ОПЕРАЦИОННЫЕ СИСТЕМЫ И СИСТЕМНОЕ ПРОГРАММИРОВАНИЕ

Лекция 15 — Нити POSIX

Преподаватель: Поденок Леонид Петрович, 505а-5

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

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

Кафедра ЭВМ, 2022

2023.05.17

Оглавление

pthreads — потоки POSIX

В POSIX.1 определён набор интерфейсов (функции, заголовочные файлы) для работы с потоками, более известными как потоки **POSIX** или **Pthreads**.

В одном процессе может быть несколько потоков, которые выполняют одну программу. Эти потоки работают с общей глобальной памятью (сегментами данных и кучи), но у каждого потока есть собственный стек (автоматические переменные).

Также, в POSIX.1 требуется, чтобы потоки имели общий диапазон других атрибутов (т. е., эти атрибуты процесса, а не потоков):

- идентификатор процесса;
- идентификатор родительского процесса;
- идентификатор группы процессов и сеанса;
- управляющий терминал;
- идентификаторы пользователя и группы;
- открытые файловые дескрипторы;
- обычные блокировки (смотрите **fcntl(2)**);
- обработчики сигналов;
- маска создания режима доступа к файлу (**umask(2)**);
- текущий каталог (chdir(2)) и корневой каталог (**chroot(2)**);
- интервальные таймеры (**setitimer(2)**) и таймеры POSIX (**timer_create(2)**);
- значение уступчивости (**setpriority(2)**);
- ограничения по ресурсам (**setrlimit(2)**);
- измерители потребления времени ЦП (**times(2)**) и ресурсов (**getrusage(2)**);

Также в POSIX.1 кроме стека определены другие атрибуты, которые уникальны в каждом потоке:
- идентификатор потока (тип данных **pthread_t**);
- маска сигналов (**pthread_sigmask(3)**);
- переменная **errno** (определена как макрос, задающий для каждого потока собственное l-value);
- альтернативный стек сигнала (**sigaltstack(2)**);
- алгоритм и приоритет планирования реального времени (**sched(7)**);

Возвращаемые значения из функций pthreads

Большинство функций pthreads при успешном выполнении возвращает 0 или номер ошибки в противном случае.

Функции **pthreads** не изменяют **errno**. Для каждой функции pthreads, которая может вернуть ошибку, в POSIX.1-2001 определено, что функция никогда не может завершиться с ошибкой **EINTR**.

Ошибка EINTR

Многие системные вызовы сообщают/возвращают код ошибки **EINTR**, если во время выполнения этого системного вызова возник сигнал. На самом деле никакой ошибки не произошло – об этом сообщается, потому что система не может автоматически возобновить системный вызов.

Например, это может произойти, если программа использует **alarm()** для асинхронного запуска некоторого кода по истечении времени таймера. Если тайм-аут происходит, когда программа вызывает **write()**, следует просто повторить системный вызов (read/write и прочие).

```
int c = write(fd, &v, sizeof(v));
if (c == -1 && errno != EINTR) {
    perror("Write to output file");
    exit(EXIT_FAILURE);
}
```

Идентификатор потока

Каждому потоку процесса назначается уникальный идентификатор потока (имеет тип **pthread_t**). Этот идентификатор возвращается вызывающему **pthread_create(3)**, а в самом потоке его идентификатор можно получить с помощью **pthread_self(3)**.

Внутри процесса гарантируется уникальность идентификаторов потоков (во всех функциях pthreads, которые принимают аргумент идентификатора потока, подразумевается, что указан поток из вызывающего процесса).

ОС может повторно использовать идентификатор потока после объединения завершённого потока или отсоединения завершённого поток. В POSIX сказано: «Если приложение пытается использовать идентификатор потока, у который закончился срок жизни, то поведение не предсказуемо».

Потокобезопасные функции

Потокобезопасная функция — это функция, которую можно безопасно (т. е., это приведёт к единым результатам независимо от окружения) вызывать из нескольких потоков одновременно или вместе с другими потокобезопасными функциями.

В POSIX.1-2001 и POSIX.1-2008 требуется, чтобы все функции, описанные в стандарте, были потокобезопасными, за исключением следующих функций:

asctime	**drand48**	**gethostbyname****	**getutxid**	**ptsname**	**ttyname**
basename	**ecvt**	**gethostent**	**getutxline**	**putc_unlocked**	**unsetenv**
catgets	**encrypt**	**getlogin**	**gmtime**	**putchar_unlocked**	**wcrtomb******
crypt	**endgrent**	**getnetbyaddr**	**hcreate**	**putenv**	**wcsrtombs******
ctermid*	**endpwent**	**getnetbyname**	**hdestroy**	**pututxline**	**wcstombs**
ctime	**endutxent**	**getnetent**	**hsearch**	**rand**	**wctomb**
dbm_clearerr	**fcvt****	**getopt**	**inet_ntoa**	**readdir**	
dbm_close	**ftw**	**getprotobyname**	**l64a**	**setenv**	
dbm_delete	**gcvt****	**getprotobynumber**	**lgamma**	**setgrent**	
dbm_error	**getc_unlocked**	**getprotoent**	**lgammaf**	**setkey**	
dbm_fetch	**getchar_unlocked**	**getpwent**	**lgammal**	**setpwent**	
dbm_firstkey	**getdate**	**getpwnam**	**localeconv**	**setutxent**	
dbm_nextkey	**getenv**	**getpwuid**	**localtime**	**strerror**	
dbm_open	**getgrent**	**getservbyname**	**lrand48**	**strsignal*****	
dbm_store	**getgrgid**	**getservbyport**	**mrand48**	**strtok**	
dirname	**getgrnam**	**getservent**	**nftw**	**system*****	
dlerror	**gethostbyaddr****	**getutxent**	**nl_langinfo**	**tmpnam***	

* — если передаётся аргумент NULL (используется внутренний буфер);

** — [только POSIX.1-2001 (удалена из POSIX.1-2008)]

*** — [добавлена в POSIX.1-2008]

**** — если **mbstate_t *ps == NULL** (внутренний объект для хранения состояния сдвига)

Безопасные асинхронно отменяемые функции

Безопасная асинхронно отменяемая функция (async-cancel-safe function) — это функция, которую можно безопасно вызывать из потока, для которого разрешена асинхронная отмена.

Согласно POSIX.1-2001/2008 безопасными асинхронно отменяемыми должны быть только:

```
pthread_cancel()
pthread_setcancelstate() // отменяемая/неотменяемая
pthread_setcanceltype()  // асинхронный/отложенный
```

Точки отмены

В POSIX.1 определено, что некоторые функции должны, а несколько других могут было точками отмены. Если поток отменяемый и его тип отменяемости отложенный, то после получения запроса на отмену поток отменяется, когда он вызывает функцию, которая является точкой отмены.

