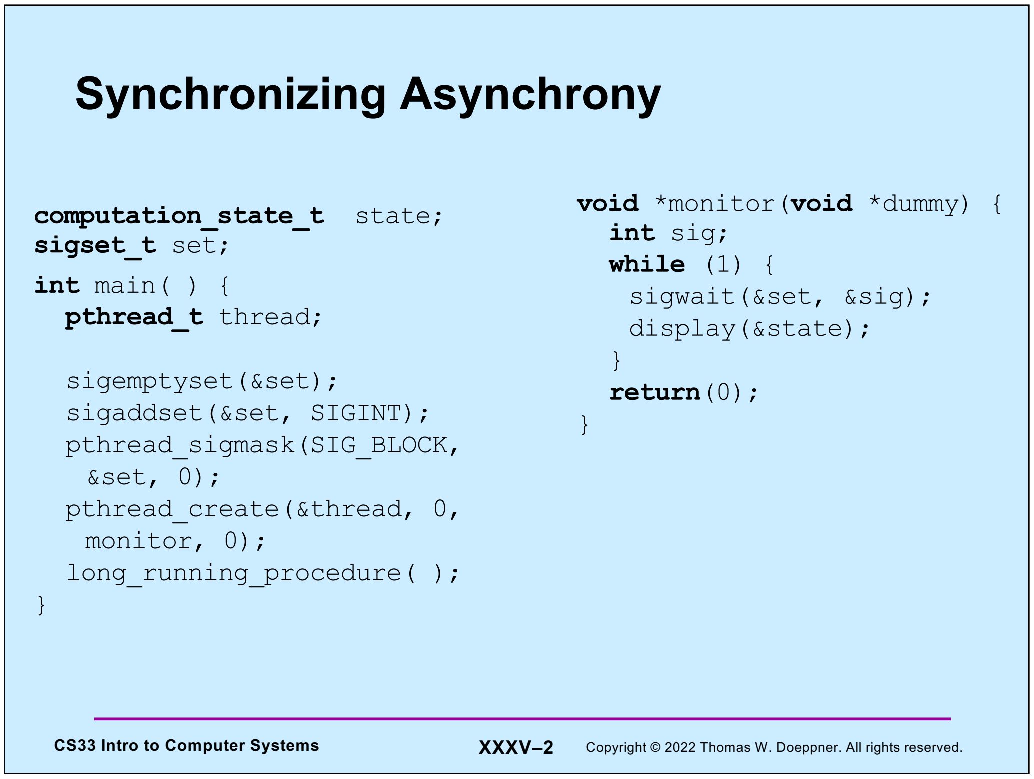
CS 33

Multithreaded Programming V



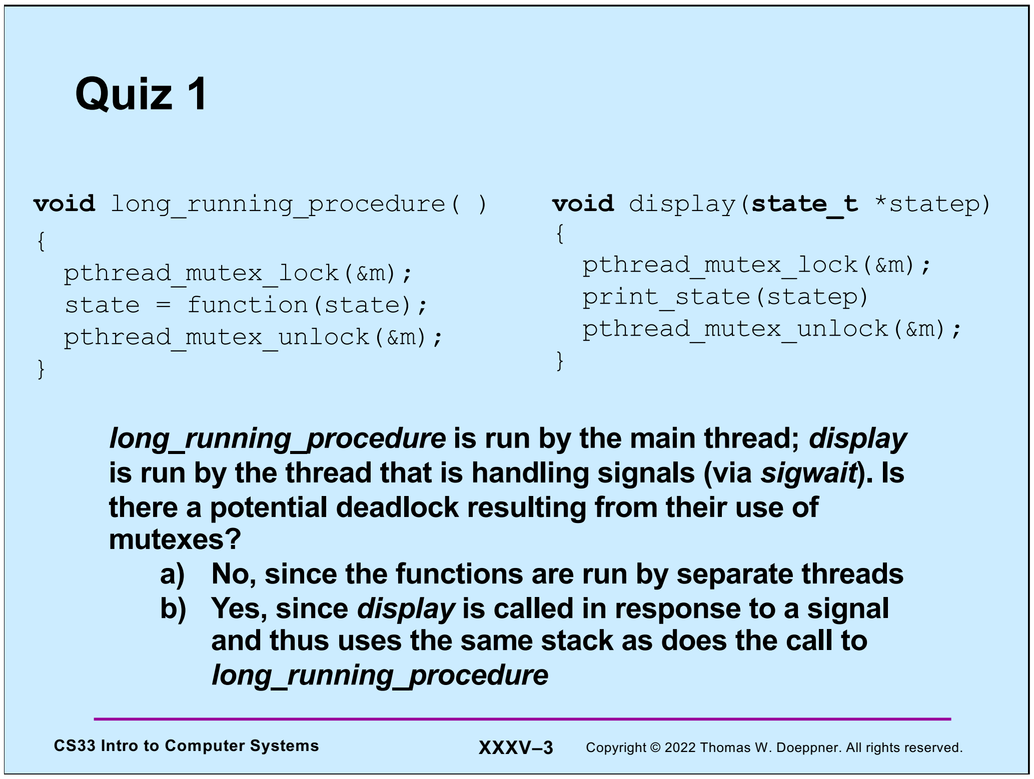
Synchronizing Asynchrony

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

  sigemptyset(&set);
  sigaddset(&set, SIGINT);
  pthread_sigmask(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** function. 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.



Quiz 1

```
void long_running_procedure( )
{
  pthread_mutex_lock(&m);
  state = function(state);
  pthread_mutex_unlock(&m);
}
```

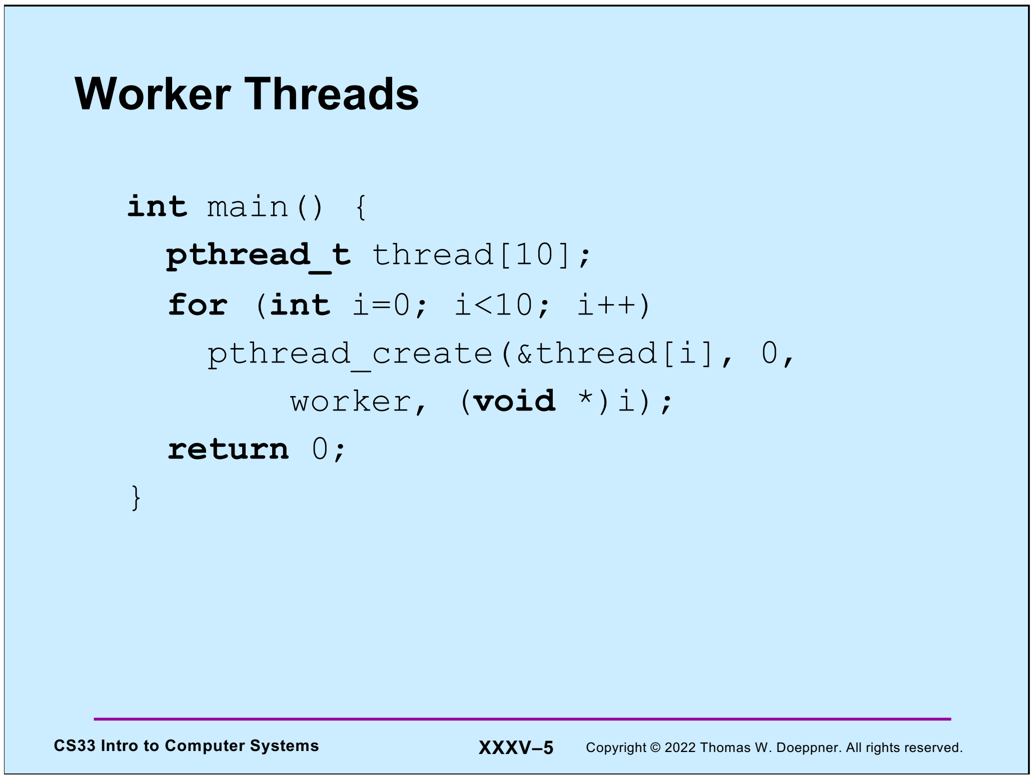
```
void display(state_t *statep)
{
  pthread_mutex_lock(&m);
  print_state(statep)
  pthread_mutex_unlock(&m);
}
```

long_running_procedure **is run by the main thread;** ***display*** **is run by the thread that is handling signals (via** ***sigwait*****). Is there a potential deadlock resulting from their use of mutexes?**

- **a) No, since the functions are run by separate threads**
- **b) Yes, since** ***display*** **is called in response to a signal and thus uses the same stack as does the call to** ***long_running_procedure***

Some Thread Gotchas ...

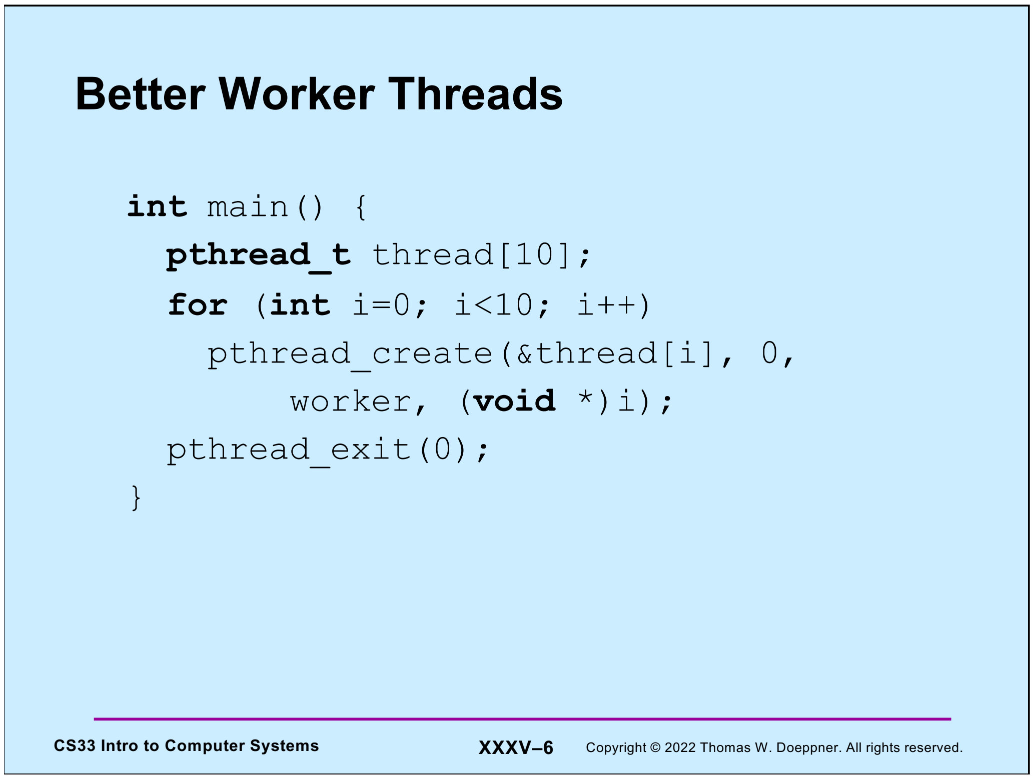
- Exit vs. pthread_exit
- Handling multiple arguments



Worker Threads

```
int main() {
  pthread_t thread[10];
  for (int i=0; i<10; i++)
    pthread_create(&thread[i], 0,
        worker, (void *)i);
  return 0;
}
```

