NAME

pthreads − POSIX threads

DESCRIPTION

POSIX.1 specifies a set of interfaces (functions, header files) for threaded programming commonly known as POSIX threads, or Pthreads. A single process can contain multiple threads, all of which are executing the same program. These threads share the same global memory (data and heap segments), but each thread has its own stack (automatic variables).

POSIX.1 also requires that threads share a range of other attributes (i.e., these attributes are process-wide rather than per-thread):

- process ID
- parent process ID
- process group ID and session ID
- controlling terminal
- user and group IDs
- open file descriptors
- record locks (see **fcntl**(2))
- signal dispositions
- file mode creation mask (**umask**(2))
- current directory (**chdir**(2)) and root directory (**chroot**(2))
- interval timers (**setitimer**(2)) and POSIX timers (**timer_create**(2))
- nice value (**setpriority**(2))
- resource limits (**setrlimit**(2))
- measurements of the consumption of CPU time (**times**(2)) and resources (**getrusage**(2))

As well as the stack, POSIX.1 specifies that various other attributes are distinct for each thread, including:

- thread ID (the *pthread_t* data type)
- signal mask (**pthread_sigmask**(3))
- the *errno* variable
- alternate signal stack (**sigaltstack**(2))
- real-time scheduling policy and priority (**sched**(7))

The following Linux-specific features are also per-thread:

- capabilities (see **capabilities**(7))
- CPU affinity (**sched_setaffinity**(2))

Pthreads function return values

Most pthreads functions return 0 on success, and an error number on failure. The error numbers that can be returned have the same meaning as the error numbers returned in *errno* by conventional system calls and C library functions. Note that the pthreads functions do not set *errno*. For each of the pthreads functions that can return an error, POSIX.1-2001 specifies that the function can never fail with the error **EINTR**.

Thread IDs

Each of the threads in a process has a unique thread identifier (stored in the type *pthread_t*). This identifier is returned to the caller of **pthread_create**(3), and a thread can obtain its own thread identifier using **pthread_self**(3).

Thread IDs are guaranteed to be unique only within a process. (In all pthreads functions that accept a thread ID as an argument, that ID by definition refers to a thread in the same process as the caller.)

The system may reuse a thread ID after a terminated thread has been joined, or a detached thread has terminated. POSIX says: "If an application attempts to use a thread ID whose lifetime has ended, the behavior is undefined."

Thread-safe functions

A thread-safe function is one that can be safely (i.e., it will deliver the same results regardless of whether it is) called from multiple threads at the same time.

POSIX.1-2001 and POSIX.1-2008 require that all functions specified in the standard shall be thread-safe, except for the following functions:

```
asctime()
basename()
catgets()
crypt()
ctermid() if passed a non-NULL argument
ctime()
dbm_clearerr()
dbm_close()
dbm_delete()
dbm_error()
dbm_fetch()
dbm_firstkey()
dbm_nextkey()
dbm_open()
dbm_store()
dirname()
dlerror()
drand48()
ecvt() [POSIX.1-2001 only (function removed in POSIX.1-2008)]
encrypt()
endgrent()
endpwent()
endutxent()
fcvt() [POSIX.1-2001 only (function removed in POSIX.1-2008)]
ftw()
gcvt() [POSIX.1-2001 only (function removed in POSIX.1-2008)]
getc_unlocked()
getchar_unlocked()
getdate()
getenv()
getgrent()
getgrgid()
getgrnam()
gethostbyaddr() [POSIX.1-2001 only (function removed in
                 POSIX.1-2008)]
gethostbyname() [POSIX.1-2001 only (function removed in
                 POSIX.1-2008)]
gethostent()
getlogin()
getnetbyaddr()
getnetbyname()
getnetent()
getopt()
getprotobyname()
getprotobynumber()
```

```
getprotoent()
getpwent()
getpwnam()
getpwuid()
getservbyname()
getservbyport()
getservent()
getutxent()
getutxid()
getutxline()
gmtime()
hcreate()
hdestroy()
hsearch()
inet_ntoa()
l64a()
lgamma()
lgammaf()
lgammal()
localeconv()
localtime()
lrand48()
mrand48()
nftw()
nl_langinfo()
ptsname()
putc_unlocked()
putchar_unlocked()
putenv()
pututxline()
rand()
readdir()
setenv()
setgrent()
setkey()
setpwent()
setutxent()
strerror()
strsignal() [Added in POSIX.1-2008]
strtok()
system() [Added in POSIX.1-2008]
tmpnam() if passed a non-NULL argument
ttyname()
unsetenv()
wcrtomb() if its final argument is NULL
wcsrtombs() if its final argument is NULL
wcstombs()
wctomb()
```

Async-cancel-safe functions

An async-cancel-safe function is one that can be safely called in an application where asynchronous cancelability is enabled (see **pthread_setcancelstate**(3)).