Согласно POSIX.1-2001 и/или POSIX.1-2008 следующие функции ***должны*** быть точками отмены:

accept	**mq_receive**	**pread**	**recvfrom**	**sigwaitinfo**
aio_suspend	**mq_send**	**pselect**	**recvmsg**	**sleep**
clock_nanosleep	**mq_timedreceive**	**pthread_cond_timedwait**	**select**	**system**
close	**mq_timedsend**	**pthread_cond_wait**	**sem_timedwait**	**tcdrain**
connect	**msgrcv**	**pthread_join**	**sem_wait**	**usleep*****
creat	**msgsnd**	**pthread_testcancel**	**send**	**wait**
fcntl(F_SETLKW)	**msync**	**putmsg**	**sendmsg**	**waitid**
fdatasync	**nanosleep**	**putpmsg**	**sendto**	**waitpid**
fsync	**open**	**pwrite**	**sigpause****	**write**
getmsg	**openat***	**read**	**sigsuspend**	**writev**
getpmsg	**pause**	**readv**	**sigtimedwait**	
lockf(F_LOCK)	**poll**	**recv**	**sigwait**	

* — добавлена в POSIX.1-2008

** — только POSIX.1-2001 (перемещена в список «может» в POSIX.1-2008)

*** — только POSIX.1-2001 (функция удалена в POSIX.1-2008)

Следующие функции ***могут*** быть точками отмены согласно POSIX.1-2001 и/или POSIX.1-2008:

access	**endgrent**	**fpathconf**	**getc**	**getnetbyname**	**iconv_close**
asctime	**endhostent**	**fprintf**	**getc_unlocked**	**getnetent**	**iconv_open**
asctime_r	**endnetent**	**fputc**	**getchar**	**getopt******	**ioctl**
catclose	**endprotoent**	**fputs**	**getchar_unlocked**	**getprotobyname**	**link**
catgets	**endpwent**	**fscanf**	**getcwd**	**getprotobynumber**	**linkat***
catopen	**endservent**	**fseek**	**getdate**	**getprotoent**	**lio_listio***
chmod*	**endutxent**	**fputwc**	**getdelim***	**getpwent**	**localtime**
chown*	**faccessat***	**fputws**	**getgrent**	**getpwnam**	**localtime_r**
closedir	**fchmod***	**fread**	**getgrgid**	**getpwnam_r**	**lockf***
closelog	**fchmodat***	**freopen**	**getgrgid_r**	**getpwuid**	**lseek**
ctermid	**fchown***	**fseeko**	**getgrnam**	**getpwuid_r**	**lstat**
ctime	**fchownat***	**fsetpos**	**getgrnam_r**	**gets**	**mkdir***
ctime_r	**fclose**	**fstat**	**gethostbyaddr*****	**getservbyname**	**mkdirat***
dbm_close	**fcntl****	**fstatat***	**gethostbyname*****	**getservbyport**	**mkdtemp***
dbm_delete	**fflush**	**ftell**	**gethostent**	**getservent**	**mkfifo***
dbm_fetch	**fgetc**	**ftello**	**gethostid**	**getutxent**	**mkfifoat***
dbm_nextkey	**fgetpos**	**ftw**	**gethostname**	**getutxid**	**mknod***
dbm_open	**fgets**	**futimens***	**getline***	**getutxline**	**mknodat***
dbm_store	**fgetwc**	**fwprintf**	**getlogin**	**getwc**	**mkstemp**
dlclose	**fgetws**	**fwrite**	**getlogin_r**	**getwchar**	**mktime**
dlopen	**fmtmsg**	**fwscanf**	**getnameinfo**	**getwd*****	**nftw**
dprintf*	**fopen**	**getaddrinfo**	**getnetbyaddr**	**glob**	**opendir**

* — добавлена в POSIX.1-2008
** — для любого значения аргумента cmd
*** — только SUSv3 (функция удалена из POSIX.1-2008)
**** — если opterr не равно 0

openlog
pathconf
pclose
perror
popen
posix_fadvise
posix_fallocate
posix_madvise
posix_openpt
posix_spawn
posix_spawnp
posix_trace_clear
posix_trace_close
posix_trace_create
posix_trace_create_withlog
posix_trace_eventtypelist_getnext_id
posix_trace_eventtypelist_rewind
posix_trace_flush
posix_trace_get_attr
posix_trace_get_filter
posix_trace_get_status
posix_trace_getnext_event
posix_trace_open
posix_trace_rewind
posix_trace_set_filter
posix_trace_shutdown
posix_trace_timedgetnext_event
posix_typed_mem_open
printf
psiginfo*
psignal*
pthread_rwlock_rdlock
pthread_rwlock_timedrdlock
pthread_rwlock_timedwrlock
pthread_rwlock_wrlock
putc
putc_unlocked
putchar
putchar_unlocked
puts
pututxline
putwc
putwchar
readdir
readdir_r
readlink*
readlinkat*
remove
rename
renameat*
rewind
rewinddir
scandir*
scanf
seekdir
semop
setgrent
sethostent
setnetent
setprotoent
setpwent
setservent
setutxent
sigpause*
stat
strerror
strerror_r
strftime
symlink
symlinkat*
sync
syslog
tmpfile
tmpnam
ttyname
ttyname_r
tzset
ungetc
ungetwc
unlink
unlinkat*
utime*
utimensat*
utimes*
vdprintf*
vfprintf
vfwprintf
vprintf
vwprintf
wcsftime
wordexp
wprintf
wscanf

* — добавлена в POSIX.1-2008

Реализация может помечать и другие функции, не указанные в стандарте, как точки отмены.

В частности, реализация, вероятно, пометит как точку отмены любую нестандартную функцию, которая может блокироваться (большинство функций, работающих с файлами).

Компиляция в Linux

В Linux, программы, использующие программный интерфейс pthreads, должны компилироваться с опцией **`-pthread`**.

Реализации потоков POSIX в Linux

За всё время в библиотеке GNU C было две реализации потоков для Linux:

LinuxThreads — Первоначальная реализация pthreads. Начиная с glibc 2.4 эта реализация больше не поддерживается. Тем не менее, используется, однако здесь не обсуждается — если кто-то налетит на эти потоки, самообразуется индивидуально.

NPTL (Натуральная библиотека потоков POSIX) — Современная реализация pthreads.

По сравнению с LinuxThreads, NPTL более точно соответствует требованиям POSIX.1 и более производительна при создании большого количества потоков.

NPTL появилась в glibc начиная с версии 2.3.2, и требует свойства, появившиеся в ядре Linux 2.6.

Обе реализации являются, так называемыми реализациями 1:1, то есть **каждый поток отображается в планируемый элемента ядра**.

Обе реализации используют системный вызов Linux **`clone(2)`**[1].

В NPTL примитивы синхронизации нитей (мьютексы, объединение потоков и т .п.) реализованы с помощью системного вызова Linux **`futex(2)`**[2].

NPTL

В NPTL все потоки процесса помещаются в одну группу потоков — все члены группы потоков имеют один PID.

В NPTL нет управляющего потока[3].

Внутри NPTL используются первые два сигнала реального времени — эти сигналы нельзя использовать в приложениях.

NPTL тоже не соответствует POSIX.1, как минимум, в одном:

- у потоков могут быть разные значения уступчивости.

---

1) clone() - создать процесс-потомок
2) futex() - быстрая блокировка в пользовательском пространстве
3) Все потоки равноправны, однако есть основной поток, который породил остальные (main())

Уступчивость

В ядре работают специальные алгоритмы, которые управляют переключением процессов/потоков и распределяют между ними системные ресурсы. Есть возможность влиять на этот механизм.

При распределении системных ресурсов ядро по умолчанию рассматривает процессы/потоки как равные. Но можно сообщить ядру, что некоторый процесс/поток желает уступить часть своего права на системные ресурсы. В таком случае говорят, что уступчивость процесса увеличивается.

Так же встречается понятие «приоритет процесса/потока», характеризующий величину, обратную уступчивости.

Как измеряется уступчивость процесса, зависит от реализации. В Linux уступчивость процесса измеряется целыми числами в диапазоне от –20 до 19. По умолчанию процессы выполняются с уступчивостью 0. Запрос на получение пониженной уступчивости (повышение приоритета) из отрицательного диапазона (от –20 до –1) может выполнять только суперпользователь. Диапазон от 0 до 19 доступен всем пользователям.

Получается, что обычный пользователь может только повышать уступчивость процессов. Для изменения уступчивости запускаемого процесса предназначена команда **`nice(1,1p)`**.

Типы Pthreads

Pthreads определяет набор типов и функций на Си.

`pthread_t` — идентификатор потока;
`pthread_attr_t` — объект атрибутов потока;
`pthread_mutex_t` — мьютекс;
`pthread_mutexattr_t` — объект атрибутов мьютекса
`pthread_cond_t` — условная переменная
`pthread_condattr_t` — объект атрибута условной переменной;
`pthread_key_t` — данные, специфичные для потока;
`pthread_once_t` — контекст контроля динамической инициализации.