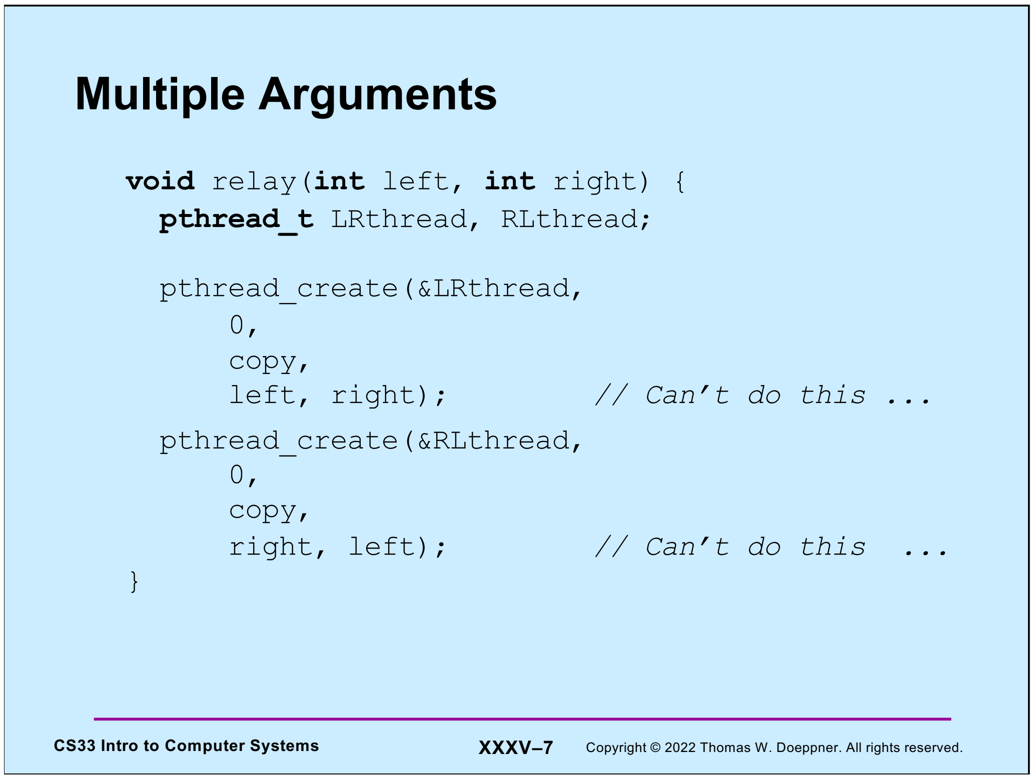
This program will probably do nothing! When the main function returns, it returns to code that calls **exit**. Exit terminates all threads in the process.



Better Worker Threads

```
int main() {
  pthread_t thread[10];
  for (int i=0; i<10; i++)
    pthread_create(&thread[i], 0,
        worker, (void *)i);
  pthread_exit(0);
}
```

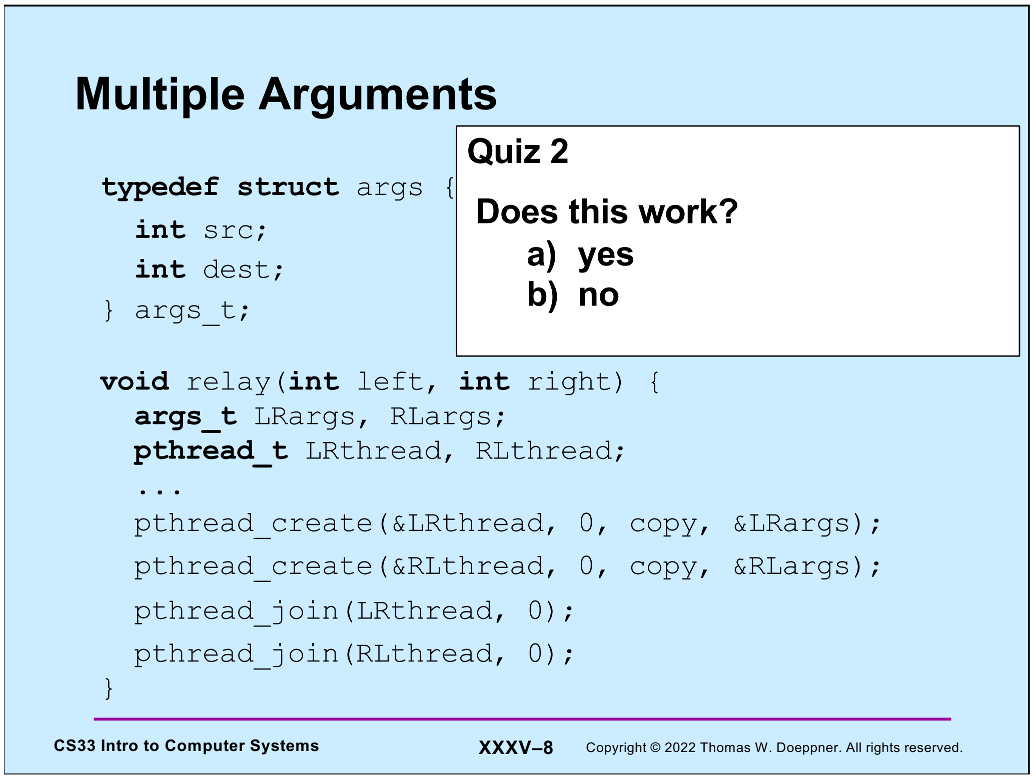
A better way to do what was intended in the previous slide is for the first thread (the one creating the workers) to terminate itself by calling **pthread_exit**. This allows the other threads to continue to run. The entire process will terminate when all of its threads have terminated.



Multiple Arguments

```
void relay(int left, int right) {
  pthread_t LRthread, RLthread;

  pthread_create(&LRthread,
      0,
      copy,
      left, right);         // Can't do this ...
  pthread_create(&RLthread,
      0,
      copy,
      right, left);         // Can't do this  ...
}
```

We discussed earlier that a limitation of **pthread_create** is that only one argument may be passed to the thread's first function.



Multiple Arguments

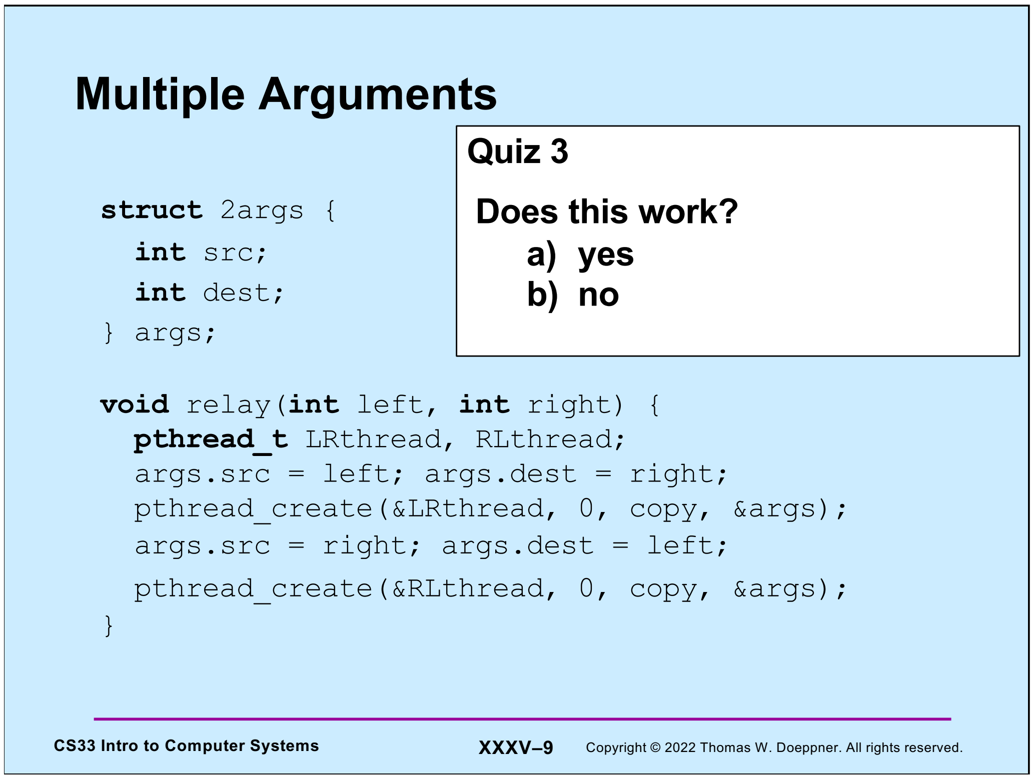
```
typedef struct args {
  int src;
  int dest;
} args_t;

void relay(int left, int right) {
  args_t LRargs, RLargs;
  pthread_t LRthread, RLthread;
  ...
  pthread_create(&LRthread, 0, copy, &LRargs);
  pthread_create(&RLthread, 0, copy, &RLargs);
  pthread_join(LRthread, 0);
  pthread_join(RLthread, 0);
}
```

Quiz 2

Does this work?

a) yes
b) no

To pass more than one argument to the first procedure of a thread, we must somehow encode multiple arguments as one. Here we pack two arguments into a structure, then pass the pointer to the structure. We saw earlier that this doesn't work if the calls to **pthread_join** are not there. Does it work if the calls are there, as in the slide?



Multiple Arguments

Quiz 3

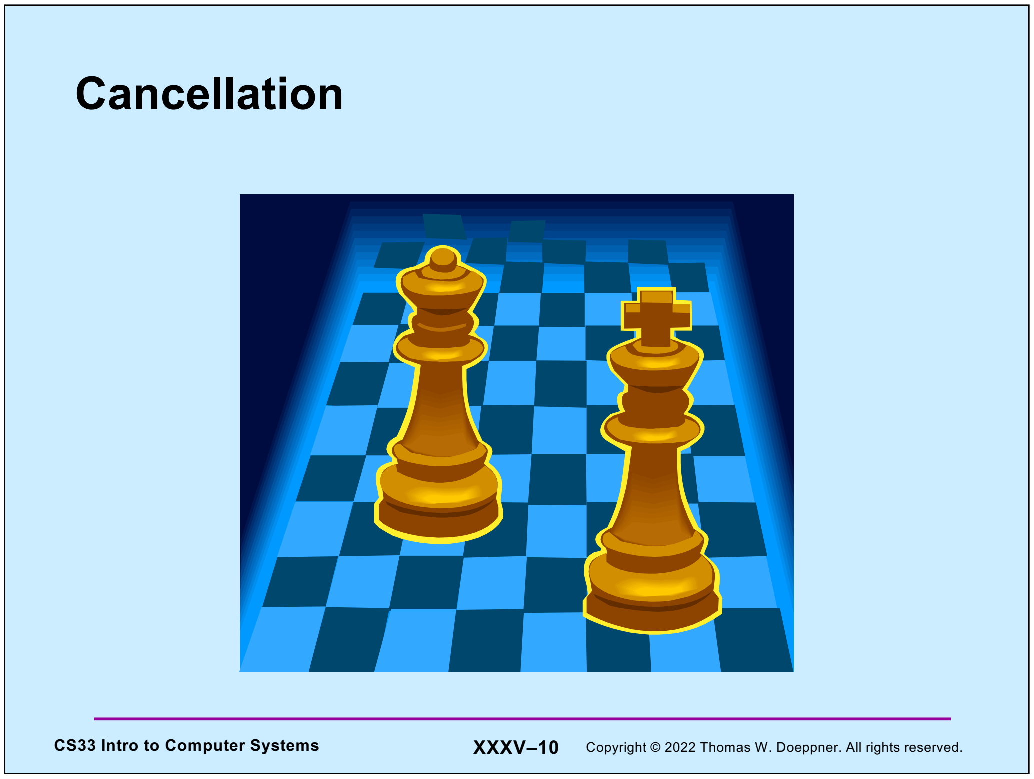
Does this work?

a) yes

b) no

```
struct 2args {
  int src;
  int dest;
} args;

void relay(int left, int right) {
  pthread_t LRthread, RLthread;
  args.src = left; args.dest = right;
  pthread_create(&LRthread, 0, copy, &args);
  args.src = right; args.dest = left;
  pthread_create(&RLthread, 0, copy, &args);
}
```

Here, **args** isn't a local variable, but is global.