Only the following functions are required to be async-cancel-safe by POSIX.1-2001 and POSIX.1-2008:

```
pthread_cancel()
```

```
pthread_setcancelstate()
pthread_setcanceltype()
```

Cancellation points

POSIX.1 specifies that certain functions must, and certain other functions may, be cancelation points. If a thread is cancelable, its cancelability type is deferred, and a cancelation request is pending for the thread, then the thread is canceled when it calls a function that is a cancelation point.

The following functions are required to be cancelation points by POSIX.1-2001 and/or POSIX.1-2008:

```
accept()
aio_suspend()
clock_nanosleep()
close()
connect()
creat()
fcntl() F_SETLKW
fdatasync()
fsync()
getmsg()
getpmsg()
lockf() F_LOCK
mq_receive()
mq_send()
mq_timedreceive()
mq_timedsend()
msgrcv()
msgsnd()
msync()
nanosleep()
open()
openat() [Added in POSIX.1-2008]
pause()
poll()
pread()
pselect()
pthread_cond_timedwait()
pthread_cond_wait()
pthread_join()
pthread_testcancel()
putmsg()
putpmsg()
pwrite()
read()
readv()
recv()
recvfrom()
recvmsg()
select()
sem_timedwait()
sem_wait()
send()
sendmsg()
sendto()
sigpause() [POSIX.1-2001 only (moves to "may" list in POSIX.1-2008)]
sigsuspend()
```

```
sigtimedwait()
sigwait()
sigwaitinfo()
sleep()
system()
tcdrain()
usleep() [POSIX.1-2001 only (function removed in POSIX.1-2008)]
wait()
waitid()
waitpid()
write()
writev()
```

The following functions may be cancelation points according to POSIX.1-2001 and/or POSIX.1-2008:

```
access()
asctime()
asctime_r()
catclose()
catgets()
catopen()
chmod() [Added in POSIX.1-2008]
chown() [Added in POSIX.1-2008]
closedir()
closelog()
ctermid()
ctime()
ctime_r()
dbm_close()
dbm_delete()
dbm_fetch()
dbm_nextkey()
dbm_open()
dbm_store()
dlclose()
dlopen()
dprintf() [Added in POSIX.1-2008]
endgrent()
endhostent()
endnetent()
endprotoent()
endpwent()
endservent()
endutxent()
faccessat() [Added in POSIX.1-2008]
fchmod() [Added in POSIX.1-2008]
fchmodat() [Added in POSIX.1-2008]
fchown() [Added in POSIX.1-2008]
fchownat() [Added in POSIX.1-2008]
fclose()
fcntl() (for any value of cmd argument)
fflush()
fgetc()
fgetpos()
fgets()
fgetwc()
```

```
fgetws()
fmtmsg()
fopen()
fpathconf()
fprintf()
fputc()
fputs()
fputwc()
fputws()
fread()
freopen()
fscanf()
fseek()
fseeko()
fsetpos()
fstat()
fstatat() [Added in POSIX.1-2008]
ftell()
ftello()
ftw()
futimens() [Added in POSIX.1-2008]
fwprintf()
fwrite()
fwscanf()
getaddrinfo()
getc()
getc_unlocked()
getchar()
getchar_unlocked()
getcwd()
getdate()
getdelim() [Added in POSIX.1-2008]
getgrent()
getgrgid()
getgrgid_r()
getgrnam()
getgrnam_r()
gethostbyaddr() [POSIX.1-2001 only (function removed in
                 POSIX.1-2008)]
gethostbyname() [POSIX.1-2001 only (function removed in
                 POSIX.1-2008)]
gethostent()
gethostid()
gethostname()
getline() [Added in POSIX.1-2008]
getlogin()
getlogin_r()
getnameinfo()
getnetbyaddr()
getnetbyname()
getnetent()
getopt() (if opterr is nonzero)
getprotobyname()
getprotobynumber()
```

```
getprotoent()
getpwent()
getpwnam()
getpwnam_r()
getpwuid()
getpwuid_r()
gets()
getservbyname()
getservbyport()
getservent()
getutxent()
getutxid()
getutxline()
getwc()
getwchar()
getwd() [POSIX.1-2001 only (function removed in POSIX.1-2008)]
glob()
iconv_close()
iconv_open()
ioctl()
link()
linkat() [Added in POSIX.1-2008]
lio_listio() [Added in POSIX.1-2008]
localtime()
localtime_r()
lockf() [Added in POSIX.1-2008]
lseek()
lstat()
mkdir() [Added in POSIX.1-2008]
mkdirat() [Added in POSIX.1-2008]
mkdtemp() [Added in POSIX.1-2008]
mkfifo() [Added in POSIX.1-2008]
mkfifoat() [Added in POSIX.1-2008]
mknod() [Added in POSIX.1-2008]
mknodat() [Added in POSIX.1-2008]
mkstemp()
mktime()
nftw()
opendir()
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() [Added in POSIX.1-2008]
psignal() [Added in POSIX.1-2008]
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() [Added in POSIX.1-2008]
readlinkat() [Added in POSIX.1-2008]
remove()
rename()
renameat() [Added in POSIX.1-2008]
rewind()
rewinddir()
scandir() [Added in POSIX.1-2008]
scanf()
seekdir()
semop()
setgrent()
sethostent()
setnetent()
setprotoent()
setpwent()
setservent()
setutxent()
sigpause() [Added in POSIX.1-2008]
stat()
strerror()
strerror_r()
strftime()
symlink()
```

```
symlinkat() [Added in POSIX.1-2008]
sync()
syslog()
tmpfile()
tmpnam()
ttyname()
ttyname_r()
tzset()
ungetc()
ungetwc()
unlink()
unlinkat() [Added in POSIX.1-2008]
utime() [Added in POSIX.1-2008]
utimensat() [Added in POSIX.1-2008]
utimes() [Added in POSIX.1-2008]
vdprintf() [Added in POSIX.1-2008]
vfprintf()
vfwprintf()
vprintf()
vwprintf()
wcsftime()
wordexp()
wprintf()
wscanf()
```

