TSP defined for SCHED_FIFO, SCHED_RR, and SCHED_SPORADIC shall be used in an implementation-defined manner. Each thread with system scheduling contention scope competes for the processors in its scheduling allocation domain in an implementation-defined manner according to its priority. Threads with process scheduling contention scope are scheduled relative to other threads within the same scheduling contention scope in the process.

TSP If _POSIX_THREAD_SPORADIC_SERVER is defined, the rules defined for SCHED_SPORADIC in Scheduling Policies (on page 501) shall be used in an implementation-defined manner for application threads whose scheduling allocation domain size is greater than one.

Scheduling Documentation

If _POSIX_PRIORITY_SCHEDULING is defined, then any scheduling policies beyond
TSP SCHED_OTHER, SCHED_FIFO, SCHED_RR, and SCHED_SPORADIC, as well as the effects of the scheduling policies indicated by these other values, and the attributes required in order to support such a policy, are implementation-defined. Furthermore, the implementation shall document the effect of all processor scheduling allocation domain values supported for these policies.

2.9.5 Thread Cancellation

The thread cancellation mechanism allows a thread to terminate the execution of any other thread in the process in a controlled manner. The target thread (that is, the one that is being canceled) is allowed to hold cancellation requests pending in a number of ways and to perform application-specific cleanup processing when the notice of cancellation is acted upon.

Cancellation is controlled by the cancellation control functions. Each thread maintains its own cancelability state. Cancellation may only occur at cancellation points or when the thread is asynchronously cancelable.

The thread cancellation mechanism described in this section depends upon programs having set *deferred* cancelability state, which is specified as the default. Applications shall also carefully follow static lexical scoping rules in their execution behavior. For example, use of *setjmp*( ), *return*, *goto*, and so on, to leave user-defined cancellation scopes without doing the necessary scope pop operation results in undefined behavior.

Use of asynchronous cancelability while holding resources which potentially need to be released may result in resource loss. Similarly, cancellation scopes may only be safely manipulated (pushed and popped) when the thread is in the *deferred* or *disabled* cancelability states.

2.9.5.1 Cancelability States

The cancelability state of a thread determines the action taken upon receipt of a cancellation request. The thread may control cancellation in a number of ways.

Each thread maintains its own cancelability state, which may be encoded in two bits:

1. Cancelability-Enable: When cancelability is PTHREAD_CANCEL_DISABLE (as defined in the Base Definitions volume of POSIX.1-2008, **<pthread.h>**), cancellation requests against the target thread are held pending. By default, cancelability is set to PTHREAD_CANCEL_ENABLE (as defined in **<pthread.h>**).

2. Cancelability Type: When cancelability is enabled and the cancelability type is PTHREAD_CANCEL_ASYNCHRONOUS (as defined in **<pthread.h>**), new or pending cancellation requests may be acted upon at any time. When cancelability is enabled and the cancelability type is PTHREAD_CANCEL_DEFERRED (as defined in **<pthread.h>**),

cancellation requests are held pending until a cancellation point (see below) is reached. If cancelability is disabled, the setting of the cancelability type has no immediate effect as all cancellation requests are held pending; however, once cancelability is enabled again the new type is in effect. The cancelability type is PTHREAD_CANCEL_DEFERRED in all newly created threads including the thread in which *main*( ) was first invoked.

2.9.5.2 *Cancellation Points*

Cancellation points shall occur when a thread is executing the following functions:

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

† When the *cmd* argument is F_SETLKW.

†† When the *function* argument is F_LOCK.

A cancellation point may also occur when a thread is executing the following functions:

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

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

An implementation shall not introduce cancellation points into any other functions specified in this volume of POSIX.1-2008.

The side-effects of acting upon a cancellation request while suspended during a call of a function are the same as the side-effects that may be seen in a single-threaded program when a call to a function is interrupted by a signal and the given function returns [EINTR]. Any such side-effects occur before any cancellation cleanup handlers are called.

Whenever a thread has cancelability enabled and a cancellation request has been made with that thread as the target, and the thread then calls any function that is a cancellation point (such as *pthread_testcancel*( ) or *read*( )), the cancellation request shall be acted upon before the function returns. If a thread has cancelability enabled and a cancellation request is made with the thread as a target while the thread is suspended at a cancellation point, the thread shall be awakened and the cancellation request shall be acted upon. It is unspecified whether the cancellation request is acted upon or whether the cancellation request remains pending and the thread resumes normal execution if:

- The thread is suspended at a cancellation point and the event for which it is waiting occurs

† For any value of the *cmd* argument.

†† If *opterr* is non-zero.

- A specified timeout expired

before the cancellation request is acted upon.

2.9.5.3 *Thread Cancellation Cleanup Handlers*

Each thread maintains a list of cancellation cleanup handlers. The programmer uses the *pthread_cleanup_push*( ) and *pthread_cleanup_pop*( ) functions to place routines on and remove routines from this list.

When a cancellation request is acted upon, or when a thread calls *pthread_exit*( ), the thread first disables cancellation by setting its cancelability state to PTHREAD_CANCEL_DISABLE and its cancelability type to PTHREAD_CANCEL_DEFERRED. The cancelability state shall remain set to PTHREAD_CANCEL_DISABLE until the thread has terminated. The behavior is undefined if a cancellation cleanup handler or thread-specific data destructor routine changes the cancelability state to PTHREAD_CANCEL_ENABLE.

The routines in the thread's list of cancellation cleanup handlers are invoked one by one in LIFO sequence; that is, the last routine pushed onto the list (Last In) is the first to be invoked (First Out). When the cancellation cleanup handler for a scope is invoked, the storage for that scope remains valid. If the last cancellation cleanup handler returns, thread-specific data destructors (if any) associated with thread-specific data keys for which the thread has non-NULL values will be run, in unspecified order, as described for *pthread_key_create*( ).

After all cancellation cleanup handlers and thread-specific data destructors have returned, thread execution is terminated. If the thread has terminated because of a call to *pthread_exit*( ), the *value_ptr* argument is made available to any threads joining with the target. If the thread has terminated by acting on a cancellation request, a status of PTHREAD_CANCELED is made available to any threads joining with the target. The symbolic constant PTHREAD_CANCELED expands to a constant expression of type (**void** *) whose value matches no pointer to an object in memory nor the value NULL.

A side-effect of acting upon a cancellation request while in a condition variable wait is that the mutex is re-acquired before calling the first cancellation cleanup handler. In addition, the thread is no longer considered to be waiting for the condition and the thread shall not have consumed any pending condition signals on the condition.

A cancellation cleanup handler cannot exit via *longjmp*( ) or *siglongjmp*( ).

2.9.5.4 *Async-Cancel Safety*

The *pthread_cancel*( ), *pthread_setcancelstate*( ), and *pthread_setcanceltype*( ) functions are defined to be async-cancel safe.

No other functions in this volume of POSIX.1-2008 are required to be async-cancel-safe.

2.9.6 Thread Read-Write Locks

Multiple readers, single writer (read-write) locks allow many threads to have simultaneous read-only access to data while allowing only one thread to have exclusive write access at any given time. They are typically used to protect data that is read more frequently than it is changed.

One or more readers acquire read access to the resource by performing a read lock operation on the associated read-write lock. A writer acquires exclusive write access by performing a write lock operation. Basically, all readers exclude any writers and a writer excludes all readers and