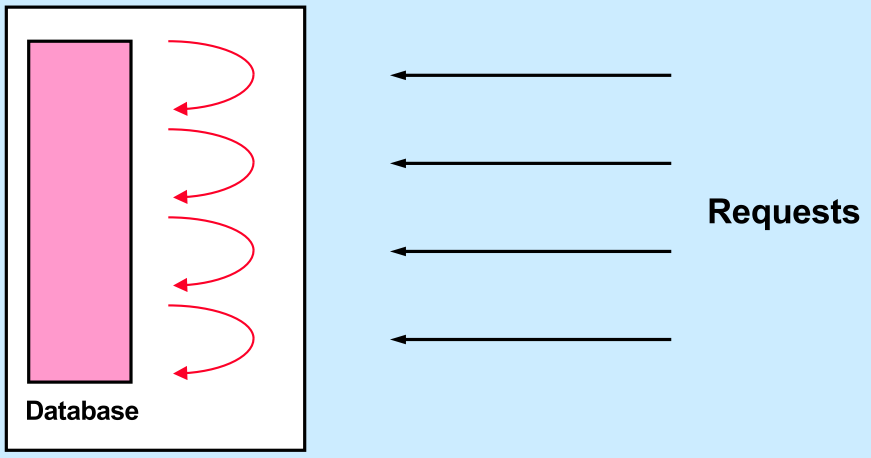


Cancellation

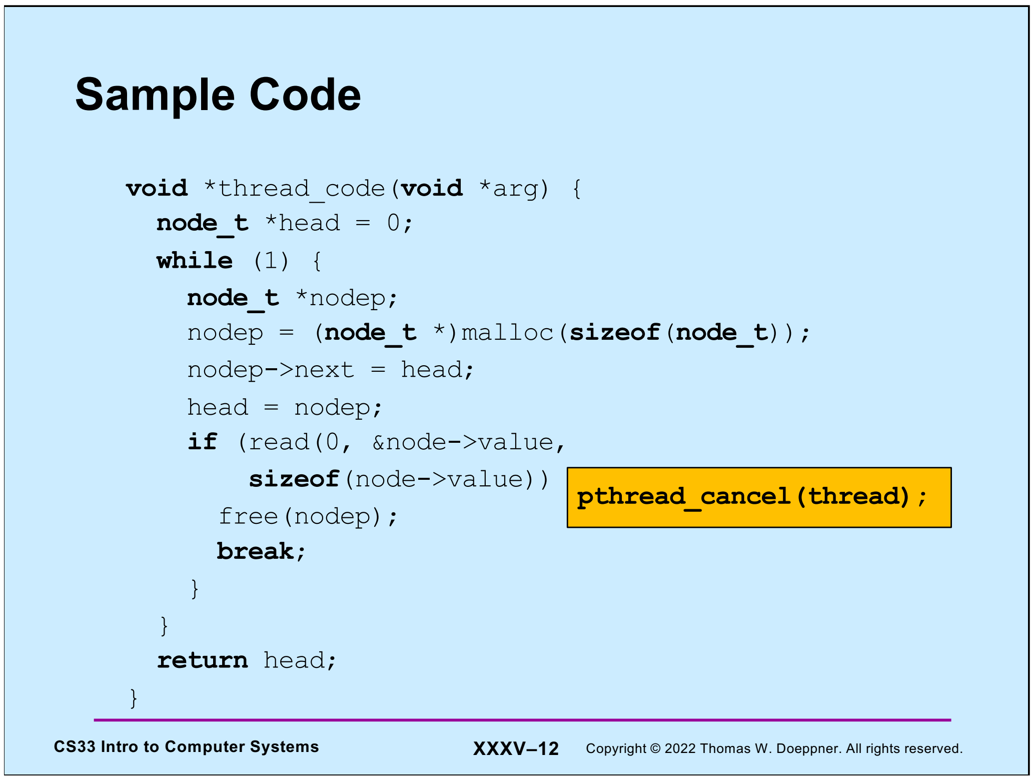
In a number of situations one thread must tell another to cease whatever it is doing. For example, suppose we've implemented a chess-playing program by having multiple threads search the solution space for the next move. If one thread has discovered a quick way of achieving a checkmate, it would want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

Multithreaded Database Server



In the database project, you are to write a multithreaded database server. It will have multiple concurrent clients; each client is handled by a separate thread. As we've already discussed, the database will be implemented using a binary search tree. We'd like to be able to terminate clients, perhaps all of them, without doing any damage to the database.



Sample Code

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

`pthread_cancel(thread);`

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

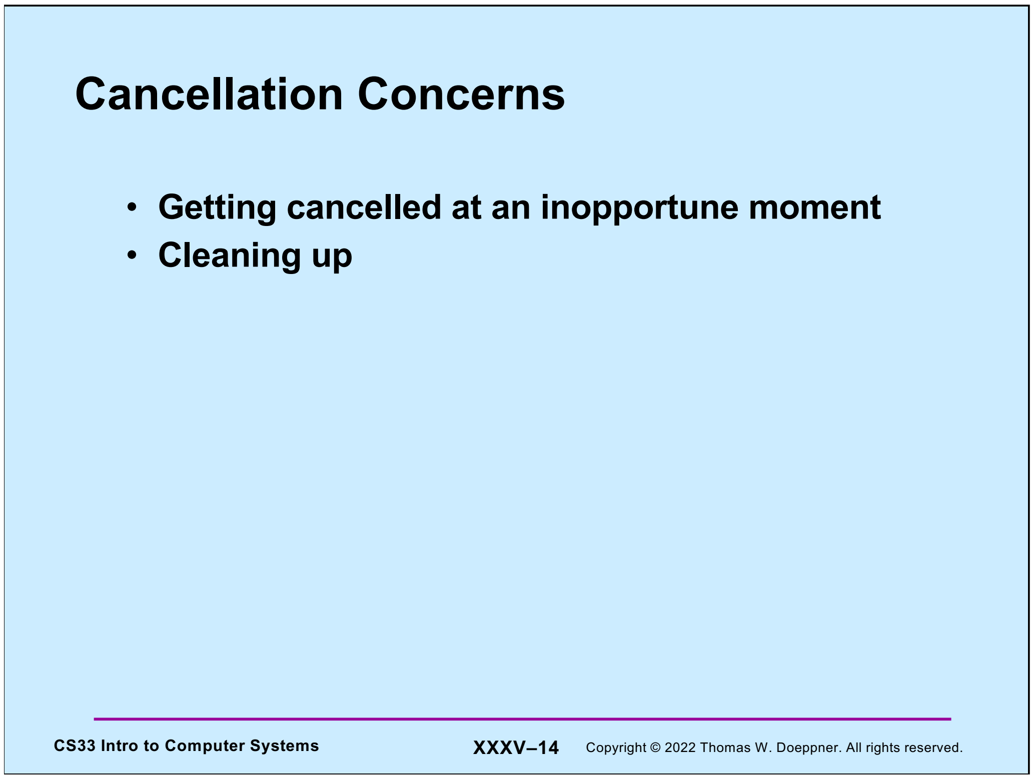
Suppose our thread is forced to terminate in the midst of its execution (some other thread invokes the operation **pthread_cancel** on it). What sort of problems might ensue?

Quiz 4

```
1    void *thread_code(void *arg) {
2      node_t *head = 0;
3      while (1) {
4        node_t *nodep;
5        nodep = (node_t *)malloc(size
6        nodep->next = head;
7        head = nodep;
8        if (read(0, &node->value,
             sizeof(node->value)) == 0) {
9          free(nodep);
10         break;
11       }
12     }
13     return head;
14   }
```

Where is it safe to terminate a thread within *thread_code*?

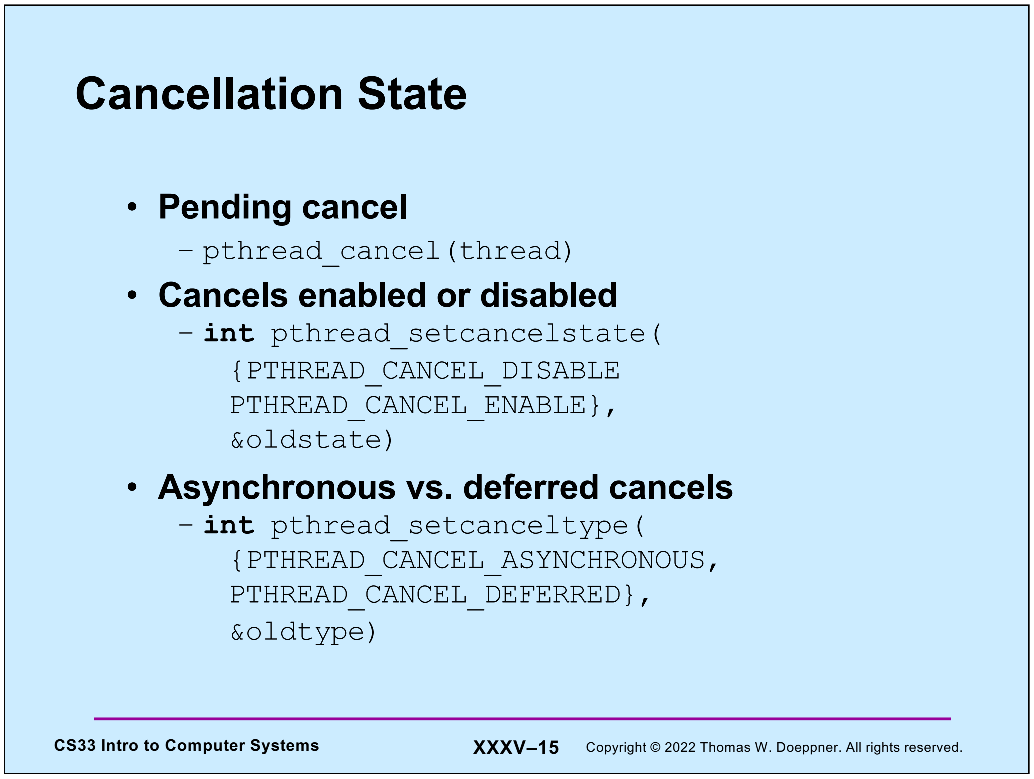
a) **At all lines**
b) **At all lines other than 5 and 9**
c) **At all lines other than 8**
d) **At all lines other than 5, 8, and 9**
e) **At no lines**



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**.

