# CS 33

## Multithreaded Programming V

# Some Thread Gotchas ...

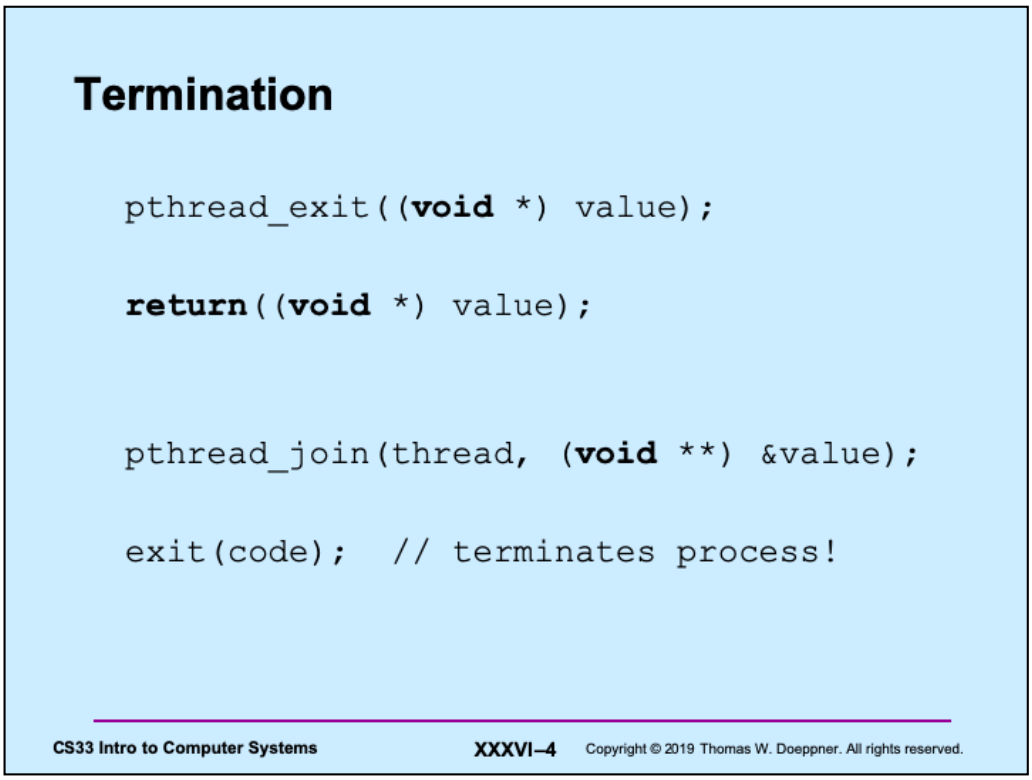
- Exit vs. pthread_exit
- Handling multiple arguments

## 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!




A thread terminates either by calling *pthread_exit* or by returning from its first procedure. In either case, it supplies a value that can be retrieved via a call (by some other thread) to *pthread_join*. The analogy to process termination and the *waitpid* system call in Unix is tempting and is correct to a certain extent — Unix's *waitpid*, like *pthread_join*, lets one caller synchronize with the termination of another. There is one important difference, however: Unix has the notion of parent/child relationships among processes. A process may wait only for its children to terminate. No such notion of parent/child relationship is maintained with POSIX threads: one thread may wait for the termination of any other thread in the process (though some threads cannot be "joined" by any thread — see the next page). It is, however, important that *pthread_join* be called for each joinable terminated thread — since threads that have terminated but have not yet been joined continue to use up some resources, resources that will be freed once the thread has been joined. The effect of multiple threads calling *pthread_join* is "undefined" — meaning that what happens can vary from one implementation to the next.

One should be careful to distinguish between terminating a thread and terminating a process. With the latter, all the threads in the process are forcibly terminated. So, if *any* thread in a process calls *exit*, the entire process is terminated, along with its threads. Similarly, if a thread returns from *main*, this also terminates the entire process, since returning from *main* is equivalent to calling *exit*. The only thread that can legally return from *main* is the one that called it in the first place. All other threads (those that did not call *main*) certainly do not terminate the entire process when they return from their first procedures, they merely terminate themselves.

If no thread calls *exit* and no thread returns from main, then the process should terminate once all threads have terminated (i.e., have called *pthread_exit* or, for threads

other than the first one, have returned from their first procedure). If the first thread calls *pthread_exit*, it self-destructs, but does not cause the process to terminate (unless no other threads are extant).

## Complications

```
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  ...
}
```

An obvious limitation of the *pthread_create* interface is that one can pass only a single argument to the first procedure of the new thread. In this example, we are trying to supply code for the *relay* example, but we run into a problem when we try to pass two parameters to each of the two threads.

## 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);
}
```

**Quiz 1**

**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.

# Multiple Arguments

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

**Quiz 2**

**Does this work?**

**a) yes**

**b) no**