Организация программы, использующей потоки

Новый поток создается функцией **pthread_create()**, которой передается имя функции, с которой начнется выполнение потока, — потоковая функция. Далее, вызывающая сторона продолжает выполнять какие-то свои действия параллельно потоковой функции.

Поток завершает выполнение задачи когда:

- потоковая функция выполняет **return** и возвращает результат произведенных вычислений;
- в результате вызова завершения исполнения потока **pthread_exit()**;
- в результате вызова отмены потока **pthread_cancel()**;
- один из потоков совершает вызов **exit()**, что завершит все потоки и сам процесс;
- основной поток в функции **main()** выполняет **return**, что завершит все потоки и сам процесс;

Если основной поток в функции **main()** выполнит **pthread_exit()** вместо просто **exit()** или вместо **return**, остальные потоки продолжат исполняться.

Ожидание потока

Чтобы синхронизировать потоки, используется функция **pthread_join()**. Она ожидает завершения указанного потока. Если этот поток к тому времени был уже завершен, то функция немедленно возвращает значение.

Для досрочного завершения потока можно воспользоваться функцией **pthread_cancel()**. Функция **pthread_cancel()** возвращается сразу, но это не означает, что поток будет в обязательном порядке завершен досрочно — поток не только может самостоятельно выбрать момент завершения в ответ на вызов **pthread_cancel()**, но и его проигнорировать.

Вызов функции **pthread_cancel()** следует рассматривать как запрос на выполнение досрочного завершения потока. Поэтому, если важно быть уверенным, что поток был удален, нужно дождаться его завершения функцией **pthread_join()**.

Потоки бывают двух типов — подсоединяемые и отсоединенные. К подсоединяемым можно подсоединиться с помощью **pthread_join()** и ожидать их завершения, а к отсоединенным подсоединиться нельзя.

По умолчанию потоки создаются подсоединяемыми.

Для отсоединения потока используется вызов **pthread_detach()**. После этого перехватить поток с помощью вызова **pthread_join()** становится невозможно, также невозможно получить статус его завершения и прочее. Отменить отсоединенное состояние также невозможно.

Если завершение подсоединяемого потока не перехватить вызовом **pthread_join()** в результате получим зомби-поток.

Если завершился отсоединенный поток, все будет нормально.

Преимущества и недостатки потоков по сравнению с процессами

Преимущества:

- потоки довольно просто обмениваются данными по сравнению с процессами;
- создавать потоки для операционной системы гораздо проще и быстрее, чем создавать процессы.

Недостатки:

- при программировании мультипоточного приложения необходимо обеспечить т.н. «потоковую безопасность» функций — ***thread safety***. К приложениям, выполняющимся в виде множества процессов, таких требований не выдвигается.
- ошибка в одном из потоков может повредить остальные, поскольку потоки делят общее адресное пространство. Процессы же более изолированы друг от друга.
- потоки конкурируют друг с другом в адресном пространстве. Стек и локальная память потока захватывают часть виртуального адресного пространства процесса и делают его недоступным для других потоков. Для встраиваемых устройств такое ограничение может иметь существенное значение.

Основные вызовы Pthreads

pthread_create(3) — создаёт новый поток

```
#include <pthread.h>

int pthread_create(pthread_t *restrict thread,          // здесь возвращается ID
                   const pthread_attr_t *restrict attr, // атрибуты потока
                   void *(*start_routine)(void*),       // функция, начинающая поток
                   void *restrict arg);                 // аргумент для start_routine
```

При компиляции и компоновке необходимо использовать опцию **-pthread**.

Функция **pthread_create()** запускает новый поток в вызывающем процессе.

Новый поток начинает выполнение с функции **start_routine()**, при этом этой функции передается единственный аргумент **arg**.

Новый поток завершается одним из следующих способов:

- Поток вызывает **pthread_exit(3)**, указывая значение статуса выхода, которое будет доступно другому потоку в этом же процессе, который вызовет **pthread_join(3)**.
- Поток возвращается из **start_routine()**. Это эквивалентно вызову **pthread_exit(3)** со значением, указанным в операторе возврата.
- Поток завершается вызовом **pthread_cancel(3)**.
- Любой из потоков в процессе вызывает **exit(3)**, или основной поток выполняет возврат из **main()**. Это вызывает завершение всех потоков в процессе.

Аргумент **attr** указывает на структуру **pthread_attr_t**, содержимое которой используется во время создания потока для определения атрибутов нового потока. Эта структура инициализируется с помощью **pthread_attr_init(3)** и родственных функций.

Если attr равен NULL, поток создается с атрибутами по умолчанию.

Перечень функций для работы с атрибутами

pthread_attr_init	—	инициализировать объект атрибутов потока
pthread_attr_destroy	—	уничтожить объект атрибутов потока
pthread_attr_getstack	—	получить атрибуты стека в объекте атрибутов потока
pthread_attr_setstack	—	установить атрибуты стека в объекте атрибутов потока
pthread_attr_getstacksize	—	получить атрибут размера стека в объекте атрибутов потока
pthread_attr_setstacksize	—	установить атрибут размера стека в объекте атрибутов потока
pthread_attr_getstackaddr	—	получить атрибут адреса стека в объекте атрибутов потока
pthread_attr_setstackaddr	—	установить атрибут адреса стека в объекте атрибутов потока
pthread_attr_getaffinity_np	—	получить атрибут соответствия ЦП в объекте атрибутов потока
pthread_attr_setaffinity_np	—	установить атрибут соответствия ЦП в объекте атрибутов потока
pthread_attr_getschedparam	—	получить атрибуты параметра планирования в объекте атрибутов потока
pthread_attr_setschedparam	—	установить атрибуты параметра планирования в объекте атрибутов потока
pthread_attr_getschedpolicy	—	получить атрибут политики планирования в объекте атрибутов потока
pthread_attr_setschedpolicy	—	установить атрибут политики планирования в объекте атрибутов потока
pthread_attr_getinheritsched	—	получить атрибут наследования планировщика в объекте атрибутов потока
pthread_attr_setinheritsched	—	установить атрибут наследования-планировщика в объекте атрибутов потока
pthread_attr_getdetachstate	—	получить атрибут состояния отсоединения в объекте атрибутов потока
pthread_attr_setdetachstate	—	установить атрибут состояния отсоединения в объекте атрибутов потока
pthread_attr_getguardsize	—	получить атрибут размера защиты в объекте атрибутов потока
pthread_attr_setguardsize	—	установить атрибут размера защиты в объекте атрибутов потока
pthread_attr_getscope	—	получить атрибут области действия в объекте атрибутов потока
pthread_attr_setscope	—	установить атрибут области действия в объекте атрибутов потока

Перед возвратом успешный вызов **pthread_create()** сохраняет идентификатор нового потока ID в буфере ***thread**, на который указывает поток. Этот идентификатор используется для ссылки на поток при последующих вызовах других функций **pthreads**.

- новый поток наследует копию маски сигнала создаваемого потока (**pthread_sigmask(3)**[4]);
- набор ожидающих сигналов для нового потока пуст (**sigpending(2)**[5]).
- новый поток не наследует альтернативный стек сигналов создающего потока (**sigaltstack(2)**);
- новый поток наследует среду с плавающей запятой вызывающего потока (**fenv(3)**[6]);
- если определен **_POSIX_THREAD_CPUTIME**, у нового потока должны быть доступны часы времени ЦП, и начальное значение этих часов должно быть установлено в 0 (**pthread_getcpuclockid(3)**).

Замечания для C++

Поскольку соглашения о связывании C и C ++ несовместимы, **pthread_create()** не может получить указатель функции C++ в качестве указателя функции подпрограммы запуска. При попытке передать указатель функции C++ в **pthread_create()**, компилятор пометит это как ошибку. Однако, можно передать функцию C или C ++ в **pthread_create()**, объявив ее как **extern "C"**.