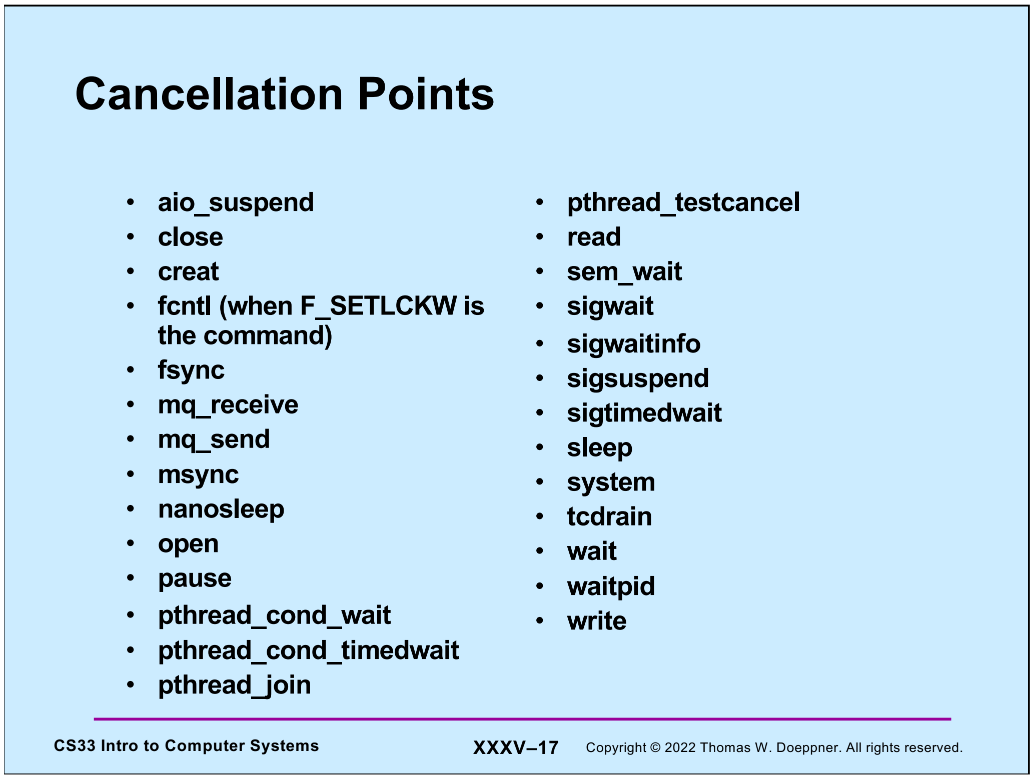
Cancellation points are intended to be well defined points in a thread's execution at which it is prepared to be canceled. They include pretty much all system and library calls in which the thread can block, with the exception of **pthread_mutex_lock**. In addition, a thread may call **pthread_testcancel**, 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.

Sample Code – Cancellation Point

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

The call to read is the only cancellation point in this program. Thus, if cancellation is deferred, the thread will act on cancels only within read.

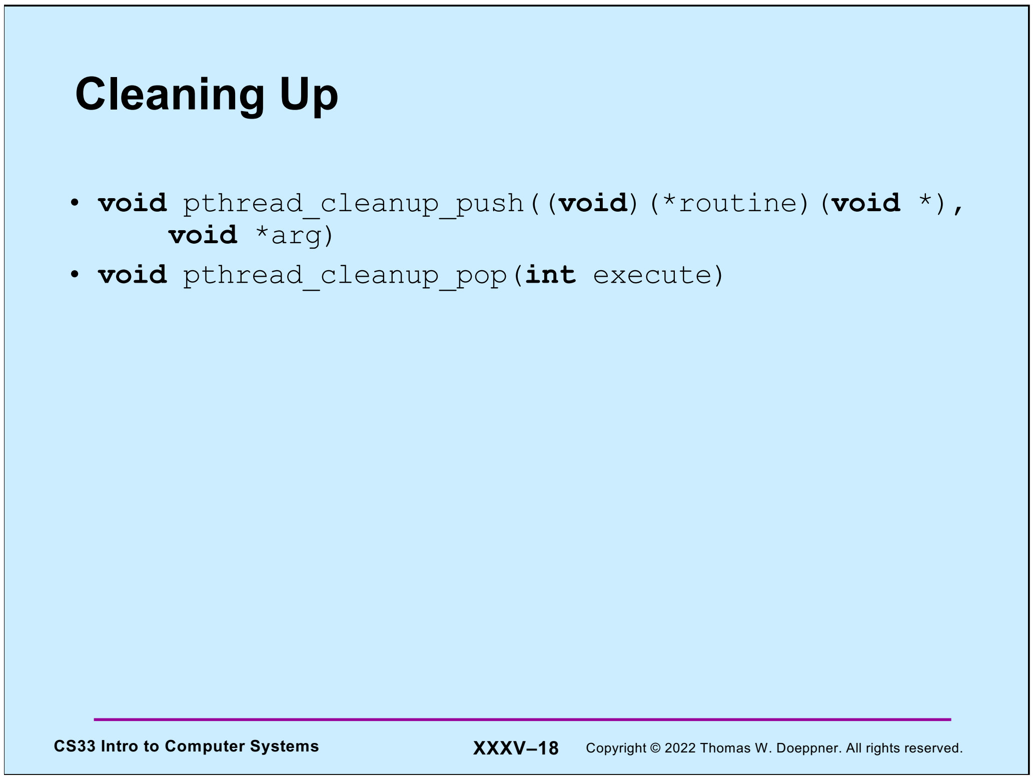


Cancellation Points

- 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.

The function **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.

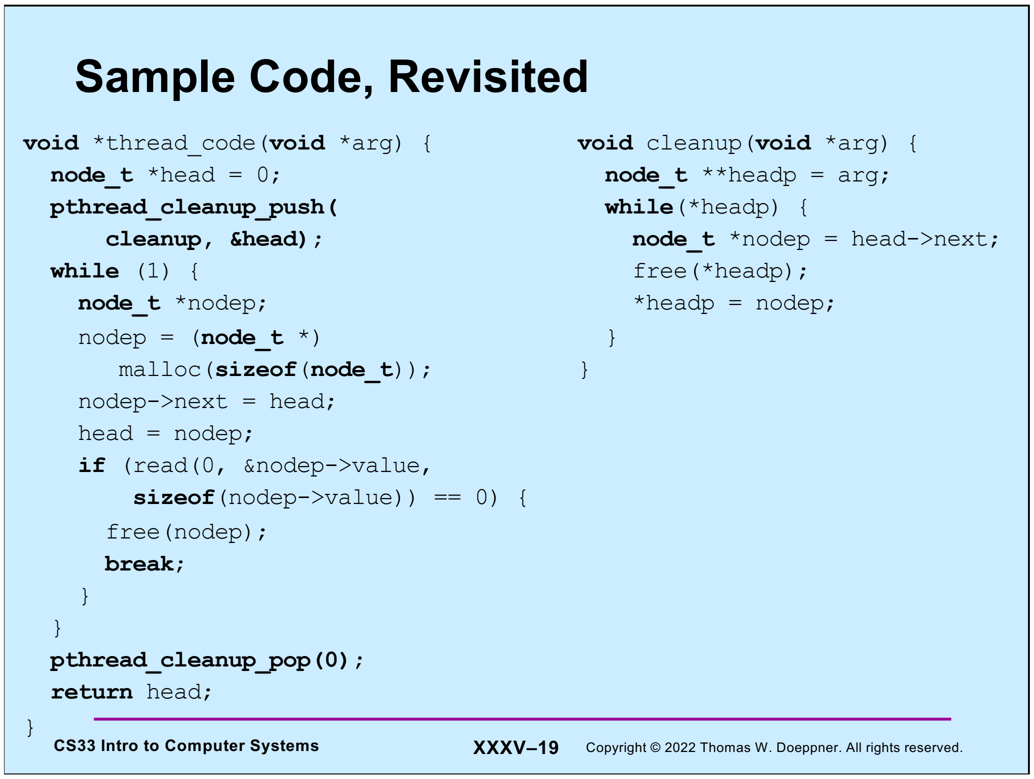


Cleaning Up

```
• 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 functions **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 function indicates whether or not the cleanup function should be called as a side effect of calling **pthread_cleanup_pop**.

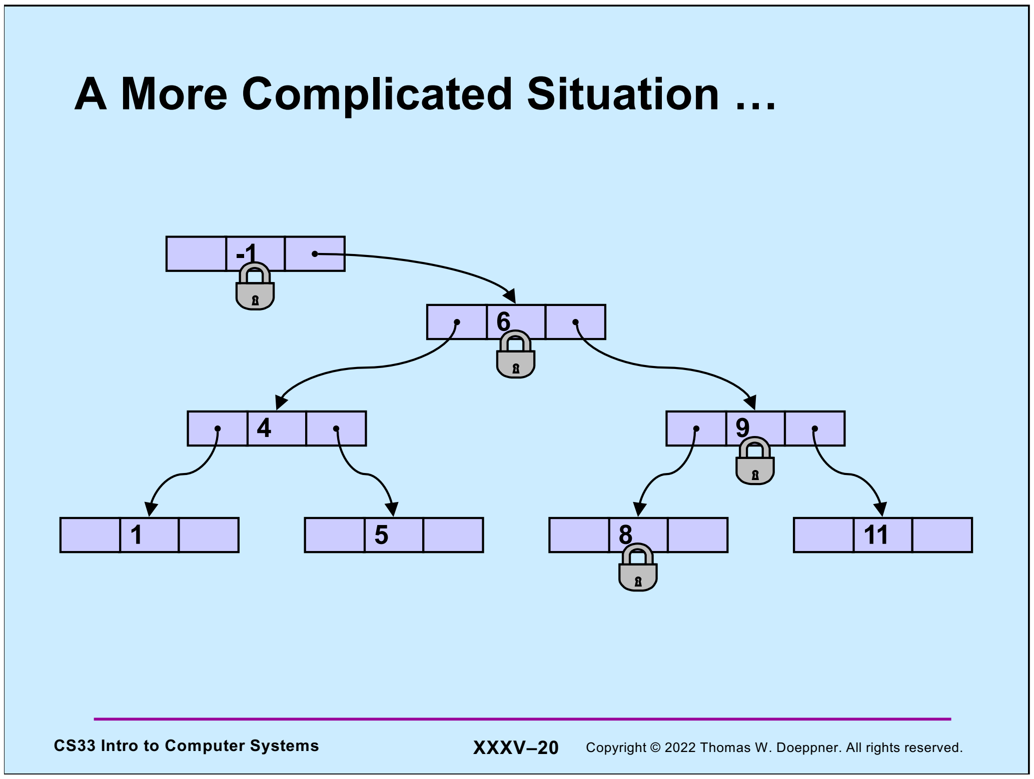


Sample Code, Revisited

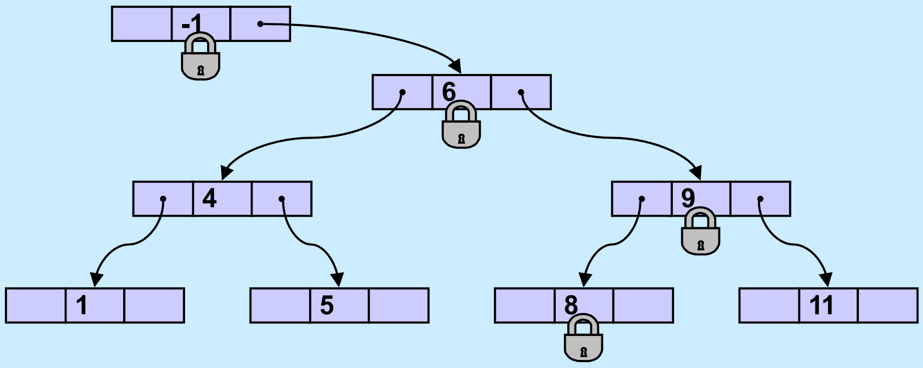
```
void *thread_code(void *arg) {
  node_t *head = 0;
  pthread_cleanup_push(
      cleanup, &head);
  while (1) {
    node_t *nodep;
    nodep = (node_t *)
       malloc(sizeof(node_t));
    nodep->next = head;
    head = nodep;
    if (read(0, &nodep->value,
        sizeof(nodep->value)) == 0) {
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