```
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);
}
```

## 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 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.

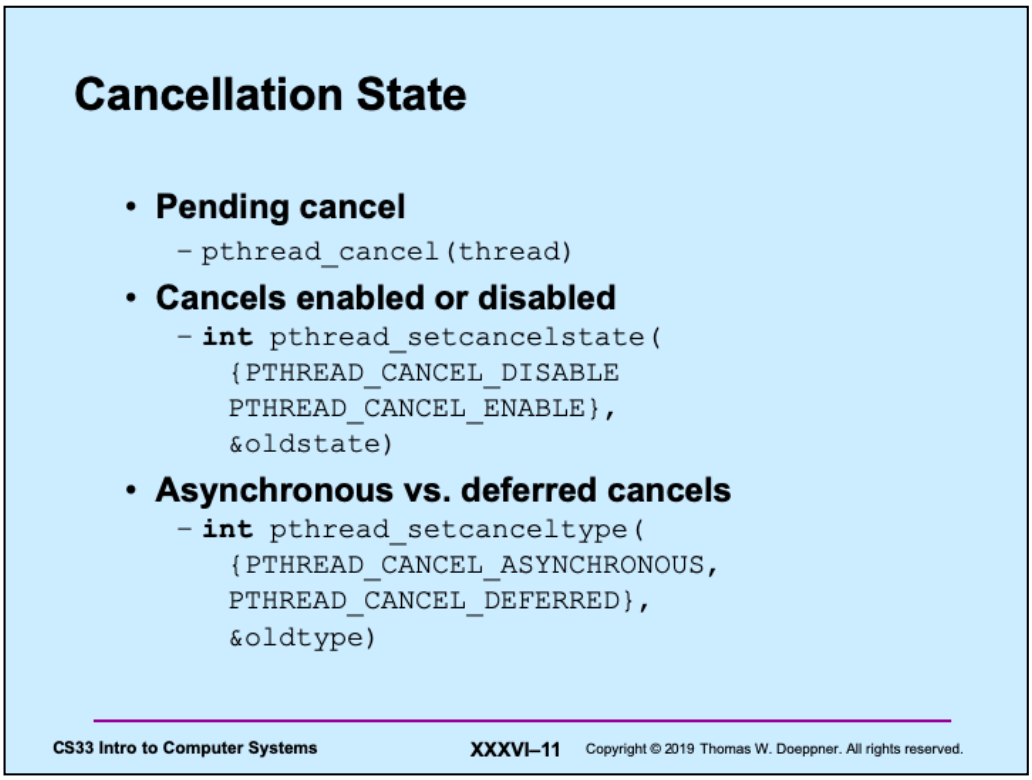
Suppose our thread is forced to terminate in the midst of its execution (some other thread invokes the operation *pthread_cancel* on it). What sort of problems might ensue?

## 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.




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.

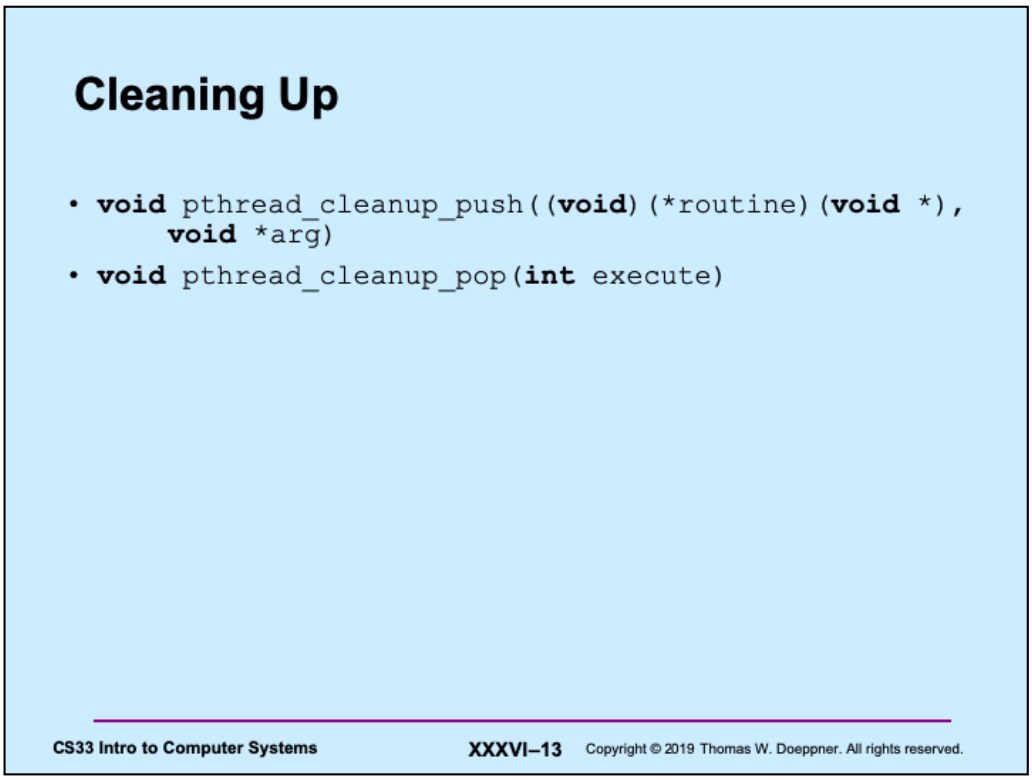
## Cancellation Points

- 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.

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.




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*.

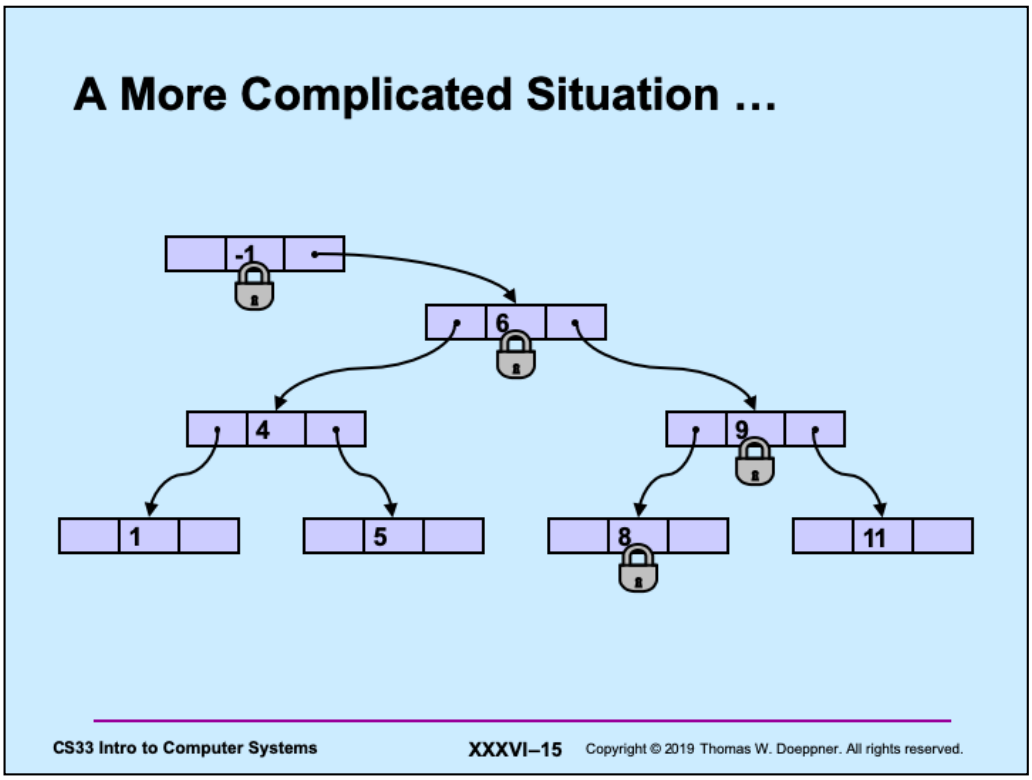
## Sample Code, Revisited

