольше всех, а в хвосте — который был в списке самое короткое время.

В соответствии с политикой **SCHED_FIFO**, изменение определяющих списков потоков выглядит следующим образом:

1. Когда выполняющийся поток вытесняется (preempted), он становится в голову списка потоков своего приоритета.

2. Когда заблокированный поток становится готовым к исполнению потоком, он становится в хвост списка потоков своего приоритета.

3. Когда выполняющийся поток вызывает функцию **sched_setscheduler()**, процесс, указанный в вызове функции, изменяется в соответствии с указанной политикой и приоритетом, указанным в аргументе **param**.

4. Когда выполняющийся поток вызывает функцию **sched_setparam()**, приоритет процесса, указанного в вызове функции, изменяется до приоритета, указанного в аргументе **param**.

5. Когда выполняющийся поток вызывает функцию **pthread_setschedparam()**, поток, указанный в вызове функции, изменяется на указанную политику и приоритет, указанный аргументом **param**.

6. Когда выполняющийся поток вызывает функцию **pthread_setschedprio()**, поток, указанный в вызове функции, изменяется до приоритета, указанного аргументом **prio**.

7. Если поток, политика или приоритет которого были изменены не с помощью **pthread_setschedprio()**, является выполняющимся потоком или может быть запущен на выполнение, тогда он становится хвостом списка потоков для своего нового приоритета.

8. Если поток, приоритет которого был изменен с помощью **pthread_setschedprio()**, является выполняющимся потоком или может быть запущен на выполнение, влияние на его позицию в списке потоков зависит от направления *модификации*, как показано ниже:

   a. если приоритет повышен, поток становится хвостом списка потоков;

   б. если приоритет не изменяется, поток не меняет позицию в списке потоков;

   с. если приоритет понижен, поток становится головой соответсвующего списка потоков.

9. Когда запущенный поток вызывает функцию **sched_yield()**, поток становится хвостом списка потоков по своему приоритету.

Допустимые пределы приоритетов для политики можно получить, используя функции **sched_get_priority_max()** и **sched_get_priority_min()**, предоставляя в качестве параметра **SCHED_FIFO**.

Минимальный диапазон приоритетов **SCHED_FIFO** содержит не менее 32 приоритетов.

SCHED_RR

Политика планирования с циклическим перебором (round-robbin — карусель).

Эта политика идентична политике **SCHED_FIFO** с дополнительным условием, что, когда обнаруживается, что исполняющийся поток выполнялся как работающий поток в течение периода времени, равного длительности, возвращаемой функцией **sched_rr_get_interval()** или дольше, этот поток становится в конец своего списка потоков, голова этого списка потоков делается выполняющимся потоком и удаляется из списка.

Результатом политики **SCHED_RR** является гарантия того, чтобы при наличии нескольких потоков с одинаковым приоритетом, какая-нибудь из них не монополизирует процессор.

Поток, который вытесняется и впоследствии возобновляет выполнение в соответствии с данной политикой, завершает неистекшую часть своего временного интервала, выделенного в рамках стратегии планирования **RR**.

Приоритетный диапазон — не менее 32 приоритетов.

SCHED_SPORADIC

Политика спорадического обслуживания основана, в основном, на двух временных параметрах:
- период пополнения[1] (replenishment period);
- доступный объем исполнения (execution capacity).

Политика спорадического обслуживания идентична политике **SCHED_FIFO** с некоторыми дополнительными условиями, которые вызывают переключение назначенного приоритета потока между значениями, задаваемыми элементами **sched_priority** и **sched_ss_low_priority** в структуре **sched_param**, в зависимости от вышеназванных временных параметров.

SCHED_OTHER

Соответствующие стандарту реализации включают еще одну политику планирования, идентифицируемую как **SCHED_OTHER** (которая может работать идентично политике планирования **FIFO** или RR).

Планирование потоков с политикой **SCHED_OTHER** в системе, в которой другие потоки выполняются под **SCHED_FIFO**, **SCHED_RR** или **SCHED_SPORADIC**, определяется реализацией.

Эта политика предоставлена, чтобы позволить строго соответствующим стандарту приложениям иметь возможность переносимым образом указывать, что им больше не нужна политика планирования в реальном времени.

---

1) replenishment — пополнение, повторное наполнение

Работа со временем

Реальное время и время процесса

Реальное время (real time) — время, измеряемое от некоторой постоянной точки, или от стандартной точки в прошлом (Epoch, календарное время), или от некоторой точки в жизни процесса, например, с момента его запуска (прошедшее время (elapsed time)).

Время процесса (process time) — количество процессорного времени, использованного процессом. Иногда его делят на пользовательское (user) и системное (system). Пользовательское время ЦП — это время, потраченное на исполнение кода в режиме пользователя. Системное время ЦП — это время, потраченное ядром, выполняющемся в системном режиме, для процесса (например, на обработку системных вызовов).