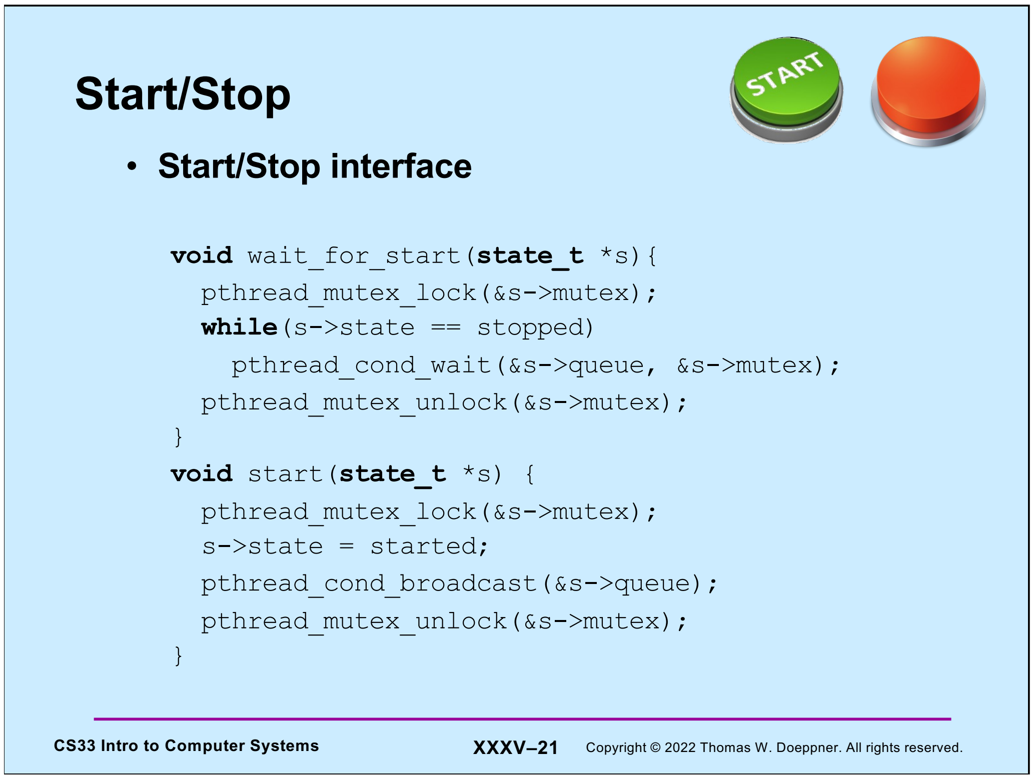
Here we’ve added a cleanup handler to our sample code. Note that our example has just one cancellation point: **read**. The cleanup handler iterates through the list, deleting each element.



A More Complicated Situation …



Whether threads are using mutexes or readers/writers locks when manipulating a search tree, if we have to deal with cancellation points in the middle of such operations, things can get pretty complicated and error-prone. Thus, the operations to lock mutexes and readers/writers locks are not cancellation points. (Note, however, that for the case of readers/writers locks, POSIX permits waiting for readers/writers locks to be cancellation points, for the sake of vendors who have poor implementations of them. Neither Linux nor OSX implements such waiting as cancellation points.)



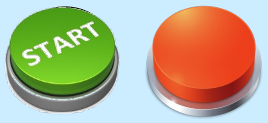
Start/Stop

- **Start/Stop interface**

```
void wait_for_start(state_t *s){
  pthread_mutex_lock(&s->mutex);
  while(s->state == stopped)
    pthread_cond_wait(&s->queue, &s->mutex);
  pthread_mutex_unlock(&s->mutex);
}
void start(state_t *s) {
  pthread_mutex_lock(&s->mutex);
  s->state = started;
  pthread_cond_broadcast(&s->queue);
  pthread_mutex_unlock(&s->mutex);
}
```

Here is our start/stop code again. Does it contain any cancellation points?

Start/Stop



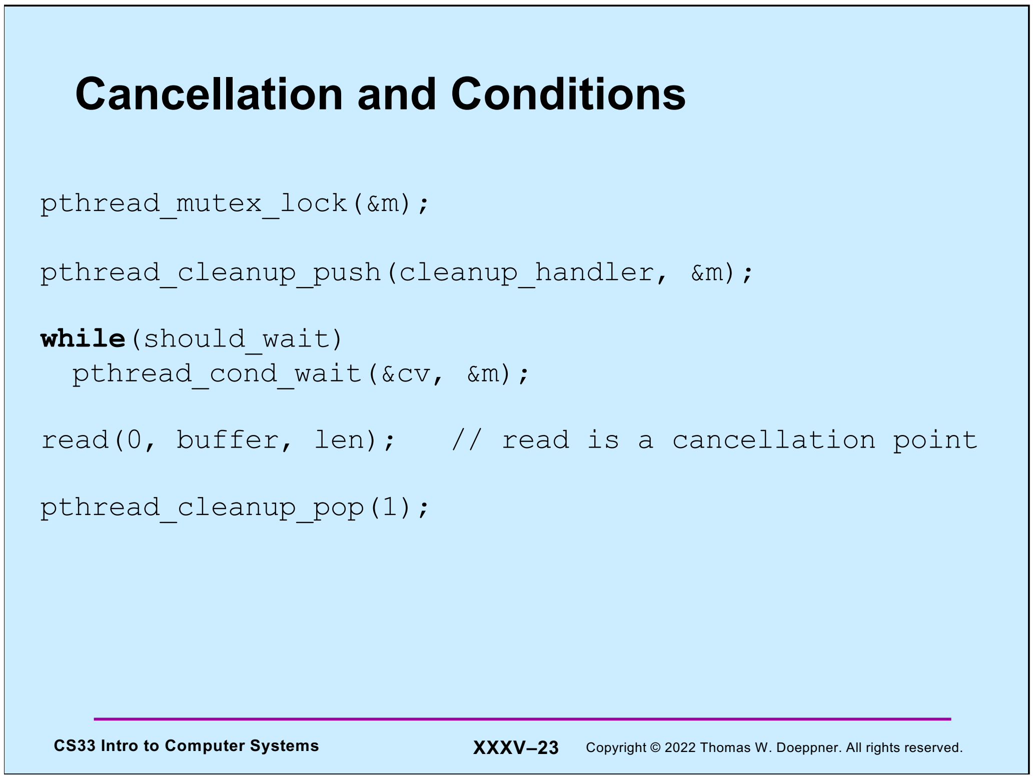
- **Start/Stop interface**

```
void wait_for_start(state_t *s){
  pthread_mutex_lock(&s->mutex);
  while(s->state == stopped)
    pthread_cond_wait(&s->queue,
      &s->mutex);
  pthread_mutex_unlock(&s->mutex);
}
void start(state_t *s) {
  pthread_mutex_lock(&s->mutex);
  s->state = started;
  pthread_cond_broadcast(&s->queue);
  pthread_mutex_unlock(&s->mutex);
}
```

Not a Quiz

You're in charge of designing POSIX threads. Should *pthread_cond_wait* be a cancellation point?

a) **no**
b) **yes; cancelled threads must acquire mutex before invoking cleanup handler**
c) **yes; but they don't acquire mutex**



Cancellation and Conditions

```
pthread_mutex_lock(&m);

pthread_cleanup_push(cleanup_handler, &m);

while(should_wait)
  pthread_cond_wait(&cv, &m);

read(0, buffer, len);   // read is a cancellation point

pthread_cleanup_pop(1);
```

This example illustrates why it's important that threads cancelled while in **pthread_cond_wait** must first lock the mutex before calling their cleanup handler. In this example, it's important (for an unspecified reason) that **read** be called while the mutex is locked and **should_wait** is true. If the thread receives a cancel and **cleanup_handler** is called, it won't be known whether the cancel occurred within **pthread_cond_wait** or within **read**. Thus **cleanup_handler** must perform the same actions in both cases. Since the thread must unlock the mutex if the cancel occurred while the thread was in **read**, it must also unlock the mutex if the cancel occurred while the thread was in **pthread_cond_wait**. Thus, it's important that a thread cancelled while in **pthread_cond_wait** lock the mutex before it calls its cleanup handler, so that it's locked when the thread enters the cleanup handler.

Quiz 5

- **Start/Stop interface**

```
void wait_for_start(state_t *s){
  pthread_mutex_lock(&s->mutex);
  pthread_cleanup_push(
    cleanup_func, cleanup_arg);
  while(s->state == stopped)
    pthread_cond_wait(&s->queue, &s->mutex);
  pthread_cleanup_pop(1);
}
void start(state_t *s) {
  pthread_mutex_lock(&s->mutex);
  s->state = started;
  pthread_cond_broadcast(&s->queue);
  pthread_mutex_unlock(&s->mutex);
}
```

What should be used for *cleanup_func* and *cleanup_arg*?

a) *pthread_mutex_unlock* and *&s->mutex*

b) that and more

c) there's no need for a cleanup function

A Problem ...

- **In thread 1:**

```
if ((ret = open(path,
    O_RDWR) == -1) {
  if (errno == EINTR) {
    ...
  }
  ...
}
```

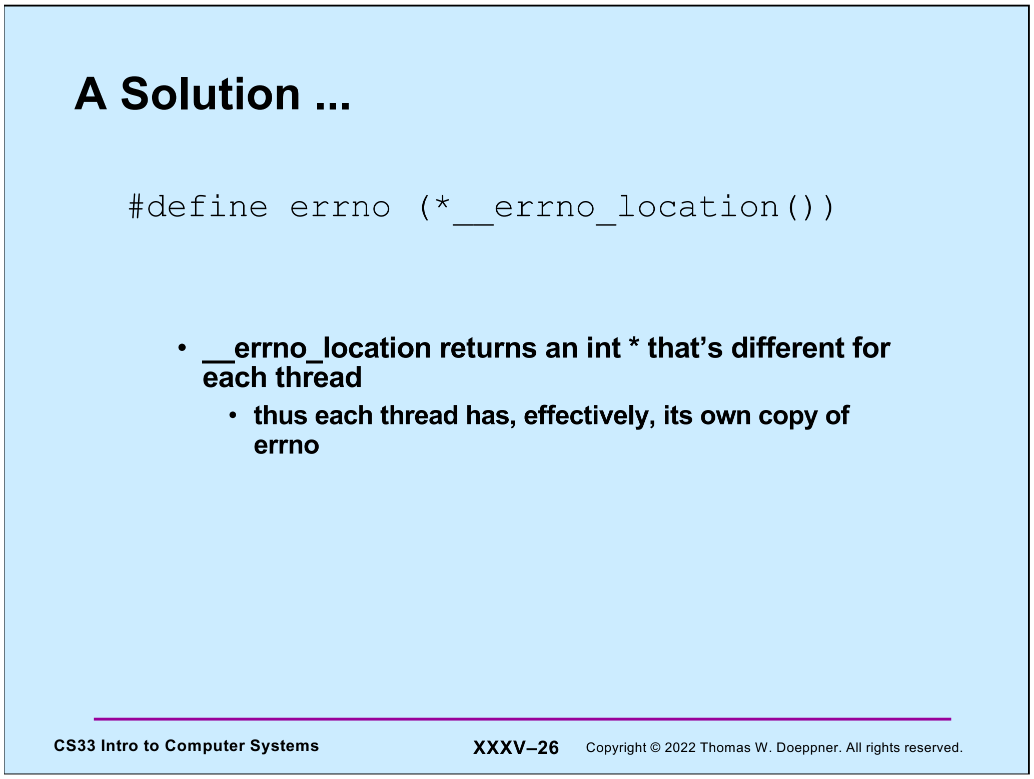
- **In thread 2:**

```
if ((ret = socket(AF_INET,
    SOCK_STREAM, 0)) {
  if (errno == ENOMEM) {
    ...
  }
  ...
```

There's only one errno!

