32

THREADS: THREAD CANCELLATION

Typically, multiple threads execute in parallel, with each thread performing its task until it decides to terminate by calling *pthread_exit()* or returning from the thread's start function.

Sometimes, it can be useful to *cancel* a thread; that is, to send it a request asking it to terminate now. This could be useful, for example, if a group of threads is performing a calculation, and one thread detects an error condition that requires the other threads to terminate. Alternatively, a GUI-driven application may provide a cancel button to allow the user to terminate a task that is being performed by a thread in the background; in this case, the main thread (controlling the GUI) needs to tell the background thread to terminate.

In this chapter, we describe the POSIX threads cancellation mechanism.

32.1 Canceling a Thread

The *pthread_cancel()* function sends a cancellation request to the specified *thread*.

```
#include <pthread.h>

int pthread_cancel(pthread_t thread);
```

Returns 0 on success, or a positive error number on error

Having made the cancellation request, *pthread_cancel()* returns immediately; that is, it doesn't wait for the target thread to terminate.

Precisely what happens to the target thread, and when it happens, depends on that thread's cancellation state and type, as described in the next section.

32.2 Cancellation State and Type

The *pthread_setcancelstate()* and *pthread_setcanceltype()* functions set flags that allow a thread to control how it responds to a cancellation request.

```
#include <pthread.h>

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

Both return 0 on success, or a positive error number on error

The *pthread_setcancelstate()* function sets the calling thread's cancelability state to the value given in *state*. This argument has one of the following values:

`PTHREAD_CANCEL_DISABLE`
: The thread is not cancelable. If a cancellation request is received, it remains pending until cancelability is enabled.

`PTHREAD_CANCEL_ENABLE`
: The thread is cancelable. This is the default cancelability state in newly created threads.

The thread's previous cancelability state is returned in the location pointed to by *oldstate*.

> If we are not interested in the previous cancelability state, Linux allows *oldstate* to be specified as `NULL`. This is the case on many other implementations as well; however, SUSv3 doesn't specify this feature, so portable applications can't rely on it. We should always specify a non-`NULL` value for *oldstate*.

Temporarily disabling cancellation (`PTHREAD_CANCEL_DISABLE`) is useful if a thread is executing a section of code where *all* of the steps must be completed.

If a thread is cancelable (`PTHREAD_CANCEL_ENABLE`), then the treatment of a cancellation request is determined by the thread's cancelability type, which is specified by the *type* argument in a call to *pthread_setcanceltype()*. This argument has one of the following values:

`PTHREAD_CANCEL_ASYNCHRONOUS`
: The thread may be canceled at any time (perhaps, but not necessarily, immediately). Asynchronous cancelability is rarely useful, and we defer discussion of it until Section 32.6.

`PTHREAD_CANCEL_DEFERRED`
: The cancellation remains pending until a cancellation point (see the next section) is reached. This is the default cancelability type in newly created threads. We say more about deferred cancelability in the following sections.

The thread's previous cancelability type is returned in the location pointed to by *oldtype*.

> As with the *pthread_setcancelstate()* *oldstate* argument, many implementations, including Linux, allow *oldtype* to be specified as `NULL` if we are not interested in the previous cancelability type. Again, SUSv3 doesn't specify this feature, and portable applications can't rely on it We should always specify a non-`NULL` value for *oldtype*.

When a thread calls *fork()*, the child inherits the calling thread's cancelability type and state. When a thread calls *exec()*, the cancelability type and state of the main thread of the new program are reset to `PTHREAD_CANCEL_ENABLE` and `PTHREAD_CANCEL_DEFERRED`, respectively.

32.3 Cancellation Points

When cancelability is enabled and deferred, a cancellation request is acted upon only when a thread next reaches a *cancellation point*. A cancellation point is a call to one of a set of functions defined by the implementation.

SUSv3 specifies that the functions shown in Table 32-1 *must* be cancellation points if they are provided by an implementation. Most of these are functions that are capable of blocking the thread for an indefinite period of time.

Table 32-1: Functions required to be cancellation points by SUSv3

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

In addition to the functions in Table 32-1, SUSv3 specifies a larger group of functions that an implementation *may* define as cancellation points. These include the *stdio* functions, the *dlopen* API, the *syslog* API, *nftw()*, *popen()*, *semop()*, *unlink()*, and various functions that retrieve information from system files such as the *utmp* file. A portable program must correctly handle the possibility that a thread may be canceled when calling these functions.

SUSv3 specifies that aside from the two lists of functions that must and may be cancellation points, none of the other functions in the standard may act as cancellation points (i.e., a portable program doesn't need to handle the possibility that calling these other functions could precipitate thread cancellation).

SUSv4 adds *openat()* to the list of functions that must be cancellation points, and removes *sigpause()* (it moves to the list of functions that *may* be cancellation points) and *usleep()* (which is dropped from the standard).

> An implementation is free to mark additional functions that are not specified in the standard as cancellation points. Any function that might block (perhaps because it might access a file) is a likely candidate to be a cancellation point. Within *glibc*, many nonstandard functions are marked as cancellation points for this reason.

Upon receiving a cancellation request, a thread whose cancelability is enabled and deferred terminates when it next reaches a cancellation point. If the thread was not detached, then some other thread in the process must join with it, in order to prevent it from becoming a zombie thread. When a canceled thread is joined, the value returned in the second argument to *pthread_join()* is a special thread return value: `PTHREAD_CANCELED`.

Example program

Listing 32-1 shows a simple example of the use of *pthread_cancel()*. The main program creates a thread that executes an infinite loop, sleeping for a second and printing the value of a loop counter. (This thread will terminate only if it is sent a cancellation request or if the process exits.) Meanwhile, the main program sleeps for 3 seconds, and then sends a cancellation request to the thread that it created. When we run this program, we see the following:

