CS 33

Multithreaded Programming III

Outline

- Thread safety
- Unix signals
- Cancellation

Global Variables

```
int IOfunc( ) {
    extern int errno;
    ...
    if (write(fd, buffer, size) == -1) {
      if (errno == EIO)
        fprintf(stderr, "IO problems ...\n");
      ...
      return(0);
    }
    ...
}
```


Unix was not designed with multithreaded programming in mind. A good example of the implications of this is the manner in which error codes for failed system calls are made available to a program: if a system call fails, it returns –1 and the error code is stored in the global variable *errno*. Though this is not all that bad for single-threaded programs, it is plain wrong for multithreaded programs.

Coping

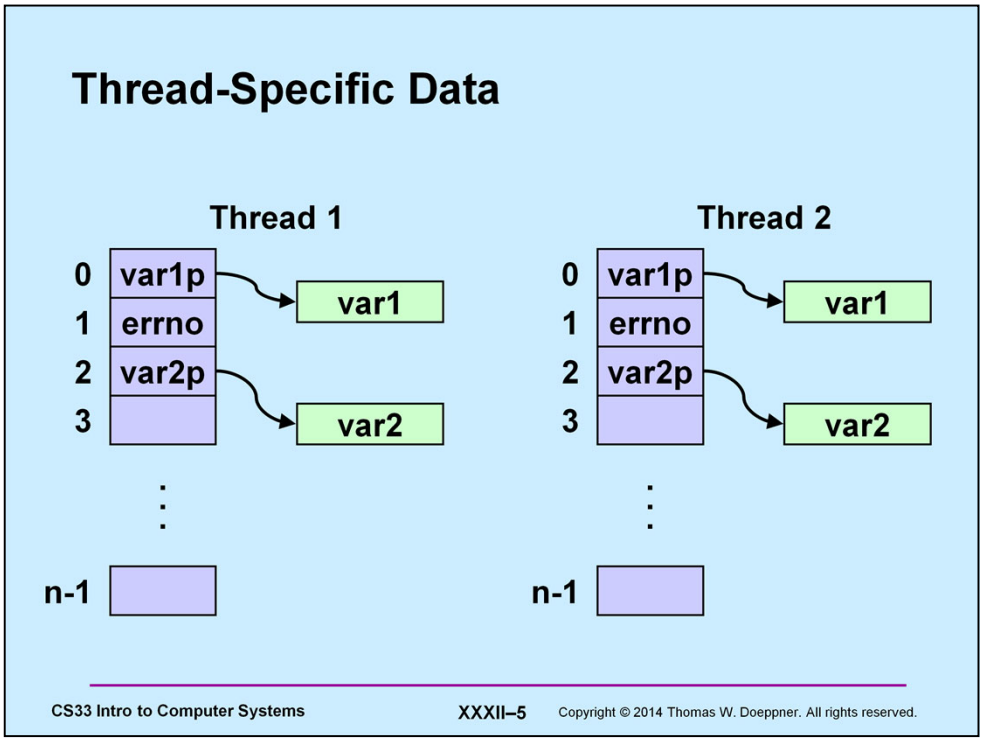
- **Fix Unix's C/system-call interface**
- **Make *errno* refer to a different location in each thread**
  - **acts like a global variable, but private to each thread**
  - **e.g.**
    - » `#define errno __errno[thread_ID]`

The ideal way to solve the "*errno* problem" would be to redesign the C/system-call interface: system calls should return only an error code. Anything else to be returned should be returned via result parameters. (This is how things are done in the Win32 interface.) Unfortunately, this is not possible (it would break pretty much every Unix program in existence).

So we are stuck with *errno*. What can we do to make *errno* coexist with multithreaded programming? What would help would be to arrange, somehow, that each thread has its own private copy of *errno*. I.e., whenever a thread refers to *errno*, it refers to a different location from any other thread when it refers to *errno*.

What can we do? As shown in the slide, we might use the C preprocessor to redefine *errno* as something a bit more complicated — references to *errno* result in accessing an array of *errno*s, one for each thread. This might turn out to be impractical, but we can devise other approaches, as shown in the next slide.

Note: in Linux, when threads are used, errno is defined to be "(*__errno_location( ))" i.e., it dereferences the result of calling a function that retrieves the thread-specific address of errno.