An implementation may also mark other functions not specified in the standard as cancelation points. In particular, an implementation is likely to mark any nonstandard function that may block as a cancelation point. (This includes most functions that can touch files.)

It should be noted that even if an application is not using asynchronous cancelation, that calling a function from the above list from an asynchronous signal handler may cause the equivalent of asynchronous cancelation. The underlying user code may not expect asynchronous cancelation and the state of the user data may become inconsistent. Therefore signals should be used with caution when entering a region of deferred cancelation.

Compiling on Linux

On Linux, programs that use the Pthreads API should be compiled using *cc −pthread*.

Linux implementations of POSIX threads

Over time, two threading implementations have been provided by the GNU C library on Linux:

LinuxThreads

This is the original Pthreads implementation. Since glibc 2.4, this implementation is no longer supported.

NPTL (Native POSIX Threads Library)

This is the modern Pthreads implementation. By comparison with LinuxThreads, NPTL provides closer conformance to the requirements of the POSIX.1 specification and better performance when creating large numbers of threads. NPTL is available since glibc 2.3.2, and requires features that are present in the Linux 2.6 kernel.

Both of these are so-called 1:1 implementations, meaning that each thread maps to a kernel scheduling entity. Both threading implementations employ the Linux **clone**(2) system call. In NPTL, thread synchronization primitives (mutexes, thread joining, and so on) are implemented using the Linux **futex**(2) system call.

LinuxThreads

The notable features of this implementation are the following:

- In addition to the main (initial) thread, and the threads that the program creates using **pthread_create**(3), the implementation creates a "manager" thread. This thread handles thread creation and termination. (Problems can result if this thread is inadvertently killed.)
- Signals are used internally by the implementation. On Linux 2.2 and later, the first three real-time signals are used (see also **signal**(7)). On older Linux kernels, **SIGUSR1** and **SIGUSR2** are used. Applications must avoid the use of whichever set of signals is employed by the implementation.
- Threads do not share process IDs. (In effect, LinuxThreads threads are implemented as processes which share more information than usual, but which do not share a common process ID.) LinuxThreads threads (including the manager thread) are visible as separate processes using **ps**(1).

The LinuxThreads implementation deviates from the POSIX.1 specification in a number of ways, including the following:

- Calls to **getpid**(2) return a different value in each thread.
- Calls to **getppid**(2) in threads other than the main thread return the process ID of the manager thread; instead **getppid**(2) in these threads should return the same value as **getppid**(2) in the main thread.
- When one thread creates a new child process using **fork**(2), any thread should be able to **wait**(2) on the child. However, the implementation allows only the thread that created the child to **wait**(2) on it.
- When a thread calls **execve**(2), all other threads are terminated (as required by POSIX.1). However, the resulting process has the same PID as the thread that called **execve**(2): it should have the same PID as the main thread.
- Threads do not share user and group IDs. This can cause complications with set-user-ID programs and can cause failures in Pthreads functions if an application changes its credentials using **seteuid**(2) or similar.
- Threads do not share a common session ID and process group ID.
- Threads do not share record locks created using **fcntl**(2).
- The information returned by **times**(2) and **getrusage**(2) is per-thread rather than process-wide.
- Threads do not share semaphore undo values (see **semop**(2)).
- Threads do not share interval timers.
- Threads do not share a common nice value.
- POSIX.1 distinguishes the notions of signals that are directed to the process as a whole and signals that are directed to individual threads. According to POSIX.1, a process-directed signal (sent using **kill**(2), for example) should be handled by a single, arbitrarily selected thread within the process. LinuxThreads does not support the notion of process-directed signals: signals may be sent only to specific threads.
- Threads have distinct alternate signal stack settings. However, a new thread's alternate signal stack settings are copied from the thread that created it, so that the threads initially share an alternate signal stack. (A new thread should start with no alternate signal stack defined. If two threads handle signals on their shared alternate signal stack at the same time, unpredictable program failures are likely to occur.)