```
$ ./t_pthread_cancel
New thread started
Loop 1
Loop 2
Loop 3
Thread was canceled
```

Listing 32-1: Canceling a thread with *pthread_cancel()*

threads/thread_cancel.c

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

static void *
threadFunc(void *arg)
{
    int j;
    printf("New thread started\n");     /* May be a cancellation point */
    for (j = 1; ; j++) {
        printf("Loop %d\n", j);         /* May be a cancellation point */
        sleep(1);                       /* A cancellation point */
    }

    /* NOTREACHED */
    return NULL;
}

int
main(int argc, char *argv[])
{
    pthread_t thr;
    int s;
    void *res;

    s = pthread_create(&thr, NULL, threadFunc, NULL);
    if (s != 0)
        errExitEN(s, "pthread_create");

    sleep(3);                           /* Allow new thread to run a while */

    s = pthread_cancel(thr);
    if (s != 0)
        errExitEN(s, "pthread_cancel");

    s = pthread_join(thr, &res);
    if (s != 0)
        errExitEN(s, "pthread_join");

    if (res == PTHREAD_CANCELED)
        printf("Thread was canceled\n");
    else
        printf("Thread was not canceled (should not happen!)\n");

    exit(EXIT_SUCCESS);
}
```

threads/thread_cancel.c

32.4 Testing for Thread Cancellation

In Listing 32-1, the thread created by *main()* accepted the cancellation request because it executed a function that was a cancellation point (*sleep()* is a cancellation point; *printf()* may be one). However, suppose a thread executes a loop that contains no cancellation points (e.g., a compute-bound loop). In this case, the thread would never honor the cancellation request.

The purpose of *pthread_testcancel()* is simply to be a cancellation point. If a cancellation is pending when this function is called, then the calling thread is terminated.

```
#include <pthread.h>

void pthread_testcancel(void);
```

A thread that is executing code that does not otherwise include cancellation points can periodically call *pthread_testcancel()* to ensure that it responds in a timely fashion to a cancellation request sent by another thread.

32.5 Cleanup Handlers

If a thread with a pending cancellation were simply terminated when it reached a cancellation point, then shared variables and Pthreads objects (e.g., mutexes) might be left in an inconsistent state, perhaps causing the remaining threads in the process to produce incorrect results, deadlock, or crash. To get around this problem, a thread can establish one or more *cleanup handlers*—functions that are automatically executed if the thread is canceled. A cleanup handler can perform tasks such as modifying the values of global variables and unlocking mutexes before the thread is terminated.

Each thread can have a stack of cleanup handlers. When a thread is canceled, the cleanup handlers are executed working down from the top of the stack; that is, the most recently established handler is called first, then the next most recently established, and so on. When all of the cleanup handlers have been executed, the thread terminates.