There are enough other situations analogous to *errno* in which each thread should have its own private copy of an otherwise global variable that a general mechanism to deal with it is important. A data item that is global but private to a thread is called *thread-specific data*. Rather than represent each such item as a separate array, as mentioned in the previous slide, what's done in POSIX threads is, effectively, to give each thread an array of pointer-size values (it doesn't have to be done this way, but the effect must be the same). Each element of the array either holds or points to a thread-specific-data (TSD) item. Thus TSD[0] might contain the address of thread-specific-data item *var1*, TSD[1] might contain the value of *errno*, and TSD[2] might contain the address of *var2*. What's important is that whenever a thread refers to the TSD array, it uses its own private array. This may sound like we're begging the question, but, as seen in the next slide, we introduce special machinery for getting at this array.

The TSD Array

- `pthread_key_create(&key, cleanup_routine)`
  - **allocates a slot in the TSD arrays**
  - **provides a function to cleanup when threads terminate**
- `value = pthread_getspecific(key)`
  - **fetches from the calling thread's array**
- `pthread_setspecific(key, value)`
  - **stores into the calling thread's array**

So that we can be certain that it's the calling thread's array that is accessed, rather than access the TSD array directly, one uses a set of POSIX threads library routines. To find an unused slot, one calls *pthread_key_create*, which returns the index of an available slot in its first argument. Its second argument is the address of a routine that's automatically called when the thread terminates, so as to do any cleanup that might be necessary (it's called with the key (index) as its sole argument, and is called only if the thread has actually stored a non-null value into the slot). To put a value in a slot, i.e., perform the equivalent of TSD[i] = x, one calls *pthread_setspecific(i,x).* To fetch from the slot, one calls *pthread_getspecific(i).*

Using Thread-Specific Data (1)

```
pthread_key_t errno_key; // global
pthread_key_create(&errno_key, 0);
   // done by only one thread

int write(…) { // wrapper for syscall
   int err = syscall(WRITE, …);
   if (err)
      pthread_setspecific(errno_key, err);
   …
}
```


Here we see how we might use thread-specific data to implement *errno* in a thread-safe way. First we allocate a slot in the TSD array to hold each thread's copy of *errno*. It's important that this be done only once, but that the slot's index be placed in a global variable (*errno_key* in this case) so that all threads can access it. (Since no cleanup is necessary when threads terminate, we don't supply a cleanup routine.) Next we see a sketch of the write library routine, which is actually a wrapper for an instruction that traps into the kernel to make the actual call. On return from the trap, if there was an error, we put its code into the TSD slot that was allocated for *errno*.

Using Thread-Specific Data (2)

```
#define errno pthread_getspecific(errno_key)
   // make things easy to read and type

if (write(fd, buffer, size) == -1) {
   if (errno == EIO)
      fprintf(stderr, "IO problems ...\n");
   ...
   return(0);
}
```


We repeat our earlier example that used *errno*, however this time we've redefined references to *errno* to use *pthread_getspecific*.

Example (1)

```
#define mystate \
  *(pthread_getspecific(state_key))
pthread_key_t state_key;
void free_state(void *);

void mainline( ) {
  ...
  pthread_key_create(&state_key,
     free_state);
  while (more_clients) {
    ...
    pthread_create(&thread, 0,
        server_start, requestp);
  }
}
```

```
void *server_start(req_t *req) {
  state_t *statep;

  statep = (state_t *)malloc(
     sizeof(*statep));
  init_state(statep);
  pthread_setspecific(state_key,
     statep);
  ...
  handle_request(req);
  ...
  return(0);
}
```


As another example of the use of thread-specific data, consider a server application that creates threads for interacting with each of its clients. Each such thread keeps track of the state of its connection with its client. Each thread mallocs the storage to hold its state, then sets a pointer to this storage in thread-specific data by calling *pthread_setspecific*. Since the storage for this state has been malloc'd, it must be freed when the thread terminates. To ensure that this happens, the call to *pthread_key_create* registers a cleanup function that frees the storage.

To make accesses to the thread-specific data relatively transparent, we define the mystate macro, which calls *pthread_getspecific* to get the pointer to the calling thread's state information, then dereferences it.

Example (2)

```
void handle_request(req_t req) {
   ...

  while(1) {
    switch(mystate) {
       ...
      mystate = ...
  }
  ...
}
```

```
void free_state(state_t *statep) {
   free(statep);
}
```


This is the remainder of the credentials example.