NPTL

With NPTL, all of the threads in a process are placed in the same thread group; all members of a thread group share the same PID. NPTL does not employ a manager thread.

NPTL makes internal use of the first two real-time signals; these signals cannot be used in applications. See **nptl**(7) for further details.

NPTL still has at least one nonconformance with POSIX.1:

- Threads do not share a common nice value.

Some NPTL nonconformances occur only with older kernels:

- The information returned by **times**(2) and **getrusage**(2) is per-thread rather than process-wide (fixed in Linux 2.6.9).
- Threads do not share resource limits (fixed in Linux 2.6.10).
- Threads do not share interval timers (fixed in Linux 2.6.12).
- Only the main thread is permitted to start a new session using **setsid**(2) (fixed in Linux 2.6.16).
- Only the main thread is permitted to make the process into a process group leader using **setpgid**(2) (fixed in Linux 2.6.16).
- Threads have distinct alternate signal stack settings. However, a new thread's alternate signal stack settings are copied from the thread that created it, so that the threads initially share an alternate signal stack (fixed in Linux 2.6.16).

Note the following further points about the NPTL implementation:

- If the stack size soft resource limit (see the description of **RLIMIT_STACK** in **setrlimit**(2)) is set to a value other than *unlimited*, then this value defines the default stack size for new threads. To be effective, this limit must be set before the program is executed, perhaps using the *ulimit −s* shell built-in command (*limit stacksize* in the C shell).

Determining the threading implementation

Since glibc 2.3.2, the **getconf**(1) command can be used to determine the system's threading implementation, for example:

```
bash$ getconf GNU_LIBPTHREAD_VERSION
NPTL 2.3.4
```

With older glibc versions, a command such as the following should be sufficient to determine the default threading implementation:

```
bash$ $( ldd /bin/ls | grep libc.so | awk '{print $3}' ) | \
                egrep −i 'threads|nptl'
        Native POSIX Threads Library by Ulrich Drepper et al
```

Selecting the threading implementation: LD_ASSUME_KERNEL

On systems with a glibc that supports both LinuxThreads and NPTL (i.e., glibc 2.3.*x*), the **LD_ASSUME_KERNEL** environment variable can be used to override the dynamic linker's default choice of threading implementation. This variable tells the dynamic linker to assume that it is running on top of a particular kernel version. By specifying a kernel version that does not provide the support required by NPTL, we can force the use of LinuxThreads. (The most likely reason for doing this is to run a (broken) application that depends on some nonconformant behavior in LinuxThreads.) For example:

```
bash$ $( LD_ASSUME_KERNEL=2.2.5 ldd /bin/ls | grep libc.so | \
                awk '{print $3}' ) | egrep −i 'threads|nptl'
        linuxthreads−0.10 by Xavier Leroy
```

SEE ALSO

clone(2), **fork**(2), **futex**(2), **gettid**(2), **proc**(5), **attributes**(7), **futex**(7), **nptl**(7), **sigevent**(7), **signal**(7)

Various Pthreads manual pages, for example: **pthread_atfork**(3), **pthread_attr_init**(3), **pthread_cancel**(3), **pthread_cleanup_push**(3), **pthread_cond_signal**(3), **pthread_cond_wait**(3), **pthread_create**(3), **pthread_detach**(3), **pthread_equal**(3), **pthread_exit**(3), **pthread_key_create**(3), **pthread_kill**(3), **pthread_mutex_lock**(3), **pthread_mutex_unlock**(3), **pthread_mutexattr_destroy**(3), **pthread_mutexattr_init**(3), **pthread_once**(3), **pthread_spin_init**(3), **pthread_spin_lock**(3), **pthread_rwlockattr_setkind_np**(3), **pthread_setcancelstate**(3), **pthread_setcanceltype**(3), **pthread_setspecific**(3), **pthread_sigmask**(3), **pthread_sigqueue**(3), and **pthread_testcancel**(3)