Команда **`time(1)`** позволяет определить количество процессорного времени, затраченного при выполнении программы. Программа может определить количество потраченного процессорного времени с помощью функций **`times(2)`**, **`getrusage(2)`** или **`clock(3)`**.

Аппаратные часы

В большинстве компьютеров, оснащённых батареей, имеются аппаратные часы, которые ядро читает при запуске для инициализации программных часов (**`rtc(4)`** и **`hwclock(8)`**).

Программные часы, HZ и миги (jiffies)

Точность различных системных вызовов, которые задают время ожидания (timeouts), например, **`select(2)`**, **`sigtimedwait(2)`**, и измеряют процессорное время, например, **`getrusage(2)`**, ограничена точностью программных часов (software clock) — часов, поддерживаемых ядром, у которых время измеряется в мигах (**`jiffies`**). Размер мига определяется значением константы ядра **`HZ`**.

Значение **`HZ`** различно в разных версиях ядра и аппаратных платформах. Для i386 — в ядрах до версии 2.4.x включительно, **`HZ`** равно 100, то есть значение мига равно 0.01 секунды. Начиная с ядра версии 2.6.0 значение **`HZ`** увеличено до 1000 и миг равен 0.001 секунды. Начиная с ядра 2.6.13 значение **`HZ`** задаётся в параметре настройки ядра и может быть равно 100, 250 (по умолчанию) или 1000, что делает значение мига равным, соответственно, 0.01, 0.004 или 0.001 секунды.

Начиная с ядра 2.6.20 добавлено ещё одна частота — 300, количество, которое делится нацело на распространённые частоты видеокадров (PAL — 25 HZ; NTSC — 30 HZ).

Системный вызов **times(2)** — это особый случай. Он выдаёт время с точностью, определяемой константой ядра **USER_HZ**. Приложения пользовательского пространства могут определить значение этой константы с помощью **sysconf(_SC_CLK_TCK)**.

Таймеры высокой точности

До Linux 2.6.21 точность системных вызовов таймера и сна (смотрите далее) была ограничена размером мига.

Начиная с Linux 2.6.21, Linux поддерживает таймеры высокой точности (**HRT**), включаемые через **CONFIG_HIGH_RES_TIMERS**. В системе, которая поддерживает HRT, точность сна и таймеров в системных вызовах больше не ограничена мигом, а только точностью аппаратуры (в современной аппаратуре, обычно, микросекундная точность). Определить, поддерживаются ли таймеры высокой точности, можно проверив результат вызова **clock_getres(2)** или поискав записи «**resolution**» в **/proc/timer_list** (владелец root).

HRT поддерживаются не на всех аппаратных архитектурах (среди имеющих отметим x86, arm и powerpc).

Эпоха

В системах UNIX время считается в секундах и начинается с эпохи (Epoch), 1970-01-01 00:00:00 +0000 (UTC).

Программа может определить календарное время с помощью часов **CLOCK_REALTIME** через **clock_gettime(2)**, которые возвращают время (в секундах и наносекундах), прошедшее с начала эпохи.

Вызов **time(2)** выдаёт подобную информацию, но с точностью только до ближайшей секунды. Системное время можно изменять с помощью **clock_settime(2)**.

Календарное время, разделённое на компоненты

Некоторые библиотечные функции используют структуру с типом **struct tm** для представления календарного времени, разделённого на компоненты (broken-down time), в которой время хранится в виде отдельных составляющих (год, месяц, день, час, минута, секунда и т. д.). Эта структура описана в **ctime(3)**, в которой также описаны функции, преобразующие календарное время в разделённое на компоненты и обратно.

Функции представления календарного времени, разделённого на компоненты, в виде печатной строки описаны в **ctime(3)**, **strftime(3)** и **strptime(3)**.

Таймеры сна и их установка

Различные системные вызовы и функции позволяют программе спать (приостанавливать выполнение) заданный промежуток времени — **nanosleep(2)**, **clock_nanosleep(2)** и **sleep(3)**.

Различные системные вызовы позволяют процессу устанавливать таймеры, которые срабатывают в какой-то момент в будущем, и, возможно, через определённые интервалы — **alarm(2)**, **getitimer(2)**, **timerfd_create(2)** и **timer_create(2)**.

Допуск таймера

Начиная с Linux 2.6.28, возможно контролировать значение «допуска таймера» (timer slack) нити. Допуск таймера — это промежуток времени, на который ядро может задержать пробуждение определённых системных вызовов, заблокированных на время. Эта задержка позволяет ядру объединять события пробуждения, таким образом сокращая количество системных пробуждений и экономя ресурсы.