Static Local Storage

```
struct hostent *gethostbyname(const char *name) {
     static struct hostent hostent;

     ... // lookup name; fill in hostent

     return(&hostent);
}
```


Another ramification of Unix's single-thread mentality is the use of static local storage in a number of library routines. An example of this is the *gethostbyname* routine, which, given the name of a network host, returns the address of a data structure that contains information about how to contact that host. This data structure is built on storage that is statically allocated inside the *gethostbyname* routine. This is a reasonable way to do things in the single-threaded world, but it fails miserably in the multithreaded world.

Coping

- Use thread-specific data
- Allocate storage internally; caller frees it
- Redesign the interface

As the slide shows, there are at least three techniques for coping with this problem. We could use thread-specific data, but this would entail associating a fair amount of storage with each thread—perhaps this is viable if just a few bytes are involved, but not for a large data structure such as a *struct hostent* which is used only by the occasional thread. We might simply allocate storage (via *malloc*) inside *gethostbyname* and return a pointer to this storage. The problem with this is that the calls to *malloc* and *free* could turn out to be expensive. Furthermore, this makes it the caller's responsibility to free the storage, introducing a likely storage leak.

The solution taken is to redesign the interface. The "thread-safe" version is called *gethostbyname_r* (the *r* stands for *reentrant*, an earlier term for "thread-safe"); it takes additional parameters that describe storage passed by the caller into which *gethostbyname_r* places the result. Thus the caller is responsible for both the allocation and the liberation of the storage containing the result; this storage is typically a local variable (allocated on the stack), so that its allocation and liberation overhead is negligible, at worst.

However, even *gethostbyname_r* isn't sufficient for dealing with modern networks employing both IPv4 and IPv6. One should now use *getaddrinfo*, which is both thread-safe and general enough for modern networks.

Shared Data

- **Thread 1:**

```
printf("goto statement reached");
```

- **Thread 2:**

```
printf("Hello World\n");
```

- **Printed on display:**

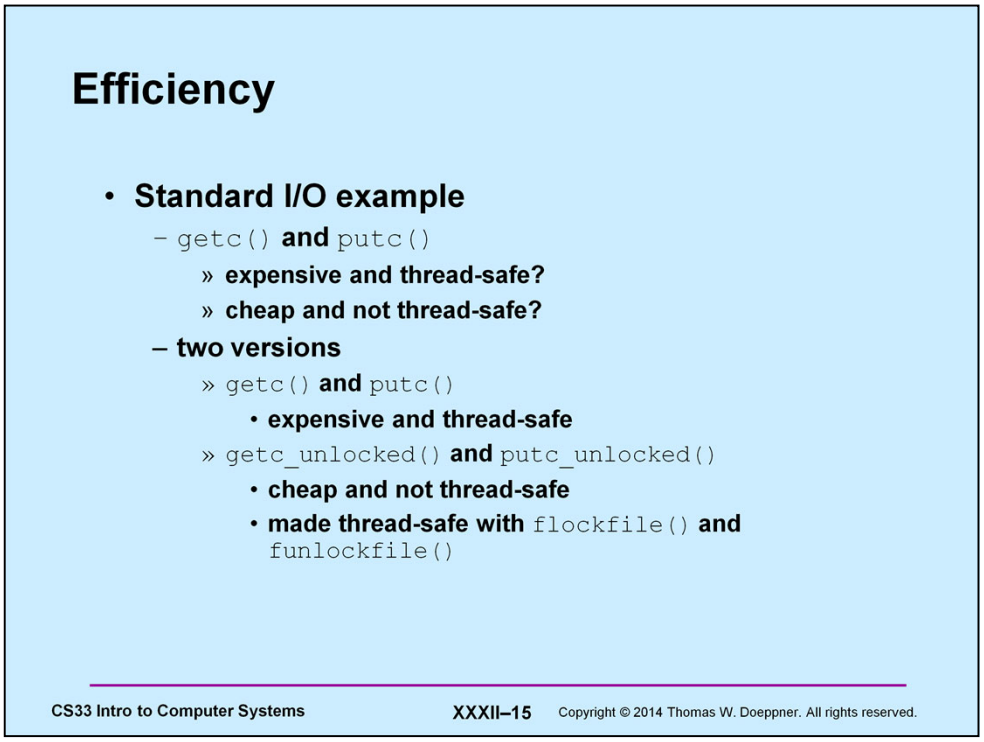
```
go to Hell
```


Yet another problem that arises when using libraries that were not designed for multithreaded programs concerns synchronization. The slide shows what might happen if one relied on the single-threaded versions of the standard I/O routines.