However, somehow it works.

What's done???



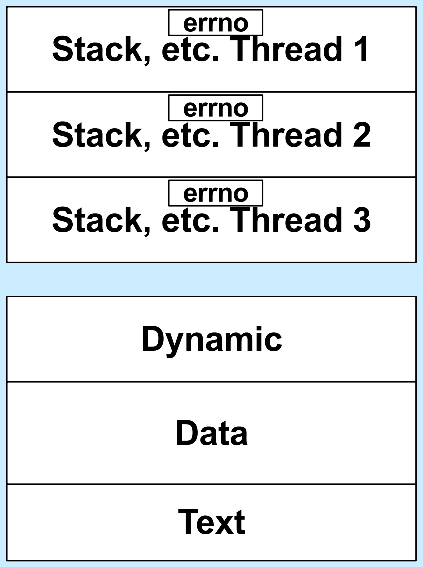
A Solution ...

```
#define errno (*__errno_location())
```

- **__errno_location returns an int * that's different for each thread**
  - **thus each thread has, effectively, its own copy of errno**

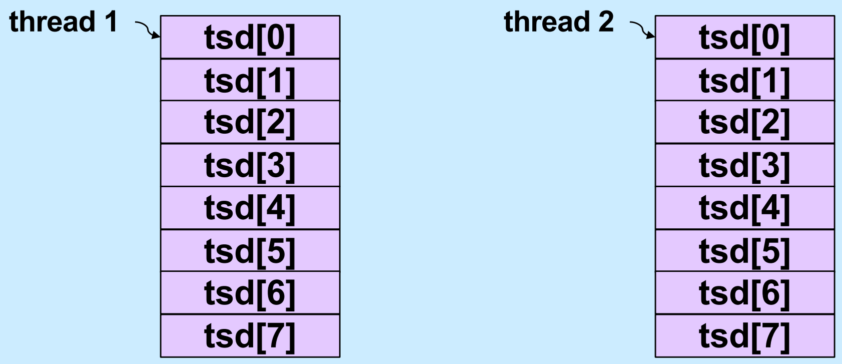
When you give gcc the –pthread flag, it, among other things, defines some preprocessor variables that cause some code in the standard header files to be compiled (that otherwise wouldn't be). In particular the #define statement given in the slide is compiled.

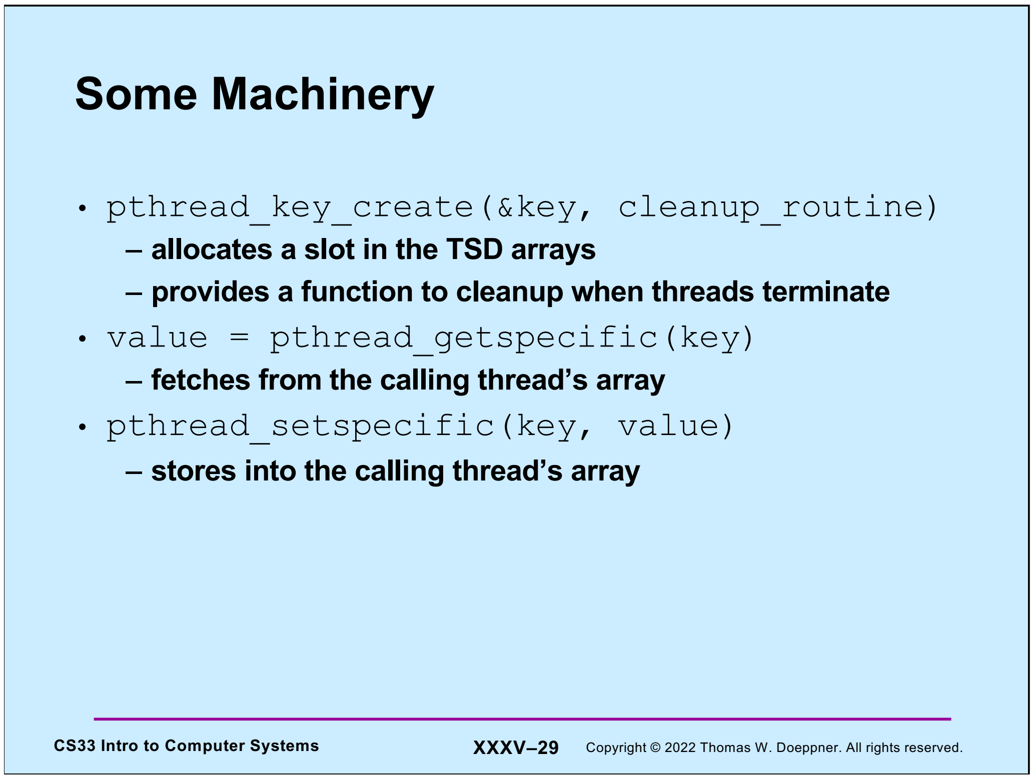
Process Address Space



Generalizing

- *Thread-specific data* (sometimes called *thread-local storage*)
  - data that's referred to by global variables, but each thread has its own private copy

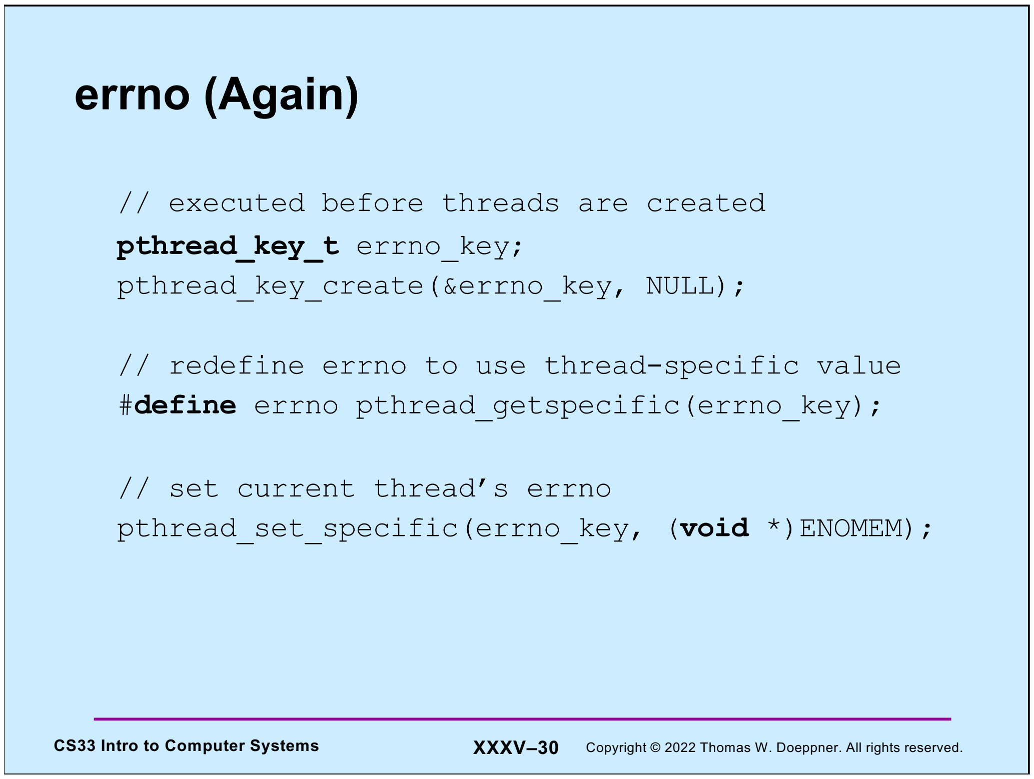




Some Machinery

- `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).*

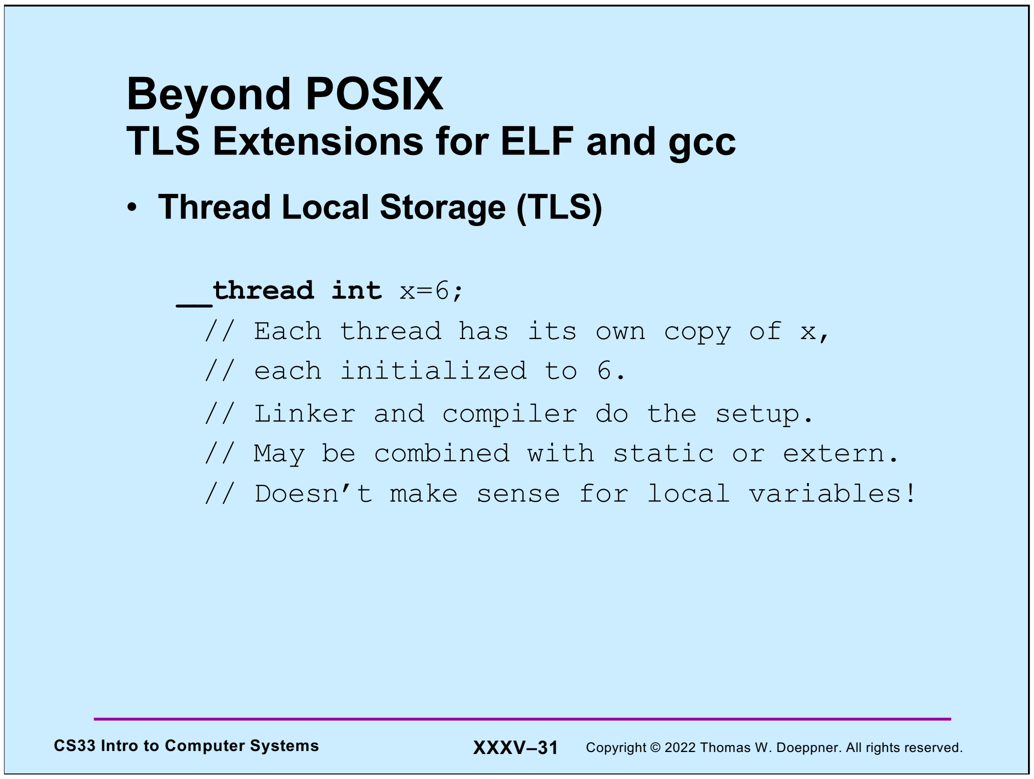


errno (Again)

```
// executed before threads are created
pthread_key_t errno_key;
pthread_key_create(&errno_key, NULL);

// redefine errno to use thread-specific value
#define errno pthread_getspecific(errno_key);

// set current thread's errno
pthread_set_specific(errno_key, (void *)ENOMEM);
```

Using the thread-specific data functions we can create an alternative (but equivalent) implementation of thread-specific errno.