Запущенный на выполнение поток обеспечивает замкнутую область видимости относительно обработки **try-throw-catch**. Выброс исключения, выполненный в функции запуска потока, или функция, вызываемая в функции запуска, вызывает раскручивание стека до подпрограммы запуска включительно (или до тех пор, пока она не будет обнаружена). Раскрутка стека не будет выходить за пределы процедуры запуска в функцию-создатель потока. Если исключение не будет обнаружено, вызывается **std::terminate()**.

Стек исключений (для **try-throw-catch**) основан на потоках. Создание условия или повторное выполнение условия потоком не влияет на обработку исключений в другом потоке, если только условие не перехвачено.

---

4) pthread_sigmask(), sigprocmask() - проверка и изменение блокировки сигналов
5) sigpending() - проверка ожидающих сигналов
6) ISO/IEC 9899:2011 – Information technology — Programming languages — C

Детали, специфичные для Linux

Новый поток наследует копии наборов возможностей вызывающего потока (**capabilities(7)**[7]) и маску соответствия ЦП[8] (**sched_setaffinity(2)**).

Возвращаемое значение

В случае успеха **pthread_create()** возвращает 0.

При ошибке возвращается номер ошибки, а содержимое ***thread** не определено.

---

7) В ядре Linux начиная с версии 2.2, все привилегии, обычно связываемые с суперпользователем, разделены на несколько частей, называемых мандатами (capabilities), которые можно разрешать и запрещать независимо друг от друга. Мандаты являются атрибутом потока.

8) Маска соответствия — это битовая маска, указывающая, на каком процессоре (процессорах) поток или процесс должен запускаться планировщиком операционной системы. Установка маски соответствия для определенных процессов может быть полезной, поскольку в системе может существовать несколько системных процессов, выполнение которых ограничено первым процессором или ядром. Таким образом, исключение первого процессора может привести к повышению производительности приложения.

Дополнительную информацию об идентификаторе потока, который возвращается в ***thread** функцией **pthread_create()**, можно найти в описании **pthread_self(3)**, которая возвращает идентификатор вызвавшего ее потока. В частности, тип **pthread_t** может быть как целым числом, так структурным типом, поэтому возвращаемый идентификатор нельзя сравнивать с пом. оператора '**==**', для этого есть функция **pthread_equal(3)**.

Если не используются политики планирования реального времени, после вызова **pthread_create()** будет неизвестно, какой поток — вызывающий или новый — будет выполняться следующим.

Поток может быть ***подсоединяемый*** или ***отсоединенный***. Если к потоку можно подсоединиться, то другой поток может вызвать **pthread_join(3)**, чтобы дождаться завершения этого потока и получить его статус выхода. Все ресурсы потока возвращаются обратно в систему только в том случае, когда завершенный подсоединяемый поток был подсоединен, иначе имеем зомби.

Ресурсы отсоединенного потока при его завершении в систему возвращаются автоматически. Однако, при этом невозможно подключиться к потоку, чтобы получить его статус выхода.

Отсоединение потока полезно для некоторых потоков типа демонов, о статусе выхода которых приложение заботиться не должно.

По умолчанию новый поток создается в подсоединяемом состоянии (если параметр **attr** (атрибуты потока) не был установлен для создания потока специально в отсоединенном состоянии[9]).

В случае потоков NPTL, если мягкий предел ресурсов **RLIMIT_STACK** во время запуска программы имеет любое значение, кроме «unlimited», он определяет для новых потоков размер стека по умолчанию. Атрибут размера стека может быть установлен в аргументе **attr** явно[10]. Это нужно, чтобы при создания потока получить размер стека, отличный от значения по умолчанию.

---

9) pthread_attr_setdetachstate()
10) pthread_attr_setstacksize()

Если для **RLIMIT_STACK** установлено значение «unlimited», в качестве размера стека используется различное значение для каждой архитектуры. Вот значения для нескольких архитектур:

Architecture	Default stack size
i386	2 MB
IA-64	**32 MB**
PowerPC	4 MB
S/390	2 MB
Sparc-32	2 MB
Sparc-64	4 MB
x86_64	2 MB

Ошибки

EAGAIN — системе не хватает ресурсов, необходимых для создания другого потока, или установленный системой лимит на общее количество потоков в процессе {**PTHREAD_THREADS_MAX**} превышен.

EAGAIN — Обнаружено установленное системой ограничение на количество потоков. Существует ряд ограничений, которые могут вызвать эту ошибку

- был достигнут мягкий предел ресурсов **RLIMIT_NPROC** (установленный с помощью **setrlimit(2)**), который ограничивает количество процессов и потоков для реального идентификатора пользователя;
- был достигнут общесистемный предел ядра на количество процессов и потоков;
- было достигнуто максимальное количество PID.

EINVAL — Неверные установки в **attr**.

EPERM — Нет соответствующих привилегий для установки требуемых параметров планирования или политики планирования, указанных в **attr**.

Пример

Программа, которую можно найти на ftp[11], демонстрирует использование **pthread_create()**, а также ряда других функций в API pthreads.

В следующем прогоне в системе, обеспечивающей реализацию потоковой передачи NPTL, размер стека по умолчанию равен значению, заданному пределом ресурса "stack size":

```
$ ulimit -s 8192
$ ./thread hole salut servus
Thread 1: top of stack near 0x7efd69d84ed8; argv_string=hole
Thread 2: top of stack near 0x7efd69583ed8; argv_string=salut
Thread 3: top of stack near 0x7efd68d82ed8; argv_string=servus
Joined with thread 1; returned value was HOLE
Joined with thread 2; returned value was SALUT
Joined with thread 3; returned value was SERVUS
```

При следующем запуске программа явно устанавливает размер стека в 1 МБ (используя **pthread_attr_setstacksize(3)**) для создаваемых потоков:

```
$ ./thread  -s 0x100000 hola salut servus
Thread 1: top of stack near 0x7f24af498ed8; argv_string=hola
Thread 2: top of stack near 0x7f24af397ed8; argv_string=salut
Thread 3: top of stack near 0x7f24af296ed8; argv_string=servus
Joined with thread 1; returned value was HOLA
Joined with thread 2; returned value was SALUT
Joined with thread 3; returned value was SERVUS
```

11) Исходный код можно найти в ftp://lsi.bas-net.by/Лекции/СПОВМ/ex/threads/thread.c

pthread_exit(3) — завершить поток

```
#include <pthread.h>

void pthread_exit(void *retval);
```

При компиляции и компоновке необходимо использовать опцию **-pthread**.

Функция **pthread_exit()** завершает вызывающий поток и возвращает значение через **retval**, которое (если поток присоединямый) будет доступно другому потоку в том же процессе, который вызовет **pthread_join(3)**.

Любые обработчики очистки (clean-up), установленные **pthread_cleanup_push**[12]**(3)**, и которые еще не были вызваны (pop), вызываются (в обратном порядке, в котором они были назначены (push)) и выполняются.

Если поток имеет какие-либо специфичные для потока данные, то после выполнения обработчиков очистки вызываются соответствующие функции-деструкторы в неопределенном порядке.

Когда поток завершается, общие для процесса ресурсы (например, мьютексы, условные переменные, семафоры и дескрипторы файлов) не освобождаются, а функции, зарегистрированные с помощью **atexit(3)**, не вызываются.

После того, как завершается последний поток в процессе, процесс завершается вызовом **exit(3)** с нулевым статусом выхода. Таким образом, высвобождаются ресурсы, совместно используемые процессом, после чего вызываются функции, зарегистрированные с помощью **atexit(3)**.

---

12) pthread_cleanup_push(), pthread_cleanup_pop() — push и pop обработчики очистки при отмене потоков

Возвращаемое значение

Функция не возвращает управление вызывающей стороне

Ошибки

Функция всегда завершается успешно.