Coping

- **Wrap library calls with synchronization constructs**
- **Fix the libraries**

To deal with this *printf* problem, we must somehow add synchronization to *printf* (and all of the other standard I/O routines). A simple way to do this would be to supply wrappers for all of the standard I/O routines ensuring that only one thread is operating on any particular stream at a time. A better way would be to do the same sort of thing by fixing the routines themselves, rather than supplying wrappers (this is what is done in most implementations).



After making a library thread-safe, we may discover that many routines have become too slow. For example, the standard-I/O routines *getc* and *putc* are normally expected to be fast — they are usually implemented as macros. But once we add the necessary synchronization, they become rather sluggish — much too slow to put in our innermost loops. However, if we are aware of and willing to cope with the synchronization requirements ourselves, we can produce code that is almost as efficient as the single-threaded code without synchronization requirements.

The POSIX-threads specification includes unsynchronized versions of *getc* and *putc* — *getc_unlocked* and *putc_unlocked*. These are exactly the same code as the single-threaded *getc* and *putc*. To use these new routines, one must take care to handle the synchronization oneself. This is accomplished with *flockfile* and *funlockfile*.

Efficiency

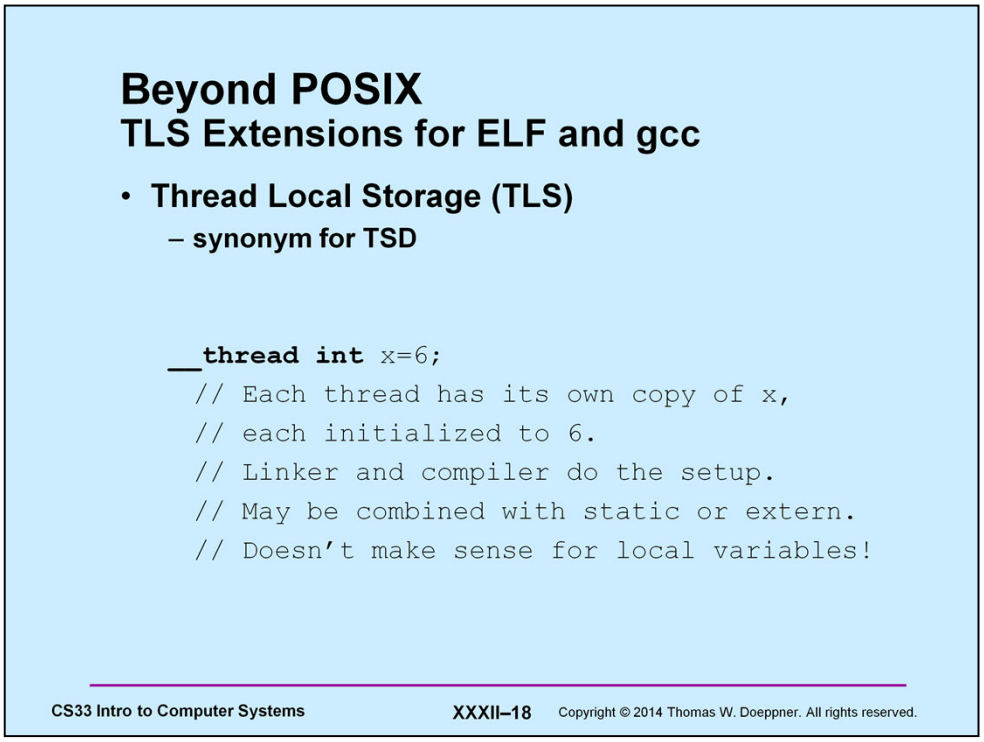
- **Naive**

```
for(i=0; i<lim; i++)
  putc(out[i]);
```

- **Efficient**

```
flockfile(stdout);
for(i=0; i<lim; i++)
  putc_unlocked(out[i]);
funlockfile(stdout);
```

According to IEEE Std. 1003.1 (POSIX), all functions they specify must be thread-safe, except for those listed above.



ELF stands for "executable and linking format" and is the standard format for executable and object files used on most Unix (and other) systems. The *__thread* attribute tells gcc that the item being declared is to be thread-local, which is the same thing as thread-specific. A detailed description of how it is implemented can be found at http://people.redhat.com/drepper/tls.pdf.