```
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. Note that our example has just one cancellation point: *read*. The cleanup handler iterates through the list, deleting each element.




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);
}
```

# 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);
}
```

**Quiz 3**

**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

# Start/Stop


- **Start/Stop interface**

```
void wait_for_start(state_t *s){
  pthread_mutex_lock(&s->mutex);
  pthread_cleanup_push(
    pthread_mutex_unlock, &s);
  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);
}
```

## Cancellation and Conditions

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

// … (code perhaps 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.

# 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 ...

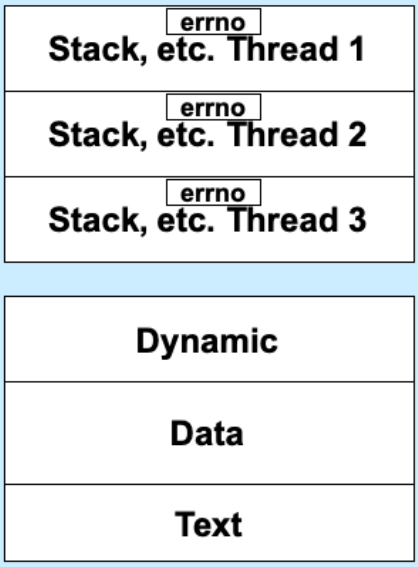
```
#define errno (*__errno_location())
```

- __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.