Замечания

Выполнение возврата любого потока из функции **start_routine()** за исключением основного приводит к неявному вызову **pthread_exit()** с использованием возвращаемого функцией значения в качестве статуса выхода потока.

Чтобы другие потоки могли продолжить выполнение, основной поток должен завершиться вызовом **pthread_exit()**, а не **exit(3)**.

Значение, на которое указывает **retval**, не должно находиться в стеке вызывающего потока, поскольку после завершения потока содержимое этого стека не определено.

Возможные проблемы

В настоящее время в логике реализации линукс-ядра есть ограничения на **wait(2)** остановленной группы потоков с лидером группы мертвых потоков. Это может проявляться в таких проблемах, как заблокированный терминал, если сигнал остановки отправляется процессу переднего плана, лидер группы потоков которого уже вызвал **pthread_exit()**.

pthread_self() — получить ID вызывающего потока

```
#include <pthread.h> // Компилируется и компонуется вместе с опцией -pthread

pthread_t pthread_self(void);
```

Функция **pthread_self()** возвращает идентификатор вызывающего потока.

Это то же значение, которое возвращается в ***thread** в вызове **pthread_create(3)**, создавшем этот поток.

Эта функция всегда завершается успешно, возвращая идентификатор вызывающего потока.

Надо отметить, что стандарт предоставляет широкую свободу реализации в выборе типа, используемого для представления идентификатора потока. Например, допускается представление с использованием арифметического типа или структуры. Следовательно, переменные типа **pthread_t** нельзя переносимо сравнивать с помощью оператора равенства языка C (**==**), вместо этого следует использовать **pthread_equal(3)**.

Идентификаторы потоков следует считать непрозрачными — любая попытка использовать идентификатор потока, отличный от вызовов **pthreads**, непереносима и может привести к неопределенным результатам.

Идентификаторы потоков гарантированно будут уникальными только внутри процесса.

Идентификатор потока может быть повторно использован после того, как завершенный поток был присоединен или отсоединенный поток был завершен.

Идентификатор потока, возвращаемый **pthread_self()**, не совпадает с идентификатором потока ядра, возвращаемым вызовом **gettid(2)**[13].

13) gettid() is Linux-specific and should not be used in programs that are intended to be portable.

pthread_equal() — сравнить идентификаторы потоков

```
#include <pthread.h>  // Компилируется и компонуется вместе с опцией -pthread

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

Функция **pthread_equal()** сравнивает два идентификатора потока.

Если два идентификатора потока равны, **pthread_equal()** возвращает ненулевое значение.
В противном случае возвращается 0.
Если либо **t1**, либо **t2** не являются допустимыми идентификаторами потока, поведение не определено.

Ошибки
Функция всегда завершается успешно.
Функция **pthread_equal()** необходима, потому что идентификаторы потоков следует считать непрозрачными — у приложений нет переносимого способа прямого сравнения двух значений **pthread_t**.

pthread_detach() — отсоединить поток

```
#include <pthread.h>  // Компилируется и компонуется вместе с опцией -pthread

int pthread_detach(pthread_t thread);
```

Функция **pthread_detach()** помечает поток, идентифицированный параметром **thread**, как отсоединенный.

Когда отсоединенный поток завершается, его ресурсы автоматически возвращаются обратно в систему без необходимости соединения (**join**) какого-либо другого потока с завершенным потоком.

Функция **pthread_detach()** информирует реализацию, что память для потока **thread** , если этот поток завершился, может быть освобождена.

Если поток не завершился, **pthread_detach()** не вызывает его завершение.

Попытка отсоединить уже отсоединенный поток приводит к неопределенному поведению.

Если значение, указанное аргументом потока для **pthread_detach()**, не относится к присоединяемому потоку, поведение не определено.

Атрибут отсоединения просто определяет поведение системы при завершении потока — он не предотвращает завершение потока, если процесс завершается с помощью **exit(3)** (или, что эквивалентно, если основной поток возвращает управление).

Возвращаемое значение

Если вызов завершился успешно, **pthread_detach()** вернет 0, в противном случае возвращает номер ошибки.

Ошибки

EINVAL — поток не является присоединяемым потоком.

ESRCH — Не удалось найти поток с указанным идентификатором потока.

После того, как поток был отсоединен, он не может быть присоединен с помощью **pthread_join(3)** или стать доступным для присоединения.

Новый поток можно создать сразу в отсоединенном состоянии с помощью функции **pthread_attr_setdetachstate(3)**, которая устанавливает атрибут **detached** в аргументе **attr** функции **pthread_create(3)**.

Чтобы можно было освободить системные ресурсы, выделяемые потоку, для каждого потока, создаваемого приложением, следует вызывать либо **pthread_join()**, либо **pthread_detach()**.

Тем не менее, ресурсы любых потоков, для которых не было выполнено одно из этих действий, будут освобождены после завершения процесса.

Функции **pthread_join()** или **pthread_detach()** в конечном итоге должны вызываться для каждого создаваемого потока, чтобы можно было освободить память, связанную с потоком.

Пример
Следующий паттерн отсоединяет вызывающий его поток:

```
pthread_detach(pthread_self());
```

Обоснование

Было высказано предположение, что в функции отсоединения нет необходимости — атрибут создания потока detach state является более, чем достаточным, поскольку поток никогда не должен отсоединяться динамически. Однако необходимость возникает как минимум в двух случаях:

1. В обработчике отмены для pthread_join() почти необходимо иметь функцию pthread_detach(), чтобы отсоединить поток, на котором ожидает pthread_join(). Без этого обработчик должен был бы сделать еще один pthread_join(), чтобы попытаться отсоединить поток, что одновременно отложило бы обработку отмены на неограниченный период и представило бы новый вызов pthread_join(), который сам может нуждаться в отмене. В этом случае практически необходимо динамическое отсоединение.

2. Чтобы отсоединить «начальный поток» (что может быть желательно в процессах, которые настраивают серверные потоки).

Если реализация обнаруживает, что значение, указанное аргументом потока для pthread_detach(), не относится к присоединяемому потоку, функция может завершиться с ошибкой **EINVAL**.

Если реализация обнаруживает использование идентификатора потока по окончании своего жизненного цикла, рекомендуется, функция может завершиться с ошибкой **ESRCH**.

pthread_yield() — отдать процессор

```
#define _GNU_SOURCE             /* See feature_test_macros(7) */
#include <pthread.h>

int pthread_yield(void);
```

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

pthread_yield() заставляет вызывающий поток отказаться от ЦП.
Поток помещается в конец очереди выполнения согласно его статического приоритета, и запускается другой поток.

Возвращаемое значение
В случае успеха **pthread_yield()** возвращает 0, при ошибке возвращается номер ошибки.

Ошибки
В Linux этот вызов всегда выполняется успешно (но переносимые и перспективные приложения, тем не менее, должны обрабатывать возможный возврат ошибки).
Этот вызов нестандартный, но присутствует в нескольких других системах. Вместо этого следует использовать стандартизированный **sched_yield(2)**.

В Linux эта функция реализована как вызов **sched_yield(2)**.
pthread_yield() предназначен для использования с политиками планирования в реальном времени (например, **SCHED_FIFO** или **SCHED_RR**). Использование **pthread_yield()** с недетерминированными политиками планирования, такими как **SCHED_OTHER**, не определено и, скорее всего, означает, что дизайн приложения неверный.

sched_yield(2) – освободить процессор