Performance Concerns

- Referencing TLS
  - involves finding it: levels of indirection
- Each module
  - potentially defines TLS
- Each thread
  - potentially needs private copy of each module's TLS
- Creating a thread
  - requires setup of TLS
- Loading a module
  - adds TLS to current and future threads

Optimizations

- Allocate/initialize TLS on demand
  - not all threads use all modules
- Short-circuit access to TLS where possible
  - staticly linked modules
  - intra-module access

Deviations

- Signals

vs.

- Cancellation
  - tamed lightning

Deviations are things that modify a thread's normal flow of control. Unix has long had *signals*, and these must be dealt with in multithreaded improvements to Unix. There are actually two fairly different classes of signals: *asynchronous signals* and *synchronous signals*. The former are caused by events beyond the process's control, such as I/O events, clock events, system calls issued by other processes, etc. The latter are responses to what the current thread has just done, such as divide by zero, addressing exceptions, etc.

Cancellation is a new concept that pertains strictly to multithreaded programming. It is the means by which one thread can request the termination of another and provides a way for the terminating thread to terminate cleanly.

Signals

- 
  - who gets them?
  - who needs them?
- 
  - how do you respond to them?

Asynchronous signals were designed (like almost everything else) with single-threaded processes in mind. A signal is delivered to the process; if the signal is *caught*, the process stops whatever it is doing, deals with the signal, and then resumes normal processing. But what happens when a signal is delivered to a multithreaded process? Which thread or threads deal with it?

Asynchronous signals, by their very nature, are handled asynchronously. But one of the themes of multithreaded programming is that threads are a cure for asynchrony. Thus we should be able to use threads as a means of getting away from the "drop whatever you are doing and deal with me" approach to asynchronous signals.

Synchronous signals often are an indication that something has gone wrong: there really is no point continuing execution in this part of the program because you have already blown it. Traditional Unix approaches for dealing with this bad news are not terribly elegant.

Dealing with Signals

- Per-thread signal masks
- Per-process signal vectors
- One delivery per signal

The standard Unix model has a process-wide signal mask and a vector indicating what is to be done in response to each kind of signal. When a signal is delivered to a process, an indication is made that this signal is pending. If the signal is unmasked, then the vector is examined to determine the response: to suspend the process, to resume the process, to terminate the process, to ignore the signal entirely, or to invoke a signal handler.

A number of issues arise in translating this model into a multithreaded-process model. First of all, if we invoke a signal handler, which thread or threads should execute the handler? What seems to be closest to the spirit of the original signal semantics is that exactly one thread should execute the handler. Which one? The consensus is that it really does not matter, just as long as exactly one thread executes the signal handler. But what about the signal mask? Since one sets masks depending on a thread's local behavior, it makes sense for each thread to have its own private signal mask. Thus a signal is delivered to any one thread that has the signal unmasked (if more than one thread has the signal unmasked, a thread is chosen randomly to handle the signal). If all threads have the signal masked, then the signal remains pending until some thread unmasks it.

A related issue is the vector indicating the response to each signal. Should there be one such vector per thread? If so, what if one thread specifies process termination in response to a signal, while another thread supplies a handler? For reasons such as this, it was decided that, even for multithreaded processes, there would continue to be a single, process-wide signal-disposition vector.

Signals and Threads

```
int pthread_kill(pthread_t thread, int signo);
```

– thread equivalent of *kill*

```
int pthread_sigmask(int how,
     const sigset_t *newmask,
     sigset_t oldmask);
```

– (unnecessary) thread equivalent of *sigprocmask*

Signals may be sent to individual threads using *pthread_kill*. Though the targeted thread will handle the signal, the behavior is as set for the entire process (or clone group in Linux) using *sigaction*. Each thread may independently block and unblock signals using *pthread_sigmask*.

Asynchronous Signals (1)

```
main( ) {
   void handler(int);
   signal(SIGINT, handler);

   ...

}

void handler(int sig) {
   ...
}
```


The slide shows the standard approach for dealing with signals: one sets up a handler that's invoked by the thread that received the signal.

Asynchronous Signals (2)

```
main( ) {
  void handler(int);

  signal(SIGINT, handler);

  ...    // complicated program

  printf("important message: "
     "%s\n", message);

  ...    // more program

}
```

```
void handler(int sig) {

  ...    // deal with signal

  printf("equally important "
     "message: %s\n",
  message);
}
```


Here we have the example we saw a few weeks ago of the reason for requiring that signal handlers call only async-signal-safe functions.