Before any threads (other than the original thread in the process) are created, pthread_key_create is called to initialize errno_key, a global variable. We define errno to be the value returned by pthread_getspecific – the calling thread's value of errno. Then to set a thread's errno, we use pthread_setspecific.

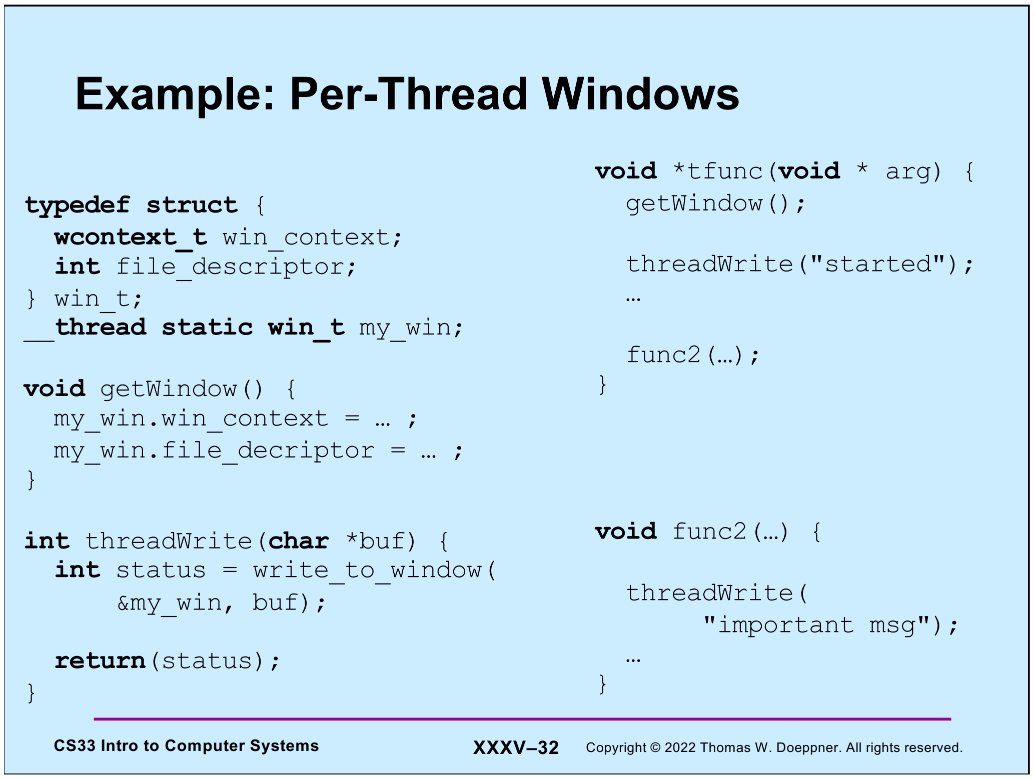


Beyond POSIX
TLS Extensions for ELF and gcc

- **Thread Local Storage (TLS)**

```
__thread int x=6;
  // Each thread has its own copy of x,
  // each initialized to 6.
  // Linker and compiler do the setup.
  // May be combined with static or extern.
  // Doesn't make sense for local variables!
```

ELF stands for "executable and linking format" and is the standard format for executable and object files used on most Unix systems. The __**thread** attribute tells gcc that each thread is to have its own copy of the variable. A detailed description of how it is implemented can be found at http://people.redhat.com/drepper/tls.pdf.

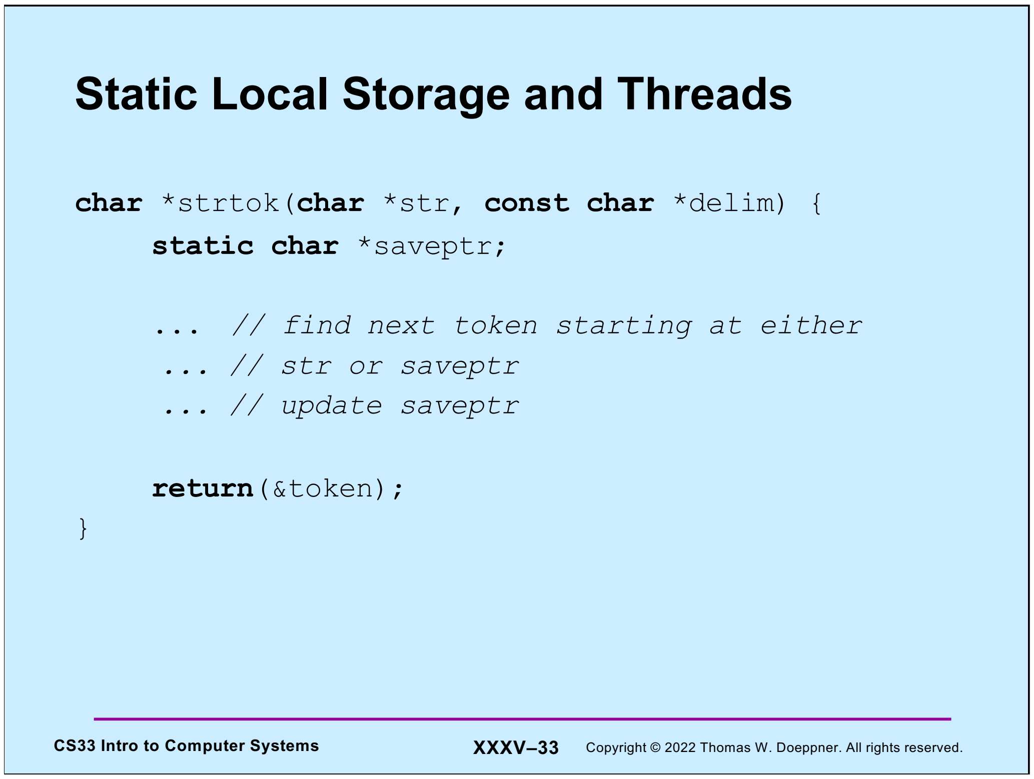


Example: Per-Thread Windows

```
typedef struct {
  wcontext_t win_context;
  int file_descriptor;
} win_t;
__thread static win_t my_win;

void getWindow() {
  my_win.win_context = … ;
  my_win.file_decriptor = … ;
}

int threadWrite(char *buf) {
  int status = write_to_window(
      &my_win, buf);

  return(status);
}
```

```
void *tfunc(void * arg) {
  getWindow();

  threadWrite("started");
  …

  func2(…);
}


void func2(…) {

  threadWrite(
       "important msg");
  …
}
```

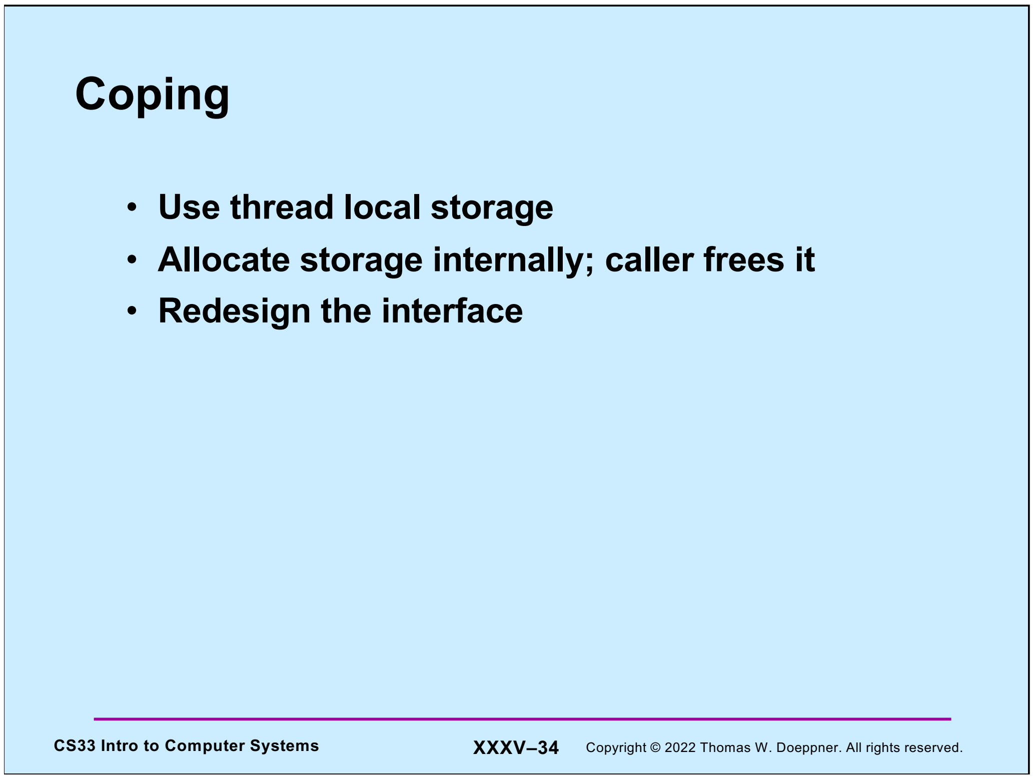
In this example, we put together per-thread windows for thread output. Threads call **getWindow** to set up a window for their exclusive use, then call **threadWrite** to send output to their windows. Individual threads can now set up their own windows and write to them without having to pass around information describing which are their windows. Each thread's window is referred to by the static global variable **my_win**.



Static Local Storage and Threads

```
char *strtok(char *str, const char *delim) {
    static char *saveptr;

    ...  // find next token starting at either
     ... // str or saveptr
     ... // update saveptr

    return(&token);
}
```

An example of the single-thread mentality in early Unix is the use of static local storage in a number of library routines. An example of this is **strtok**, which saves a pointer into the input string for use in future calls to the function (which we've used extensively in earlier projects). This works fine as long as just one thread is using the function, but fails if multiple threads use it – each will expect to find its own saved pointer in **saveptr**, but there's only one **saveptr**.

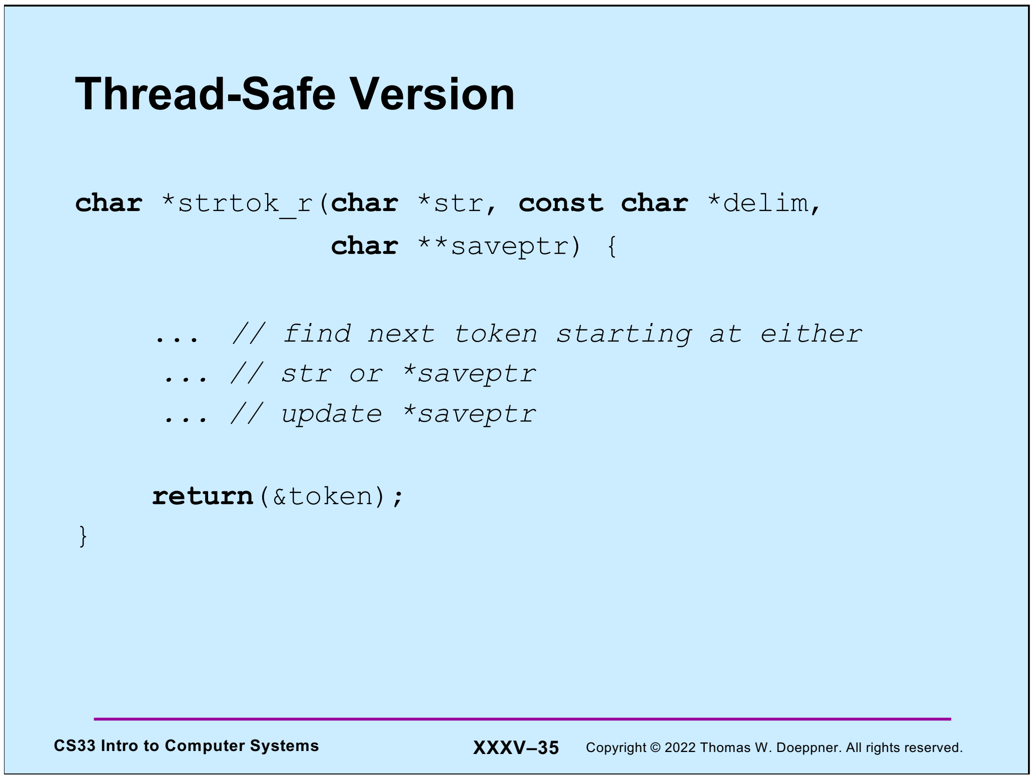


Coping

- Use thread local storage
- 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-local storage, but this would entail associating a fair amount of storage with each thread, even if it is not using **strtok**. We might simply allocate storage (via **malloc**) inside **strtok** 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.