The *pthread_cleanup_push()* and *pthread_cleanup_pop()* functions respectively add and remove handlers on the calling thread's stack of cleanup handlers.

```
#include <pthread.h>

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

The *pthread_cleanup_push()* function adds the function whose address is specified in *routine* to the top of the calling thread's stack of cleanup handlers. The *routine* argument is a pointer to a function that has the following form:

```
void
routine(void *arg)
{
    /* Code to perform cleanup */
}
```

The *arg* value given to *pthread_cleanup_push()* is passed as the argument of the cleanup handler when it is invoked. This argument is typed as *void **, but, using judicious casting, other data types can be passed in this argument.

Typically, a cleanup action is needed only if a thread is canceled during the execution of a particular section of code. If the thread reaches the end of that section without being canceled, then the cleanup action is no longer required. Thus, each call to *pthread_cleanup_push()* has an accompanying call to *pthread_cleanup_pop()*. This function removes the topmost function from the stack of cleanup handlers. If the *execute* argument is nonzero, the handler is also executed. This is convenient if we want to perform the cleanup action even if the thread was not canceled.

Although we have described *pthread_cleanup_push()* and *pthread_cleanup_pop()* as functions, SUSv3 permits them to be implemented as macros that expand to statement sequences that include an opening ({) and closing (}) brace, respectively. Not all UNIX implementations do things this way, but Linux and many others do. This means that each use of *pthread_cleanup_push()* must be paired with exactly one corresponding *pthread_cleanup_pop()* in the same lexical block. (On implementations that do things this way, variables declared between the *pthread_cleanup_push()* and *pthread_cleanup_pop()* will be limited to that lexical scope.) For example, it is not correct to write code such as the following:

```
pthread_cleanup_push(func, arg);
...
if (cond) {
    pthread_cleanup_pop(0);
}
```

As a coding convenience, any cleanup handlers that have not been popped are also executed automatically if a thread terminates by calling *pthread_exit()* (but not if it does a simple `return`).

Example program

The program in Listing 32-2 provides a simple example of the use of a cleanup handler. The main program creates a thread ⑧ whose first actions are to allocate a block of memory ③ whose location is stored in *buf*, and then lock the mutex *mtx* ④. Since the thread may be canceled, it uses *pthread_cleanup_push()* ⑤ to install a cleanup handler that is called with the address stored in *buf*. If it is invoked, the cleanup handler deallocates the freed memory ① and unlocks the mutex ②.

The thread then enters a loop waiting for the condition variable *cond* to be signaled ⑥. This loop will terminate in one of two ways, depending on whether the program is supplied with a command-line argument:

- If no command-line argument is supplied, the thread is canceled by *main()* ⑨. In this case, cancellation will occur at the call to *pthread_cond_wait()* ⑥, which is one of the cancellation points shown in Table 32-1. As part of cancellation, the cleanup handler established using *pthread_cleanup_push()* is invoked automatically.
- If a command-line argument is supplied, the condition variable is signaled ⑩ after the associated global variable, *glob*, is first set to a nonzero value. In this case, the thread falls through to execute *pthread_cleanup_pop()* ⑦, which, given a nonzero argument, also causes the cleanup handler to be invoked.

The main program joins with the terminated thread ⑪, and reports whether the thread was canceled or terminated normally.

Listing 32-2: Using cleanup handlers

threads/thread_cleanup.c

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

   static pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
   static pthread_mutex_t mtx = PTHREAD_MUTEX_INITIALIZER;
   static int glob = 0;                    /* Predicate variable */

   static void     /* Free memory pointed to by 'arg' and unlock mutex */
   cleanupHandler(void *arg)
   {
       int s;

       printf("cleanup: freeing block at %p\n", arg);
①      free(arg);

       printf("cleanup: unlocking mutex\n");
②      s = pthread_mutex_unlock(&mtx);
       if (s != 0)
           errExitEN(s, "pthread_mutex_unlock");
   }

   static void *
   threadFunc(void *arg)
   {
       int s;
       void *buf = NULL;                   /* Buffer allocated by thread */

③      buf = malloc(0x10000);              /* Not a cancellation point */
       printf("thread:  allocated memory at %p\n", buf);

④      s = pthread_mutex_lock(&mtx);       /* Not a cancellation point */
       if (s != 0)
           errExitEN(s, "pthread_mutex_lock");

⑤      pthread_cleanup_push(cleanupHandler, buf);

       while (glob == 0) {
⑥          s = pthread_cond_wait(&cond, &mtx);    /* A cancellation point */
           if (s != 0)
               errExitEN(s, "pthread_cond_wait");
       }

       printf("thread:  condition wait loop completed\n");
⑦      pthread_cleanup_pop(1);             /* Executes cleanup handler */
       return NULL;
   }

   int
   main(int argc, char *argv[])
   {
       pthread_t thr;
       void *res;
       int s;
⑧     s = pthread_create(&thr, NULL, threadFunc, NULL);
      if (s != 0)
          errExitEN(s, "pthread_create");

      sleep(2);                   /* Give thread a chance to get started */

      if (argc == 1) {            /* Cancel thread */
          printf("main:    about to cancel thread\n");
⑨         s = pthread_cancel(thr);
          if (s != 0)
              errExitEN(s, "pthread_cancel");

      } else {                    /* Signal condition variable */
          printf("main:    about to signal condition variable\n");
          glob = 1;
⑩         s = pthread_cond_signal(&cond);
          if (s != 0)
              errExitEN(s, "pthread_cond_signal");
      }

⑪     s = pthread_join(thr, &res);
      if (s != 0)
          errExitEN(s, "pthread_join");
      if (res == PTHREAD_CANCELED)
          printf("main:    thread was canceled\n");
      else
          printf("main:    thread terminated normally\n");

      exit(EXIT_SUCCESS);
  }
```