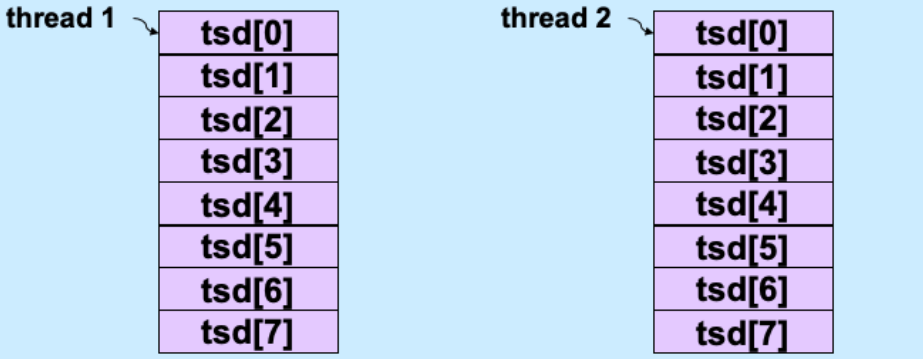
# Process Address Space



# 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**



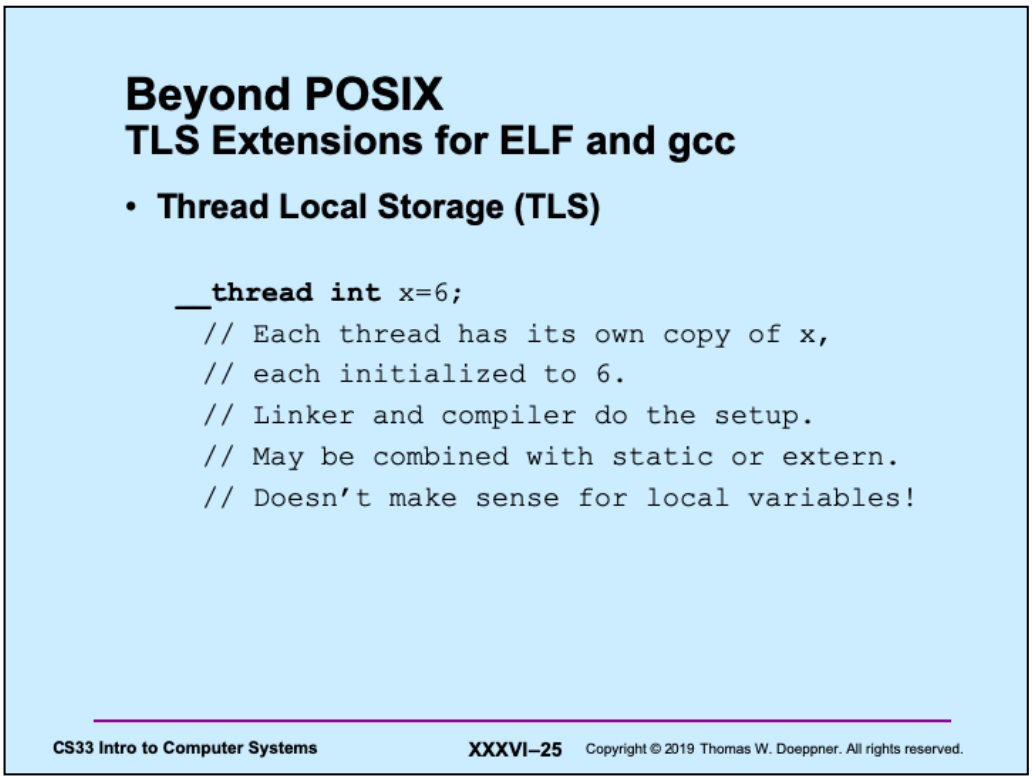

 

## Some Machinery

- `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).*




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 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.

## 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");
  …
}
```

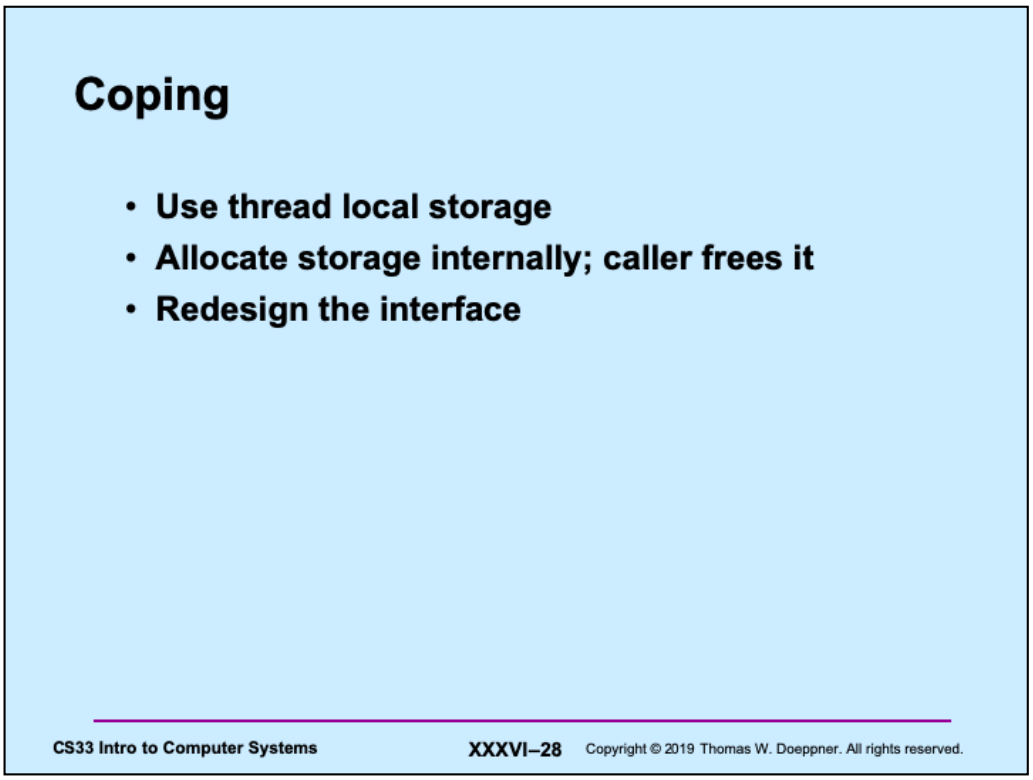
In this example, we put together per-thread windows for thread output. Threads call *getWindow* to set up a window for their exclusive use, then call *threadWrite* to send output to their windows. Individual threads can now set up their own windows and write to them without having to pass around information describing their windows.

## Static Local Storage

```
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. 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*.