```
#include <sched.h>

int sched_yield(void);
```

Функция **sched_yield()** заставляет вызывающий ее поток отказаться от процессора до тех пор, пока он снова не станет главой своего списка потоков. Аргументов не требует.

Согласно POSIX.1 функция **sched_yield()** должна возвращать 0 в случае успешного завершения, а в противном случае будет возвращено значение -1 и установлено **errno**, указывающее на ошибку.

В реализации Linux **sched_yield()** всегда завершается успешно.

В рамках концептуальной модели семантики планирования в POSIX.1 определяется набор списков потоков. Этот набор списков потоков всегда присутствует независимо от параметров планирования, поддерживаемых системой.

В системе, где параметр планирования процессов не поддерживается, переносимые приложения не должны делать никаких предположений относительно того, будут ли потоки из других процессов находиться в одном списке потоков.

Если вызывающий поток является единственным потоком в списке с наивысшим приоритетом в это время, после вызова **sched_yield()** он продолжит работу.

Системы POSIX, в которых доступен **sched_yield()**, в **<unistd.h>** определяют макрос **_POSIX_PRIORITY_SCHEDULING**.

Функция **sched_yield()** предназначена для использования с политиками планирования реального времени (**SCHED_FIFO** или **SCHED_RR**).

Использование **sched_yield()** с недетерминированными политиками планирования, такими как **SCHED_OTHER**, скорее всего, означает, что дизайн вашего приложения следует пересмотреть.

Стратегические вызовы **`sched_yield()`** могут повысить производительность, давая другим потокам или процессам возможность работать, когда вызывающая сторона освобождает сильно конкурирующие ресурсы, например, мьютексы.

Следует избегать ненужных или неуместных вызовов **`sched_yield()`**, например, когда ресурсы, необходимые другим запланированным потокам, все еще удерживаются вызывающей стороной, поскольку это приведет к ненужным переключениям контекста, что снизит производительность системы.

pthread_join(3) — ждать завершения потока

```
#include <pthread.h>

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

При компиляции и компоновке необходимо использовать опцию **-pthread**.

Функция **pthread_join()** ожидает завершения потока, указанного потоком.
Если этот поток уже завершился, **pthread_join()** немедленно возвращается.
Поток, указанный в **pthread_join()**, должен быть подсоединяемым.
Если **retval** не равно **NULL**, то **pthread_join()** копирует статус выхода целевого потока (то есть значение, которое целевой поток предоставил **pthread_exit(3)**) в то место, на которое указывает **retval**.
Если целевой поток был отменен, в то место, на которое указывает **retval**, помещается значение **PTHREAD_CANCELED**[14].
Если несколько потоков одновременно пытаются подсоединиться к одному и тому же потоку, результаты не определены.
Если поток, вызывающий **pthread_join()**, отменен, то целевой поток останется доступным для подсоединения (т.е. он не будет отсоединен).

Возвращаемое значение
В случае успеха **pthread_join()** возвращает 0; при ошибке возвращается номер ошибки.

14) #define PTHREAD_CANCELED ((void *) -1)

После успешного завершения **pthread_join(3)** вызывающей стороне гарантируется, что целевой поток завершен. Затем вызывающая сторона может выбрать любую очистку, которая требуется после завершения потока (например, освобождение памяти или других ресурсов, которые были выделены целевому потоку).

Присоединение ранее присоединенного потока приводит к неопределенному поведению.

Неприсоединение к потоку, к которому можно подсоединиться, создает «зомби-поток».

Следует избегать этого, поскольку каждый зомби-поток потребляет некоторые системные ресурсы, и когда накопится достаточно зомби-потоков, создание новых потоков (или процессов) станет невозможным.

В **pthreads** нет аналога **waitpid(-1, &status, 0)**, то есть «дождаться любого завершенного потока». Разработчики Pthreads на сей счет говорят:

«Если вы считаете, что вам нужна эта функциональность, возможно, вам нужно переосмыслить дизайн своего приложения».

Все потоки в процессе являются одноранговыми — любой поток может присоединиться к любому другому потоку в процессе.

Ошибки

EDEADLK — Обнаружена тупиковая ситуация (например, два потока пытались подсоединиться друг к другу), или **thread** указывает вызывающий поток.

EINVAL — поток не является присоединяемым потоком.

EINVAL — другой поток уже ожидает присоединения к этому потоку.

ESRCH — не удалось найти поток с указанным идентификатором потока.

Пример создания и удаления потока:

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

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

int main(void) {
    int        ar[1000000];
    pthread_t  th1, th2; // планируем создание двух потоков
    subarray   sb1, sb2; // объекты типа subarray для них в качестве аргументов вызова

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

    sb2.ar = &ar[500000];
    sb2.n  = 500000;
    (void) pthread_create(&th2, NULL, incrementer, &sb2); // возврат не нужен

    (void) pthread_join(th1, NULL); // retval = NULL -- статус завершения не нужен
    (void) pthread_join(th2, NULL);
    return 0;
}
```

pthread_attr_init/destroy(3) — инициализировать/уничтожить объект атрибутов потока

```
#include <pthread.h> // При компиляции и компоновке необходимо использовать -pthread.

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

Функция **pthread_attr_init()** инициализирует объект атрибутов потока, на который указывает **attr**, значениями атрибутов по умолчанию. После этого вызова отдельные атрибуты объекта могут быть установлены с помощью различных родственных функций **pthread_attr_***, после чего объект может использоваться в одном или нескольких вызовах **pthread_create(3)**, создающих потоки.

Вызов pthread_attr_init() для уже инициализированного объекта атрибутов потока приводит к неопределенному поведению.

Когда объект атрибутов потока больше не требуется, его следует уничтожить с помощью функции **pthread_attr_destroy()**.

Уничтожение объекта атрибутов потока не влияет на потоки, созданные с использованием этого объекта.

После уничтожения объекта атрибутов потока его можно повторно инициализировать с помощью **pthread_attr_init()**.

Любое другое использование объекта атрибутов уничтоженного потока приводит к неопределенным результатам.

Тип **pthread_attr_t** следует рассматривать как непрозрачный — любой доступ к объекту, кроме функций **pthreads**, является непереносимым и дает неопределенные результаты.

Возвращаемое значение

В случае успеха эти функции возвращают 0.

В случае ошибки они возвращают ненулевой номер ошибки.

Ошибки

POSIX.1 документирует ошибку **ENOMEM** для **pthread_attr_init()**.

В Linux эти функции всегда выполняются успешно (но переносимые и перспективные приложения, тем не менее, должны обрабатывать возможный возврат ошибки).

Пример

В приведенной ниже программе необязательно используется **pthread_attr_init()** и различные связанные с объектом атрибутов функции для инициализации объекта атрибутов потока, который используется для создания одного потока. После создания поток использует функцию **pthread_getattr_np(3)** (нестандартное расширение GNU) для получения атрибутов потока, а затем отображает эти атрибуты.

Если программа запускается без аргумента командной строки, она передает **NULL** в качестве аргумента **attr** функции **pthread_create(3)**, так что поток создается с атрибутами по умолчанию.

Скомпилировав и запустив программу на Linux/x86-64 с реализацией потоковой передачи NPTL, мы увидим следующее:

```
$ ulimit -s # No stack limit ==> default stack size is 2 MB unlimited (FC36 -- 8k)
$ ./tattr
Thread attributes:
        Detach state        = PTHREAD_CREATE_JOINABLE
        Scope               = PTHREAD_SCOPE_SYSTEM
        Inherit scheduler   = PTHREAD_INHERIT_SCHED
        Scheduling policy   = SCHED_OTHER
        Scheduling priority = 0
        Guard size          = 4096 bytes
        Stack address       = 0x7f89320b3000
        Stack size          = 0x800000 bytes
```

Когда мы указываем размер стека в качестве аргумента командной строки, программа инициализирует объект атрибутов потока, устанавливает различные атрибуты в этом объекте и передает указатель на объект при вызове **pthread_create(3)**.