Synchronizing Asynchrony

```
computation_state_t  state;
sigset_t set;
main( ) {
  pthread_t thread;

  sigemptyset(&set);
  sigaddset(&set, SIGINT);
  sigprocmask(SIG_BLOCK,
   &set, 0);
  pthread_create(&thread, 0,
   monitor, 0);
  long_running_procedure( );
}
```

```
void *monitor(void *dummy) {
  int sig;
  while (1) {
   sigwait(&set, &sig);
   display(&state);
  }
  return(0);
}
```


Here we use a different technique for dealing with the signal. Rather than have the thread performing the long-running computation be interrupted by the signal, we dedicate a thread to dealing with the signal. We make use of a new signal-handling routine, *sigwait*. This routine puts its caller to sleep until one of the signals specified in its argument occurs, at which point the call returns and the number of the signal that occurred is stored in the location pointed to by the second argument. As is done here, *sigwait* is normally called with the signals of interest masked off; *sigwait* responds to signals even if they are masked. (Note also that a new thread inherits the signal mask of its creator.)

Among the advantages of this approach is that there are no concerns about async-signal safety since a signal handler is never invoked. The signal-handling thread waits for signals synchronously — it is not interrupted. Thus it is safe for it to use even mutexes, condition variables, and semaphores from inside of the *display* routine. Another advantage is that, if this program is run on a multiprocessor, the "signal handling" can run in parallel with the mainline code, which could not happen with the previous approach.

Killing Time ...

```
struct timespec timeout, remaining_time;

timeout.tv_sec = 3;        // seconds
timeout.tv_nsec = 1000;    // nanoseconds

nanosleep(&timeout, &remaining_time);
```


It is sometimes useful for a thread to wait for a certain period of time before continuing. The traditional Unix approach of using alarm and SIGALRM not only is not suitable for multithreaded programming, but also does not provide fine enough granularity. The routine *nanosleep* provides a better approach. A thread calls it with two arguments; the first indicates (in seconds and nanoseconds) how long the thread wishes to wait. The second argument is relevant only if the thread is interrupted by a signal: it indicates how much additional time remains until the originally requested time period expires.

Note that most Unix implementations do not have a clock that measures time in nanoseconds: the first argument to *nanosleep* is rounded up to an integer multiple of whatever sleep resolution is supported.

Timeouts

```
struct timespec relative_timeout, absolute_timeout;
struct timeval now;

relative_timeout.tv_sec = 3;          // seconds
relative_timeout.tv_nsec = 1000;      // nanoseconds
gettimeofday(&now, 0);
absolute_timeout.tv_sec = now.tv_sec + relative_timeout.tv_sec;
absolute_timeout.tv_nsec = 1000*now.tv_usec +
  relative_timeout.tv_nsec;

if (absolute_timeout.tv_nsec >= 1000000000) {   // deal with the carry
  absolute_timeout.tv_nsec -= 1000000000;
  absolute_timeout.tv_sec++;
}
pthread_mutex_lock(&m);
while (!may_continue)
  pthread_cond_timedwait(&cv, &m, &absolute_timeout);
pthread_mutex_unlock(&m);
```


POSIX threads provides a version of *pthread_cond_wait* that has a timeout: *pthread_cond_timedwait*. It takes an additional argument indicating when the thread should give up on being awoken by a *pthread_cond_signal*. This argument is an absolute time, as opposed to a relative time (as used in the previous slide); i.e., it is the clock time at which the call times out. To convert from an relative time to an absolute time, one must perform the machinations shown in the slide (or something similar)—note that *gettimeofday* returns seconds and *microseconds*, whereas *pthread_cond_timedwait* wants seconds and *nanoseconds*.

Why is it done this way? Though at first (and most subsequent) glances it seems foolish to require an absolute timeout value rather than a relative one, the use of the former makes some sense if you keep in mind that *pthread_cond_timedwait* could return with the "may_continue" condition false even before the timeout has expired (either because it's returned spontaneously or because the "may_continue" was falsified after the thread was released from the condition-variable queue). By having the timeout be absolute, there's no need to compute a new relative timeout when *pthread_cond_timedwait* is called again.

Cancellation

In a number of situations one thread must tell another to cease whatever it is doing. For example, suppose we've implemented a chess-playing program by having multiple threads search the solution space for the next move. If one thread has discovered a quick way of achieving a checkmate, it would want to notify the others that they should stop what they're doing, the game has been won.