threads/thread_cleanup.c

If we invoke the program in Listing 32-2 without any command-line arguments, then *main()* calls *pthread_cancel()*, the cleanup handler is invoked automatically, and we see the following:

```
$ ./thread_cleanup
thread:  allocated memory at 0x804b050
main:    about to cancel thread
cleanup: freeing block at 0x804b050
cleanup: unlocking mutex
main:    thread was canceled
```

If we invoke the program with a command-line argument, then *main()* sets *glob* to 1 and signals the condition variable, the cleanup handler is invoked by *pthread_cleanup_pop()*, and we see the following:

```
$ ./thread_cleanup s
thread:  allocated memory at 0x804b050
main:    about to signal condition variable
thread:  condition wait loop completed
cleanup: freeing block at 0x804b050
cleanup: unlocking mutex
main:    thread terminated normally
```

32.6 Asynchronous Cancelability

When a thread is made asynchronously cancelable (cancelability type `PTHREAD_CANCEL_ASYNCHRONOUS`), it may be canceled at any time (i.e., at any machine-language instruction); delivery of a cancellation is not held off until the thread next reaches a cancellation point.

The problem with asynchronous cancellation is that, although cleanup handlers are still invoked, the handlers have no way of determining the state of a thread. In the program in Listing 32-2, which employs the deferred cancelability type, the thread can be canceled only when it executes the call to *pthread_cond_wait()*, which is the only cancellation point. By this time, we know that *buf* has been initialized to point to a block of allocated memory and that the mutex *mtx* has been locked. However, with asynchronous cancelability, the thread could be canceled at any point; for example, before the *malloc()* call, between the *malloc()* call and locking the mutex, or after locking the mutex. The cleanup handler has no way of knowing where cancellation has occurred, or precisely which cleanup steps are required. Furthermore, the thread might even be canceled *during* the *malloc()* call, after which chaos is likely to result (Section 7.1.3).

As a general principle, an asynchronously cancelable thread can't allocate any resources or acquire any mutexes, semaphores, or locks. This precludes the use of a wide range of library functions, including most of the Pthreads functions. (SUSv3 makes exceptions for *pthread_cancel()*, *pthread_setcancelstate()*, and *pthread_setcanceltype()*, which are explicitly required to be *async-cancel-safe*; that is, an implementation must make them safe to call from a thread that is asynchronously cancelable.) In other words, there are few circumstances where asynchronous cancellation is useful. One such circumstance is canceling a thread that is in a compute-bound loop.

32.7 Summary

The *pthread_cancel()* function allows one thread to send another thread a cancellation request, which is a request that the target thread should terminate.

How the target thread reacts to this request is determined by its cancelability state and type. If the cancelability state is currently set to disabled, the request will remain pending until the cancelability state is set to enabled. If cancelability is enabled, the cancelability type determines when the target thread reacts to the request. If the type is deferred, the cancellation occurs when the thread next calls one of a number of functions specified as cancellation points by SUSv3. If the type is asynchronous, cancellation may occur at any time (this is rarely useful).

A thread can establish a stack of cleanup handlers, which are programmer-defined functions that are invoked automatically to perform cleanups (e.g., restoring the states of shared variables, or unlocking mutexes) if the thread is canceled.

Further information

Refer to the sources of further information listed in Section 29.10.