Скомпилировав и запустив программу на Linux/x86-64 с реализацией потоковой передачи NPTL, мы увидим следующее:

```
$ gcc -Wall -W -std=c11 -Wextra -Wno-unused-parameter -pthread -o tattr threadattr.c
$ ./tattr 0x3000000
posix_memalign() allocated at 0x7f8c61017000
Thread attributes:
Detach state        = PTHREAD_CREATE_DETACHED
Scope               = PTHREAD_SCOPE_SYSTEM
Inherit scheduler   = PTHREAD_EXPLICIT_SCHED
Scheduling policy   = SCHED_OTHER
Scheduling priority = 0
Guard size          = 0 bytes
Stack address       = 0x7f8c61017000
Stack size          = 0x3000000 bytes
```

Исходный код лежит на **ftp://lsi.bas-net.by/Лекции/СПОВМ/ex/thread2/ threadattr.c**

pthread_cancel(3) — Запрос отмены (запрос на завершении потока)

```
#include <pthread.h> // При компиляции и компоновке необходимо использовать -pthread

int pthread_cancel(pthread_t thread);
```

Функция **pthread_cancel()** отправляет ***запрос отмены*** потоку **thread**.

Отреагирует ли целевой поток на запрос отмены, и когда, зависит от двух атрибутов, находящихся под контролем этого потока – его состояния отмены (cancelability state) и типа отмены.

Состояние возможности отмены потока

Состояние возможности отмены потока определяется **pthread_setcancelstate(3)**, может быть включено (по умолчанию включено для новых потоков) или отключено.

Если поток **отключил** отмену, то запрос отмены остается в очереди до тех пор, пока поток не разрешит отмену.

Если поток **включил** отмену, то его *тип отмены* определяет, когда происходит отмена.

Тип отмены потока

Тип отмены потока определяется **pthread_setcanceltype(3)**, может быть асинхронным или отложенным (по умолчанию для новых потоков).

Асинхронная возможность отмены означает, что поток может быть отменен в любое время, обычно немедленно, но система этого не гарантирует.

Отложенная возможность отмены означает, что отмена будет отложена до тех пор, пока поток не вызовет функцию, которая является точкой отмены.

Список функций, которые являются или могут быть точками отмены, представлен в **pthreads(7)**.

Когда выполняется действие по запросу отмены, для потока выполняются следующие шаги (именно в данном порядке):

1. Вызываются (popped) обработчики очистки отмены в обратном порядке тому, в котором они были зарегистрированы (pushed) (**pthread_cleanup_push(3)**).
2. В неопределенном порядке вызываются деструкторы данных потока (**pthread_key_create(3)**).
3. Поток завершается (**pthread_exit(3)**).

Вышеупомянутые шаги выполняются асинхронно по отношению к вызову **pthread_cancel().** Статус возврата функции **pthread_cancel()** просто информирует вызывающего, был ли запрос отмены успешно поставлен в очередь.

При попытке соединиться[15] с потоком после его завершения по запросу отмены в качестве статуса выхода потока будет использоваться **PTHREAD_CANCELED**.

Соединение с потоком — единственный способ узнать, что отмена завершена.

В Linux отмена осуществляется с помощью сигналов. В реализации потоковой передачи NPTL для этой цели используется первый сигнал реального времени (т.е. сигнал с номером 32).

В LinuxThreads используется второй сигнал реального времени, если доступны сигналы реального времени, в противном случае используется **SIGUSR2**.

---

15) pthread_join(3)

Пример

Программа создает поток, после чего его отменяет.

Основной поток соединяется с отмененным потоком, чтобы проверить, что его статус выхода был **PTHREAD_CANCELED**.

Когда мы запускаем программу происходит следующее:

```
$ gcc -Wall -W -std=c11 -Wextra -Wno-unused-parameter -pthread -o tcanc threadcan.c
$ ./tcanc
thread_func(): started; cancellation disabled
main(): sending cancellation request
thread_func(): about to enable cancellation
main(): thread was canceled
```

Исходный код лежит на **ftp://lsi.bas-net.by/Лекции/СПОВМ/ex/thread3/threadcan.c**

pthread_setcancelstate(), pthread_setcanceltype() – изменяет состояния и тип отменяемости

```
#include <pthread.h> // Компилируется и компонуется вместе с -pthread

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

Функция **pthread_setcancelstate( )** изменяет состояние возможности отмены (отменяемости) вызывающего потока на значение **state**. Предыдущее состояние отменяемости потока возвращается в буфер, на который указывает **oldstate**.

Аргументом **state** должно быть одно из следующих значений:

PTHREAD_CANCEL_ENABLE – поток является отменяемым. Для всех новых потоков, включая начальный, это состояние является основным по умолчанию. Тип отменяемости потока определяет, когда поток будет отвечать на запрос об отмене.

PTHREAD_CANCEL_DISABLE – поток является неотменяемым. Если был получен запрос об отмене, он будет блокироваться до тех пор, пока не будет включена отменяемость.

Функция **pthread_setcanceltype( )** изменяет тип отменяемости вызывающего потока на значение **type**. Предыдущий тип отменяемости потока возвращается в буфер, на который указывает **oldstate**.

Аргументом **type** должно быть одно из следующих значений:

PTHREAD_CANCEL_DEFERRED – запрос отменяемости откладывается до тех пор, пока поток не вызовет функцию, являющуюся точкой отмены. Данный тип устанавливается по умолчанию для всех потоков, включая начальный.

PTHREAD_CANCEL_ASYNCHRONOUS – поток может быть отменен в любой момент, практически сразу же после получения запроса об отмене, однако система этого не гарантирует.

Операции установки и получения (set-and-get), выполняемые каждой из этих функций, являются атомарными для предотвращения пересечения с другими процессами, вызывающими ту же функцию.

Асинхронная отменяемость

Задание типа отменяемости **PTHREAD_CANCEL_ASYNCHRONOUS** полезно редко. Так как поток может быть отменена в любой момент, невозможно безопасно резервировать ресурсы, например, выделять память с помощью **malloc()**, захватывать мьютексы, семафоры или блокировки и так далее. Резервирование ресурсов в этом случае небезопасно, так как приложение не может узнать состояние этих ресурсов при отмене потока, то есть, произошла ли отмена до резервирования ресурсов, во время резервирования или после их освобождения? Кроме того, некоторые внутренние структуры данных, например, связные списки свободных блоков, управляемые семейством функций **malloc()**, могут остаться в не целостном состоянии, если отмена происходит в середине вызова функции. Следовательно, обработчики очистки становятся бессмысленными.

Функции, при выполнении которых поток может быть безопасно асинхронно отменен, называются функциями async-cancel-safe. В POSIX.1-2001 и POSIX.1-2008 требуется, чтобы такими функция были только **pthread_cancel(3)**, **pthread_setcancelstate()** и **pthread_setcanceltype()**. В общем случае из асинхронно отменяемого потока нельзя безопасно вызывать другие функции библиотеки.

Одной из нескольких ситуаций, в которых асинхронная отменяемость полезна, является отменяемость потока, который находится в цикле и занимается только вычислениями.

Возвращают

При успешном выполнении эти функции возвращают 0; при ошибке возвращается ненулевой номер ошибки.

Ошибки

Функция pthread_setcancelstate() может завершиться со следующей ошибкой:

EINVAL – неправильное значение для state.

Функция pthread_setcanceltype() может завершиться со следующей ошибкой:

EINVAL – неправильное значение для type.

pthread_testcancel() — запросить доставку любого ожидающего запроса на отмену

```
#include <pthread.h>  // Компилируется и компонуется вместе с опцией -pthread

void pthread_testcancel(void);
```

Вызов **pthread_testcancel()** создает точку отмены в вызывающем потоке, поэтому поток, который выполняет код, не содержащий точек отмены, ответит на запрос отмены по достижению данной функции.

Если возможность отмены отключена[16] или нет ожидающего запроса на отмену, то вызов **pthread_testcancel()** не имеет никакого эффекта.

Возвращаемое значение

Эта функция не возвращает значения. Если вызывающий поток отменяется вследствие вызова этой функции, функция не возвращает управления.

Эта функция всегда завершается успешно.

16) pthread_setcancelstate()

pthread_cleanup_push()/pop() — установить обработчики очистки при завершении