One might think that this is an ideal use for per-thread signals, but there's a cleaner mechanism for doing this sort of thing in POSIX threads, called *cancellation*.

Sample Code

```
void *thread_code(void *arg) {
  node_t *head = 0;
  while (1) {
    node_t *nodep;
    nodep = (node_t *)malloc(sizeof(node_t));
    if (read(0, &node->value,
        sizeof(node->value)) == 0) {
      free(nodep);
      break;
    }
    nodep->next = head;
    head = nodep;
  }
  return head;
}
```

pthread_cancel(thread);

This code is invoked by a thread (as its first procedure). The thread reads values from stdin, which it then puts on a singly linked list that it allocates on the fly, and returns a pointer to the list.

Suppose our thread is forced to terminate in the midst of its execution (some other thread invokes the operation *pthread_cancel* on it). What sort of problems might ensue?

Cancellation Concerns

- Getting cancelled at an inopportune moment
- Cleaning up

We have two concerns about the forced termination of threads resulting from cancellation: a thread might be in the middle of doing something important that it must complete before self-destructing; and a canceled thread must be given the opportunity to clean up.

Cancellation State

- **Pending cancel**
  - `pthread_cancel(thread)`
- **Cancels enabled or disabled**
  - `int pthread_setcancelstate( {PTHREAD_CANCEL_DISABLE PTHREAD_CANCEL_ENABLE}, &oldstate)`
- **Asynchronous vs. deferred cancels**
  - `int pthread_setcanceltype( {PTHREAD_CANCEL_ASYNCHRONOUS, PTHREAD_CANCEL_DEFERRED}, &oldtype)`

A thread issues a cancel request by calling *pthread_cancel*, supplying the ID of the target thread as the argument. Associated with each thread is some state information known as its *cancellation state* and its *cancellation type*. When a thread receives a cancel request, it is marked indicating that it has a pending cancel. The next issue is when the thread should notice and act upon the cancel. This is governed by the cancellation state: whether cancels are *enabled* or *disabled* and by the cancellation type: whether the response to cancels is *asynchronous* or *deferred*. If cancels are *disabled*, then the cancel remains pending but is otherwise ignored until cancels are enabled. If cancels are *enabled*, they are acted on as soon as they are noticed if the cancellation type is *asynchronous*. Otherwise, i.e., if the cancellation type is *deferred*, the cancel is acted upon only when the thread reaches a *cancellation point*.

Cancellation points are intended to be well defined points in a thread's execution at which it is prepared to be canceled. They include pretty much all system and library calls in which the thread can block, with the exception of *pthread_mutex_lock*. In addition, a thread may call *pthread_testcancel*, which has no function other than being a cancellation point.

The default is that cancels are enabled and deferred. One can change the cancellation state of a thread by using the routines shown in the slide. Calls to *pthread_setcancelstate* and *pthread_setcanceltype* return the previous value of the affected portion of the cancellability state.

Cancellation Points

- aio_suspend
- close
- creat
- fcntl (when F_SETLCKW is the command)
- fsync
- mq_receive
- mq_send
- msync
- nanosleep
- open
- pause
- pthread_cond_wait
- pthread_cond_timedwait
- pthread_join
- pthread_testcancel
- read
- sem_wait
- sigwait
- sigwaitinfo
- sigsuspend
- sigtimedwait
- sleep
- system
- tcdrain
- wait
- waitpid
- write

The slide lists all of the required cancellation points in POSIX. Note that *pthread_mutex_lock* is not a cancellation point. This is because if it were, then its use in any routine would make that routine a cancellation point, which would make pretty much any routine a cancellation point.

The routine *pthread_testcancel* is strictly a cancellation point — it has no other function. If there are no pending cancels when it is called, it does nothing and simply returns.

Cleaning Up

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

When a thread acts upon a cancel, its ultimate fate has been established, but it first gets a chance to clean up. Associated with each thread may be a stack of *cleanup handlers*. Such handlers are pushed onto the stack via calls to *pthread_cleanup_push* and popped off the stack via calls to *pthread_cleanup_pop*. Thus when a thread acts on a cancel or when it calls *pthread_exit*, it calls each of the cleanup handlers in turn, giving the argument that was supplied as the second parameter of *pthread_cleanup_push*. Once all the cleanup handlers have been called, the thread terminates.