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
```

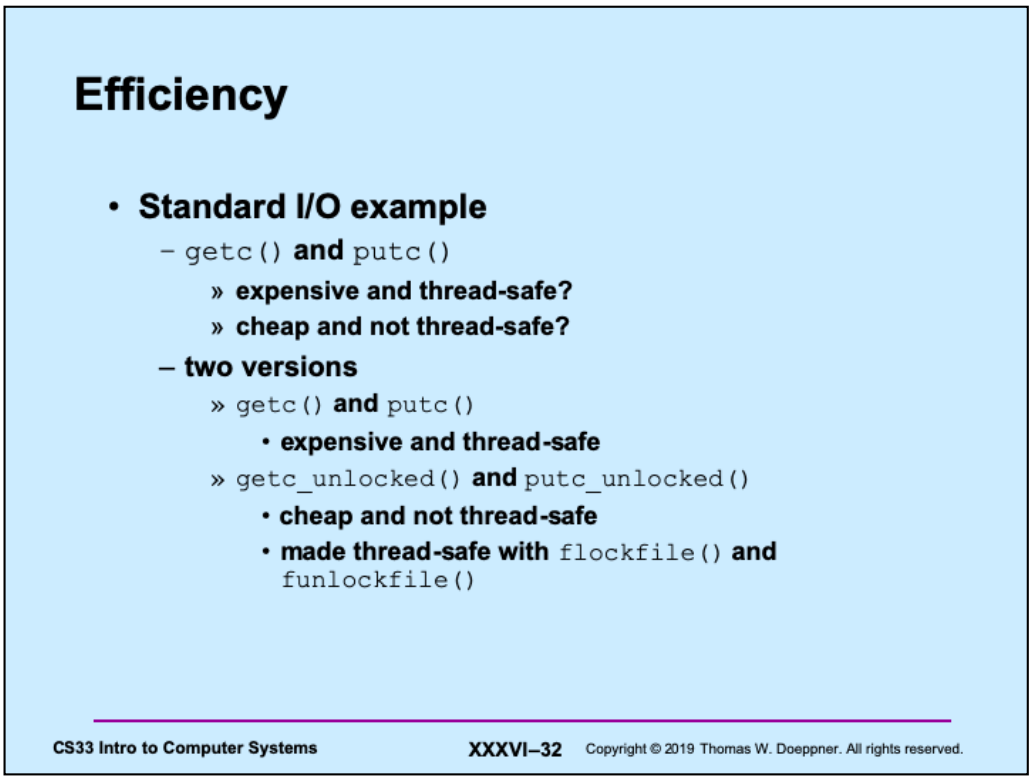
Yet another problem that arises when using libraries that were not designed for multithreaded programs concerns synchronization. The slide shows what might happen if one relied on the single-threaded versions of the standard I/O routines.

## 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 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

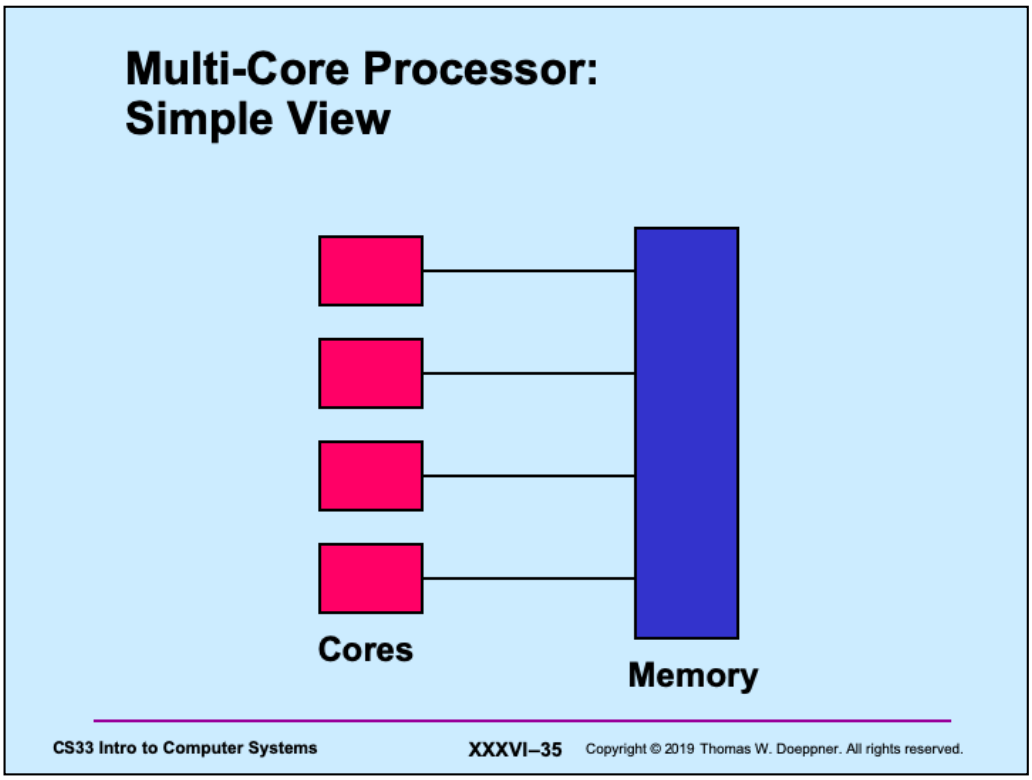
- **Naive**

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

- **Efficient**

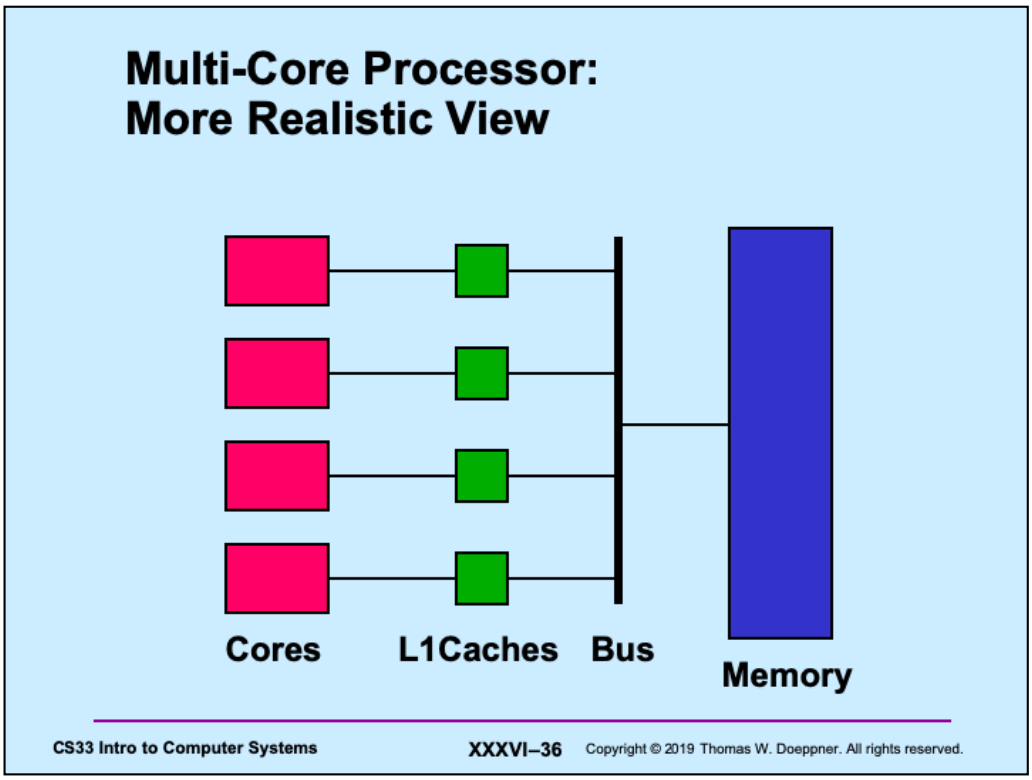
```
flockfile(stdout);
for(i=0; i<lim; i++)
  putc_unlocked(out[i]);
funlockfile(stdout);
```


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



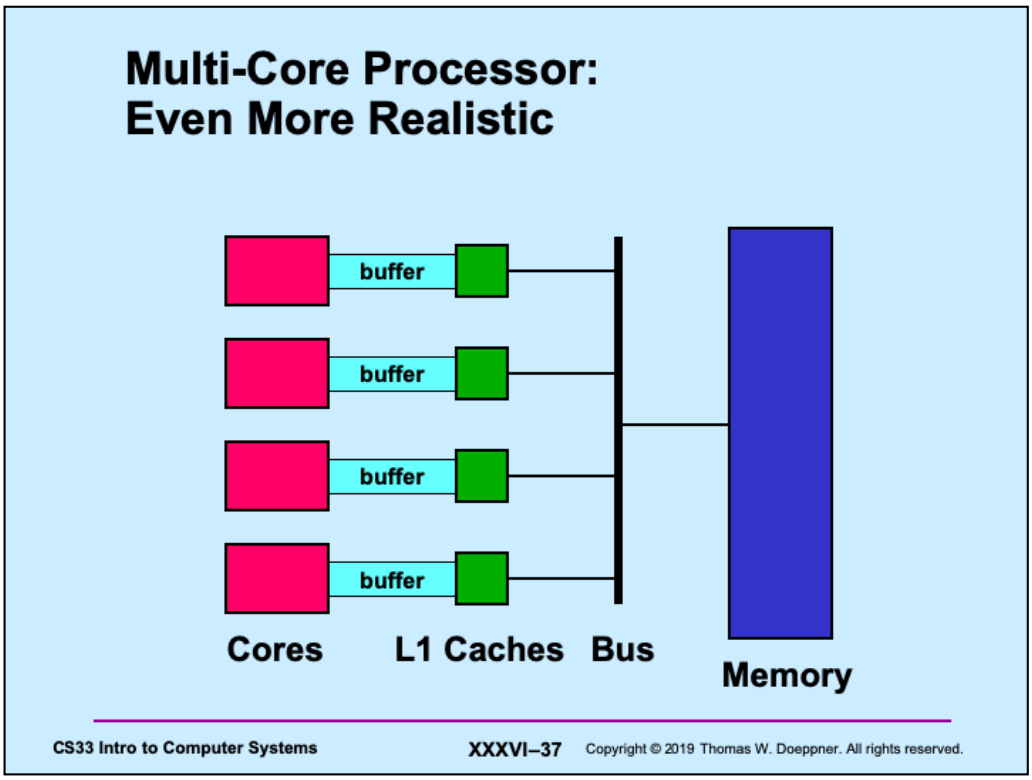

This slide illustrates the common view of the architecture of a multi-core processor: a number of processors are all directly connected to the same memory (which they share). If one core (or processor) stores into a storage location and immediately thereafter another core loads from the same storage location, the second core loads exactly what the first core stored.

Unfortunately, as we learned earlier in the course, things are not quite so simple.



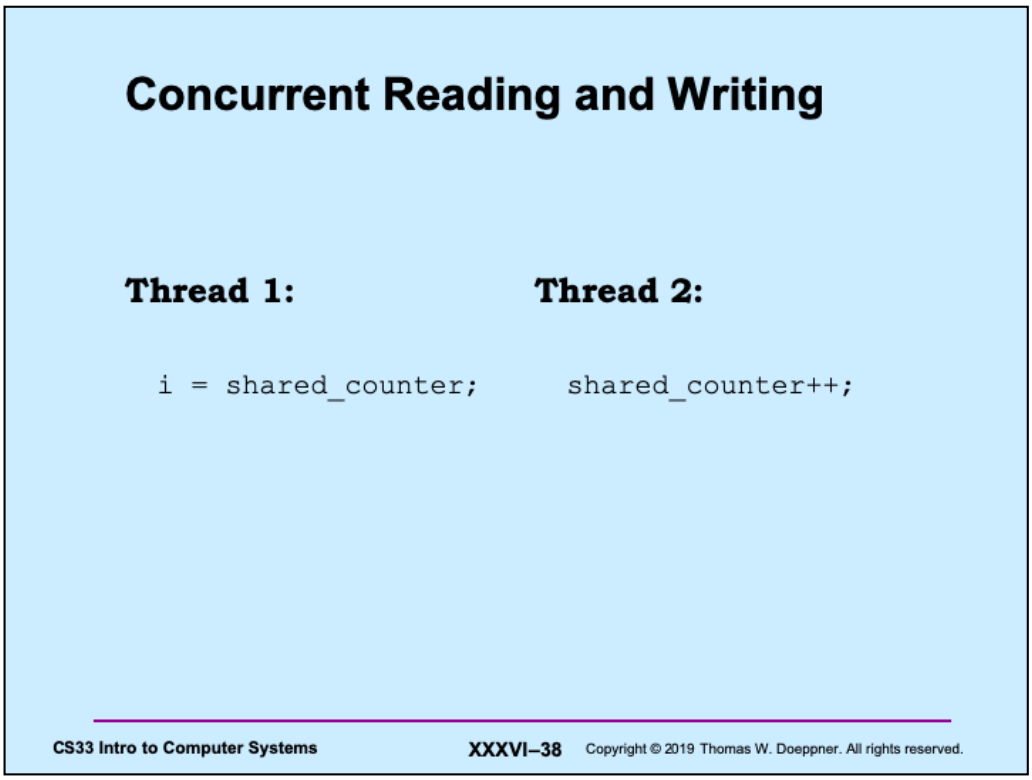

Real multi-core processors have L1 caches that sit between each core and the memory bus; there is a single connection between the bus and the memory. When a core issues a store, the store affects the L1 cache. When a core issues a load, the load is dealt with by the L1 cache if possible, and otherwise goes to memory (perhaps via a shared L2 cache). Most architectures have some sort of cache-consistency logic to insure that the shared-memory semantics of the previous page are preserved.

However, again as we learned earlier in the course, even this description is too simplistic.




This slide shows an even more realistic model, pretty much the same as what we saw is actually used in recent Intel processors. Between each core and the L1 cache is a buffer. Stores by a core go into the buffer. Sometime later the effect of the store reaches the L1 cache. In the meantime, the core is issuing further instructions. Loads by the core are handled from the buffer if the data is still there; otherwise they go to the L1 cache, and then perhaps to memory.

In all instances of this model the effect of a store, as seen by other cores, is delayed. In some instances of this model the order of stores made by one core might be perceived differently by other cores. Architectures with the former property are said to have *delayed stores*; architectures with the latter are said to have *reordered stores* (an architecture could well have both properties).




In this example, one thread running on one processor is loading from an integer in storage; another thread running on another processor is loading from and then storing into an integer in storage. Can this be done safely without explicit synchronization?

On most architectures, the answer is yes. If the integer in question is aligned on a natural (e.g., eight-byte) boundary, then the hardware (perhaps the cache) insures that loads and stores of the integer are atomic.

However, one cannot assume that this is the case on all architectures. Thus a portable program must use explicit synchronization (e.g., a mutex) in this situation.

## Mutual Exclusion w/o Mutexes