```
#include <pthread.h> // Компилируется и компонуется вместе с опцией -pthread.

void pthread_cleanup_push(void (*routine)(void *), void *arg);
void pthread_cleanup_pop(int execute);
```

Эти функции управляют стеком вызывающего потока, содержащим обработчики очистки при отмене потока.

Обработчик очистки — это функция, которая автоматически выполняется, когда поток отменяется (или в различных других обстоятельствах, описанных ниже). Он может, например, разблокировать мьютекс, чтобы он стал доступным для других потоков в процессе.

Функция **pthread_cleanup_push()** помещает указанную процедуру на вершину стека обработчиков очистки. Когда процедура будет позже вызвана, ей будет передан аргумент **arg**.

Вызов зарегистрированной процедуры происходит, когда:

- поток завершается (то есть вызывает **pthread_exit()**).
- поток действует по запросу на отмену.
- поток вызывает **pthread_cleanup_pop()** с ненулевым аргументом **execute**.

Функция **pthread_cleanup_pop()** удаляет подпрограмму из вершины стека обработчиков очистки и, если **execute** не равно нулю, ее выполняет.

Стандарт допускает, что **pthread_cleanup_push()** и **pthread_cleanup_pop()** могут быть как функциями, так и макросами.

Эффект от использования операторов **return**, **break**, **continue** и **goto** для преждевременного выхода из блока кода, описываемого парой вызовов функций **pthread_cleanup_push()** и **pthread_cleanup_pop()**, не определен.

Эти функции не возвращают значение. Ошибок нет.

Примечание

В Linux функции **`pthread_cleanup_push()`** и **`pthread_cleanup_pop()`** реализованы как макросы, которые расширяются до текста, содержащего '**`{`**' и '**`}`**' соответственно.

Это означает, что переменные, объявленные в рамках парных вызовов этих функций, будут видны только в этой области.

Две процедуры `pthread_cleanup_push()` и `pthread_cleanup_pop()`, которые регистрируют и выбирают обработчики очистки отмены, можно рассматривать как левую и правую круглые скобки. Они всегда должны быть парными.

pthread_sigmask() — проверить и изменить маску заблокированных сигналов

```
#include <signal.h>

int pthread_sigmask(int how,
                    const sigset_t *restrict set,
                    sigset_t *restrict oset);

int sigprocmask(int how,
                const sigset_t *restrict set,
                sigset_t *restrict oset);
```

При компиляции и компоновке неоходимо использовать опцию **-pthread**.

Функция **pthread_sigmask()** аналогична **sigprocmask(2)** с той разницей, что в POSIX.1 явно определено ее использование в многопоточных программах.

Функция **pthread_sigmask()** проверяет или изменяет (или и то, и другое) маску сигналов вызывающего потока, независимо от количества потоков в процессе.

Функция полностью эквивалентна **sigprocmask()**, но без ограничения, что вызов должен выполняться в однопоточном процессе.

Использование функции sigprocmask() в многопоточном процессе не определено.

В однопоточном процессе функция **sigprocmask()** проверяет или изменяет (или и то, и другое) маску сигналов вызывающего потока.

Если **set** *является* нулевым указателем (**NULL**), маска сигнала потока остается неизменной — таким образом, вызов можно использовать для запроса заблокированных в данный момент сигналов.

Если аргумент **set** *не является* нулевым указателем (**NULL**), он указывает на набор сигналов, которые будут использоваться для изменения текущего заблокированного набора.

sigset_t – тип набора сигналов POSIX.

Для работы с наборами сигналов POSIX используются следующие функции[17]:

```
#include <signal.h>                                    // _POSIX_C_SOURCE

int sigemptyset(sigset_t *set);                        // инициализация набора сигналов
int sigfillset(sigset_t *set);                         // инициализация набора сигналов
int sigaddset(sigset_t *set, int signum);              // добавить сигнал в набор
int sigdelset(sigset_t *set, int signum);              // удалить сигнал из набора
int sigismember(const sigset_t *set, int signum); // проверить присутствие сигнала
```

Аргумент **how** указывает способ изменения набора сигналов:

SIG_BLOCK — результирующий набор должен быть объединением текущего набора и набора сигналов, на который указывает **set**.

SIG_SETMASK — результирующий набор должен быть набором сигналов, на который указывает **set**.

SIG_UNBLOCK — результирующий набор должен быть пересечением текущего набора и дополнением набора сигналов, на который указывает **set**.

Если аргумент **oset** не является нулевым указателем, предыдущая маска должна быть сохранена в месте, на которое указывает **oset**.

Если **set** является нулевым указателем, значение аргумента **how** не имеет значения, и маска сигнала потока остается неизменной — таким образом, вызов можно использовать для запроса заблокированных в данный момент сигналов.

Если после вызова **sigprocmask()** есть какие-либо ожидающие разблокированные сигналы, по крайней мере один из этих сигналов будет доставлен до возврата из вызова **sigprocmask()**.

Невозможно заблокировать те сигналы, которые нельзя игнорировать. Операционная система ***должна***[18] обеспечивать это без указания ошибки.

---

17) см. LK07-Сигналы в UNIX

18) Требование POSIX

Если какой-либо из сигналов **SIGFPE**, **SIGILL**, **SIGSEGV** или **SIGBUS** генерируется в то время, когда они заблокированы, результат будет не определен, если только сигнал не был сгенерирован действием другого процесса или одной из функций **kill()**, **pthread_kill()**, **raise()** или **sigqueue()**.

Возвращаемое значение

После успешного завершения **sigprocmask()** и **pthread_sigmask()** вернут 0.

В противном случае будет возвращено **-1**, а значение **errno** будет указывать на ошибку. При этом маска сигнала процесса останется неизменной.

Ошибки

Функции **pthread_sigmask()** и **sigprocmask()** завершатся ошибкой, если:

EINVAL — значение аргумента **how** не равно ни одному из определенных значений.

Функция **pthread_sigmask()** не возвращает код ошибки **EINTR**.

Замечания:

Новый поток наследует копию маски сигнала своего создателя.

Функция glibc **pthread_sigmask()** молча игнорирует попытки заблокировать два сигнала реального времени, которые используются внутри реализации потоковой передачи NPTL (**nptl(7)**).

Приведенная ниже программа блокирует несколько сигналов в основном потоке, а затем создает специальный поток для получения этих сигналов через **sigwait(3)**.

Сеанс оболочки демонстрирует его использование:

```
$ ./run &
[1] 1912538
$ kill -QUIT %1
Signal handling thread got signal 3
$ kill -USR1 %1
Signal handling thread got signal 10
$ kill -TERM %1
[1]+  Завершено      ./run
```

Программный код

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

// Simple error handling functions

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

static void *sig_thread(void *arg) {

    sigset_t *set = arg;
    int sig;

    for (;;) {
        int s = sigwait(set, &sig);
        if (s != 0) {
            handle_error_en(s, "sigwait");
        }
        printf("Signal handling thread got signal %d\n", sig);
    }
}
```

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

    pthread_t thread;
    sigset_t  set;

    // Блокируем SIGQUIT and SIGUSR1. Другой поток, созданный main()
    // унаследует копию маски сигналов

    sigemptyset(&set);        // инициализируем пустой набор
    sigaddset(&set, SIGQUIT); // добавим SIGQUIT
    sigaddset(&set, SIGUSR1); // добавим SIGUSR1
    int s = pthread_sigmask(SIG_BLOCK, &set, NULL);
    if (s != 0) {
        handle_error_en(s, "pthread_sigmask");
    }

    s = pthread_create(&thread, NULL, &sig_thread, (void *)&set);
    if (s != 0) {
        handle_error_en(s, "pthread_create");
    }

    // Основной поток продолжает создавать другие потоки и/или
    // выполнять другую работу
    //
    pause();      // Просто пауза, чтобы протестировать работу
}
```