The two routines *pthread_cleanup_push* and *pthread_cleanup_pop* are intended to act as left and right parentheses, and thus should always be paired (in fact, they may actually be implemented as macros: the former contains an unmatched "{", the latter an unmatched "}"). The argument to the latter routine indicates whether or not the cleanup function should be called as a side effect of calling *pthread_cleanup_pop*.

Sample Code, Revisited

```
void *thread_code(void *arg) {
  node_t *head = 0;
  pthread_cleanup_push(
    cleanup, &head);
  while (1) {
    node_t *nodep;
    nodep = (node_t *)
      malloc(sizeof(node_t));
    if (read(0, &node->value,
        sizeof(node->value)) == 0) {
      free(nodep);
      break;
    }
    nodep->next = head;
    head = nodep;
  }
  pthread_cleanup_pop(0);
  return head;
}
```

```
void cleanup(void *arg) {
  node_t **headp = arg;
  while(*headp) {
    node_t *nodep = head->next;
    free(*headp);
    *headp = nodep;
  }
}
```


Here we've added a cleanup handler to our sample code, so that the thread closes the socket if it is cancelled. Note that it has just one cancellation point: *read*. The cleanup handler iterates through the list, deleting each element.

Cancellation and Conditions

```
pthread_mutex_lock(&m);
pthread_cleanup_push(
    pthread_mutex_unlock, &m);
while(should_wait)
  pthread_cond_wait(&cv, &m);

// … (code containing other cancellation points)

pthread_cleanup_pop(1);
```


In this example we handle cancels that might occur while a thread is blocked within *pthread_cond_wait*. Again we assume the thread has cancels enabled and deferred. The thread first pushes a cleanup handler on its stack — in this case the cleanup handler unlocks the mutex. The thread then loops, calling *pthread_cond_wait*, a cancellation point. If it receives a cancel, the cleanup handler won't be called until the mutex has been reacquired. Thus we are certain that when the cleanup handler is called, the mutex is locked.

What's important here is that we make sure the thread does not terminate without releasing its lock on the mutex *m*. If the thread acts on a cancel within *pthread_cond_wait* and the cleanup handler were invoked without first taking the mutex, this would be difficult to guarantee, since we wouldn't know if the thread had the mutex locked (and thus needs to unlock it) when it's in the cleanup handler.

Cleaning Up and Asynchrony

```
node_mangler( ) {
  ...
  pthread_setcanceltype(
     CANCEL_ASYNCHRONOUS,
     0);
  pthread_cleanup_push(
     cleanup, node);
  node = makenode(value);
  ...
  pthread_cleanup_pop(0);
}
void cleanup(node_t *node) {
  ...
}
```

```
node_t *makenode(int val) {
  node_t *node = NULL;
  pthread_cleanup_push(
     remove_node, node);
  node = (node_t *)malloc(
     sizeof(*node));
  ...   // initialize node
  pthread_cleanup_pop(0);
  return(node);
}

void remove_node(node_t *node) {
  if (node != NULL)
    free(node);
}
```


Here is another example of the use of cleanup handlers. We have set the response to cancels to be *asynchronous*, established a cleanup handler, and called the routine *makenode*. This routine pushes its own cleanup handler on the stack, calls *malloc*, does some further initialization of the node, and then pops its cleanup handler from the stack and returns to the caller.

Unfortunately, the fact that we have set the cancellation response to be *asynchronous* has made this program unsafe. For example, suppose a cancel occurs while the thread is in *malloc*: memory has been allocated but its address has not yet been assigned to *node*. We enter the cleanup handler, but because *node*'s value is still NULL, no cleanup takes place. Similarly, if a cancel occurs while the thread is in *makenode*, just after it pops its cleanup handler but before it returns, we again cannot clean up the storage that had been allocated.

If, on the other hand, we had left the response to cancels as *deferred*, none of these problems would be present — a cancel would not be acted upon inside *malloc* or just before we return from *makenode*. Similar to the notion of *async-signal safety* there is also a notion of *async-cancel safety*. *Malloc* and *makenode* are examples of routines that are not *async-cancel-safe*: it is not safe to call them when the response to cancels is asynchronous.

As a general rule, asynchronous cancellability should be used sparingly, if at all. It perhaps makes sense to use it only if a thread is performing a long computation that contains no cancellation points and if, of course, it calls only async-cancel-safe routines.

What's Async-Cancel-Safe?

- What can be terminated at an arbitrary moment with no ill effects?
  - pure functions
    - » no modification of anything with a lifetime longer than the function call
    - » no modification of anything visible to other threads
    - » produce a value or an error indication