```
void peterson(long me) {
  static long loser;                // shared
  static long active[2] = {0, 0};   // shared
  long other = 1 - me;              // private
  active[me] = 1;
  loser = me;
  while (loser == me && active[other])
    ;
  // critical section
  active[me] = 0;
}
```

Shown on the slide is Peterson's algorithm for handling mutual exclusion for two threads without explicit synchronization. (The *me* argument for one thread is 0 and for the other is 1.) This program works given the first two shared-memory models. Does it work with delayed-store architectures?

The algorithm is from "Myths About the Mutual Exclusion Problem," by G. L. Peterson, Information Processing Letters 12(3) 1981: 115–116.

# Quiz 4

```
void peterson(long me) {
  static long loser;                // shared
  static long active[2] = {0, 0};   // shared
  long other = 1 - me;              // private
  active[me] = 1;
  loser = me;
  while (loser == me && active[other])
    ;
  // critical section
  active[me] = 0;
}
```

**This works on sunlab machines.**

**a) true**

**b) false**

## Busy-Waiting Producer/Consumer

```
void producer(char item) {
  while(in - out == BSIZE)
   ;
  buf[in%BSIZE] = item;
  in++;
}
```

```
char consumer( ) {
  char item;
  while(in - out == 0)
   ;
  item = buf[out%BSIZE];
  out++;
  return(item);
}
```

This example is a solution, employing "busy waiting," to the producer-consumer problem for one consumer and one producer.

This solution to the producer-consumer problem is from "Proving the Correctness of Multiprocess Programs," by L. Lamport, IEEE Transactions on Software Engineering, SE-3(2) 1977: 125-143.

# Quiz 5

```
void producer(char item) {

  while(in - out == BSIZE)
    ;

  buf[in%BSIZE] = item;

  in++;
}
```

```
char consumer( ) {
  char item;
  while(in - out == 0)
    ;

  item = buf[out%BSIZE];

  out++;

  return(item);
}
```

**This works on sunlab machines.**

**a) true**

**b) false**

## Coping

- Don't rely on shared memory for synchronization
- Use the synchronization primitives

The point of the previous several slides is that one cannot rely on expected properties of shared memory to eliminate explicit synchronization. Shared memory can behave in some very unexpected ways. However, it is the responsibility of the implementers of the various synchronization primitives to make certain not only that they behave correctly, but also that they synchronize memory with respect to other threads.

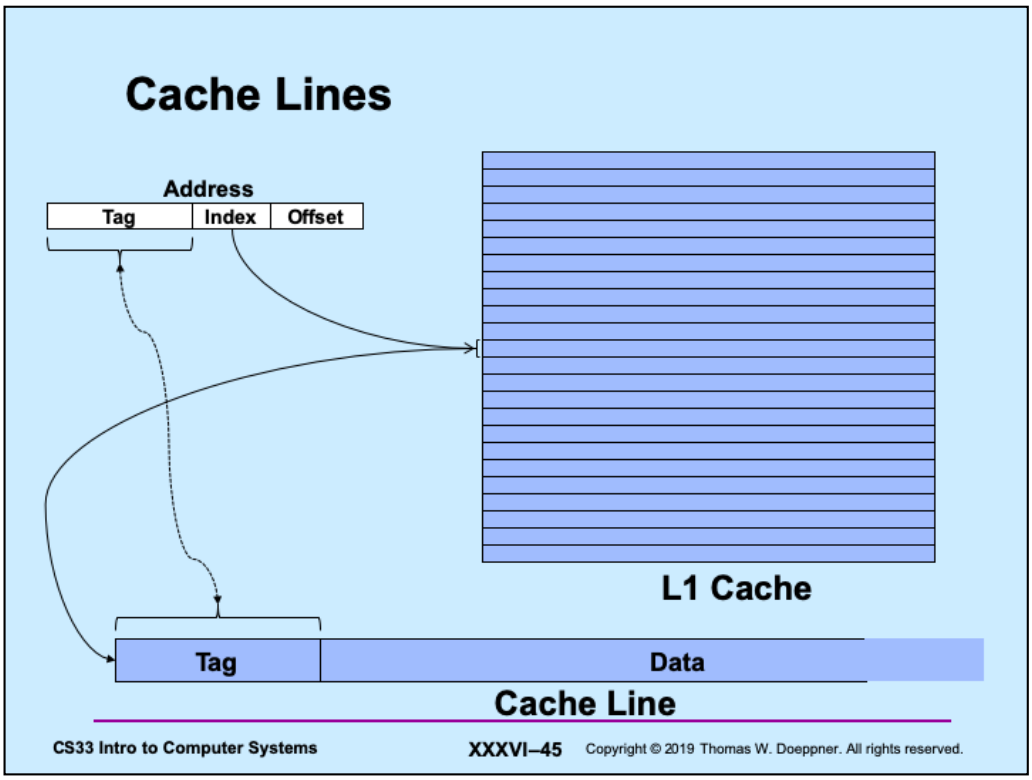
## Which Runs Faster?