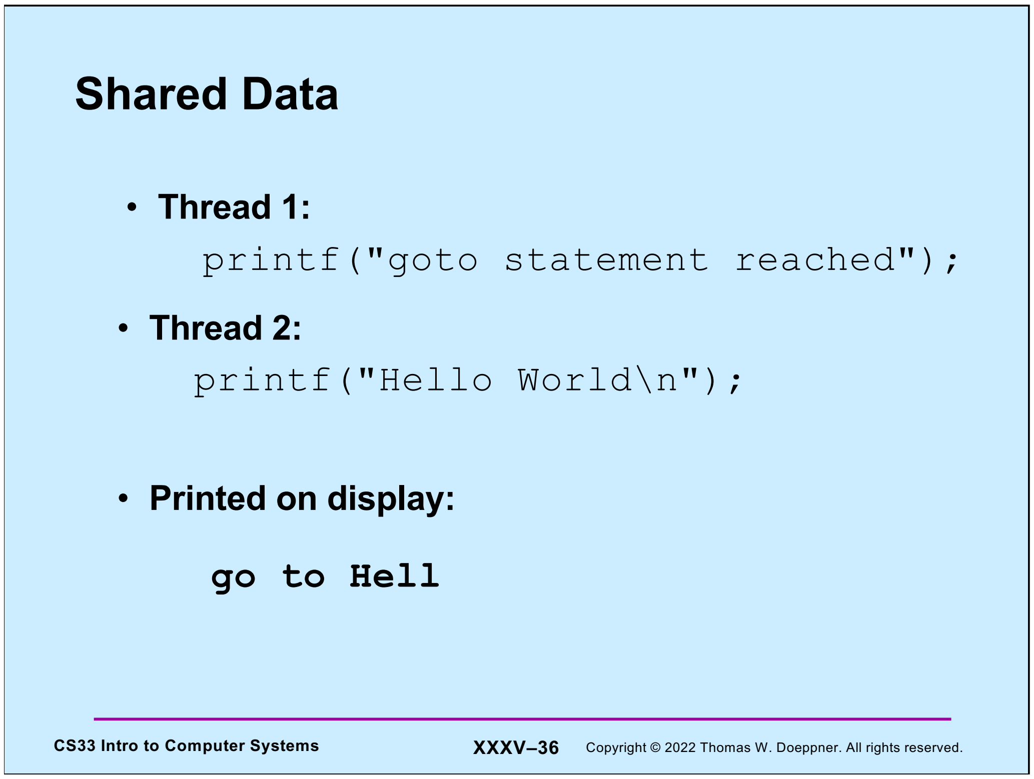
The solution taken is to redesign the interface. The "thread-safe" version is called **strtok_r** (the **r** stands for **reentrant**, an earlier term for "thread-safe"); it takes an additional parameter pointing to storage that holds saveptr. Thus the caller is responsible for both the allocation and the liberation of the storage containing saveptr; this storage is typically a local variable (allocated on the stack), so that its allocation and liberation overhead is negligible, at worst.



Thread-Safe Version

```
char *strtok_r(char *str, const char *delim,
               char **saveptr) {

    ...  // find next token starting at either
     ... // str or *saveptr
     ... // update *saveptr

    return(&token);
}
```

Here's the thread-safe version of **strtok**.



Shared Data

- **Thread 1:**

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

- **Thread 2:**

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

- **Printed on display:**

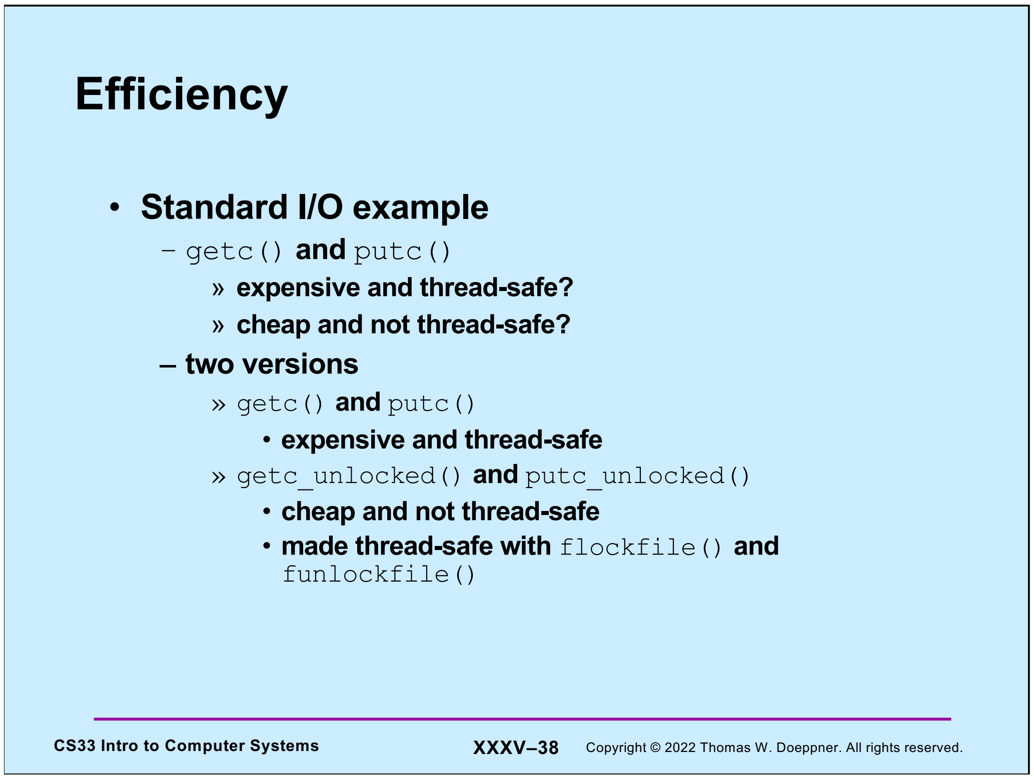
```
go to Hell
```

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 functions). A simple way to do this would be to supply wrappers for all of the standard I/O functions 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 functions themselves, rather than supplying wrappers (this is what is done in most implementations).



Efficiency

- **Standard I/O example**
  - `getc()` **and** `putc()`
    - » **expensive and thread-safe?**
    - » **cheap and not thread-safe?**
  - **two versions**
    - » `getc()` **and** `putc()`
      - **expensive and thread-safe**
    - » `getc_unlocked()` **and** `putc_unlocked()`
      - **cheap and not thread-safe**
      - **made thread-safe with** `flockfile()` **and** `funlockfile()`

After making a library thread-safe, we may discover that many functions have become too slow. For example, the standard-I/O functions **getc** and **putc** are 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 functions, one must take care to handle the synchronization oneself. This is accomplished with **flockfile** and **funlockfile**.

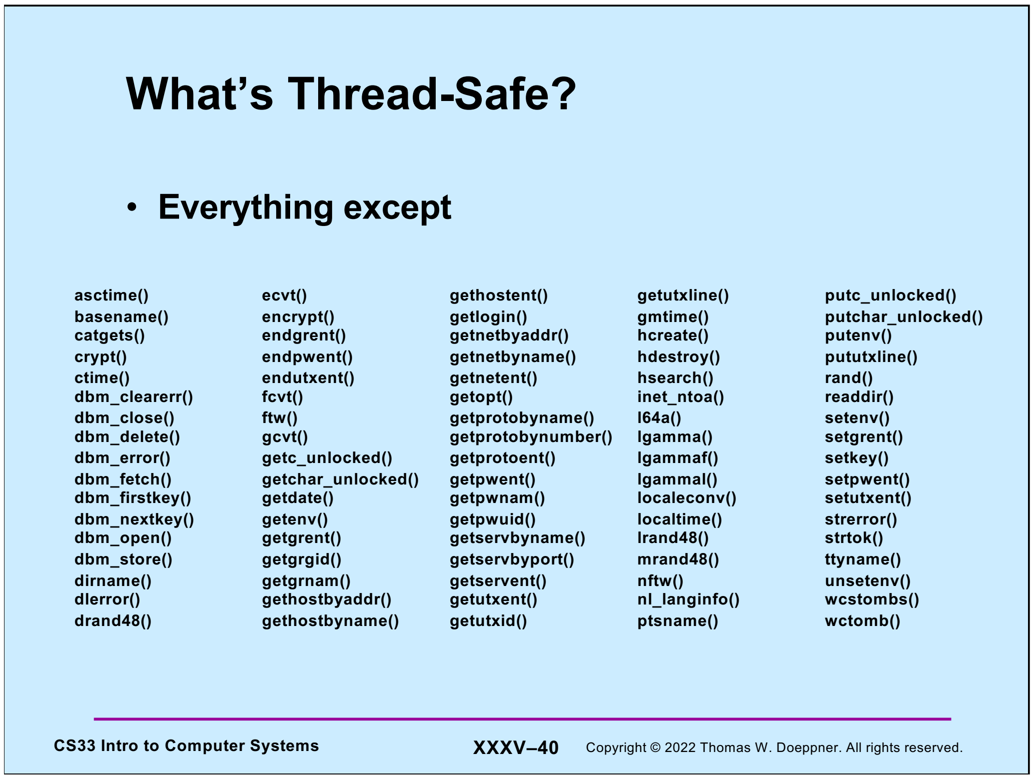
Efficiency

- **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);
```



What’s Thread-Safe?

- Everything except

asctime()	ecvt()	gethostent()	getutxline()	putc_unlocked()
basename()	encrypt()	getlogin()	gmtime()	putchar_unlocked()
catgets()	endgrent()	getnetbyaddr()	hcreate()	putenv()
crypt()	endpwent()	getnetbyname()	hdestroy()	pututxline()
ctime()	endutxent()	getnetent()	hsearch()	rand()
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()	strtok()
dbm_store()	getgrgid()	getservbyport()	mrand48()	ttyname()
dirname()	getgrnam()	getservent()	nftw()	unsetenv()
dlerror()	gethostbyaddr()	getutxent()	nl_langinfo()	wcstombs()
drand48()	gethostbyname()	getutxid()	ptsname()	wctomb()

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