```
volatile int a, b;

void *thread1(void *arg) {
  int i;
  for (i=0; i<reps; i++) {
    a = 1;
  }
}

void *thread2(void *arg) {
  int i;
  for (i=0; i<reps; i++) {
    b = 1;
  }
}
```

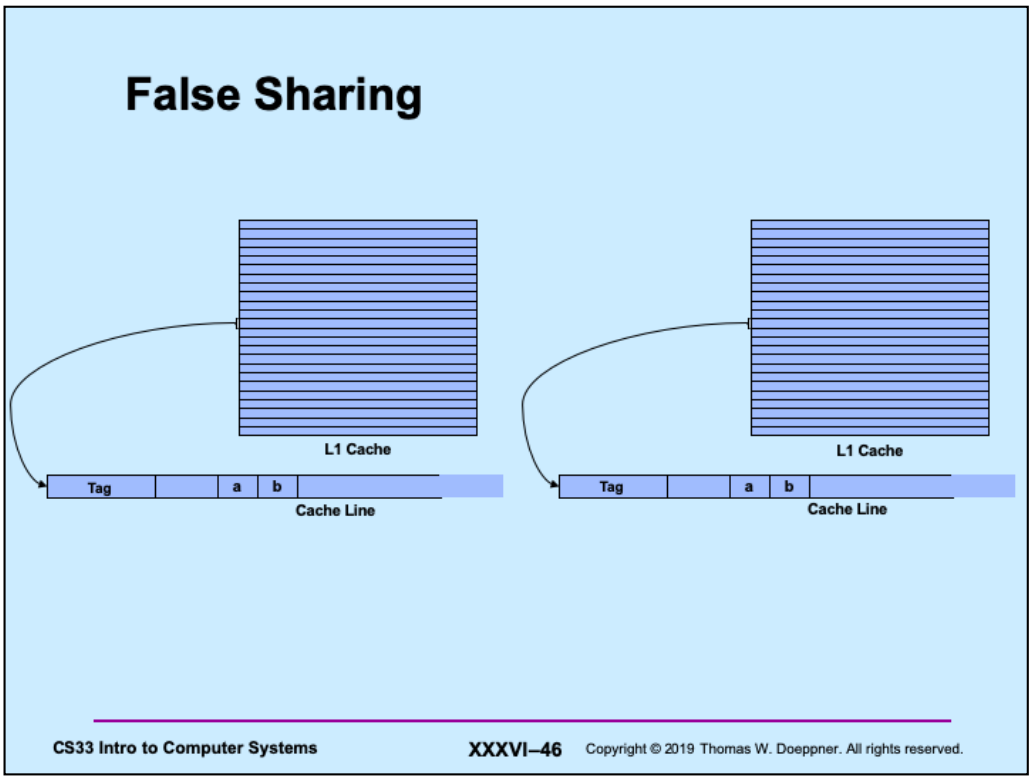
```
volatile int a,
  padding[128], b;

void *thread1(void *arg) {
  int i;
  for (i=0; i<reps; i++) {
    a = 1;
  }
}

void *thread2(void *arg) {
  int i;
  for (i=0; i<reps; i++) {
    b = 1;
  }
}
```

Assume these are run on a two-processor system: why does the two-threaded program on the right run faster than the two-threaded program on the left?




Processors usually employ data caches that are organized as a set of cache lines, typically of 64 bytes in length. Thus data is fetched from and stored to memory in units of the cache-line size. Each processor has its own data cache.




Getting back to our example: we have a two-processor system, and thus two data (L1) caches. If *a* and *b* are in the same cache line, then when either processor accesses *a*, it also accesses *b*. Thus if *a* is modified on processor 1, memory coherency will cause the entire cache line to be invalidated on processor 2. Thus when processor 2 attempts to access *b*, it will get a cache miss and be forced to go to memory to update the cache line containing *b*. From the programmer's perspective, *a* and *b* are not shared. But from the cache's perspective, they are. This phenomenon is known as *false sharing*, and is a source of performance problems.

For further information about false sharing and for tools to deal with it, see http://emeryblogger.com/2011/07/06/precise-detection-and-automatic-mitigation-of-false-sharing-oopsla-11/.