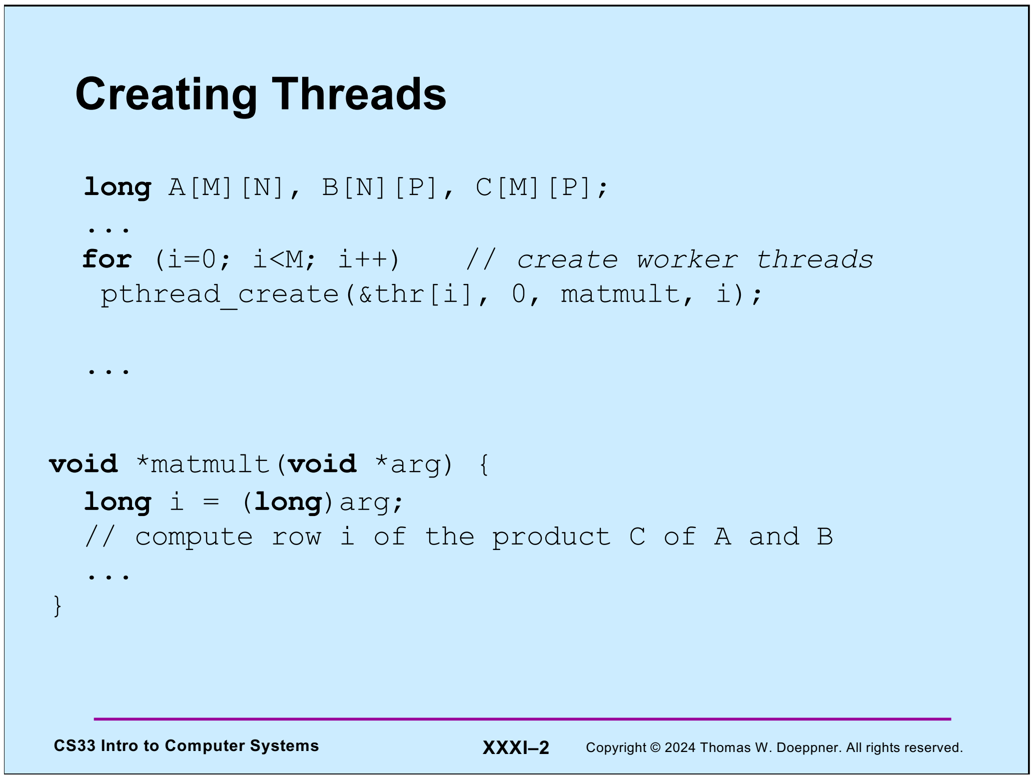
CS 33

Multithreaded Programming (2)



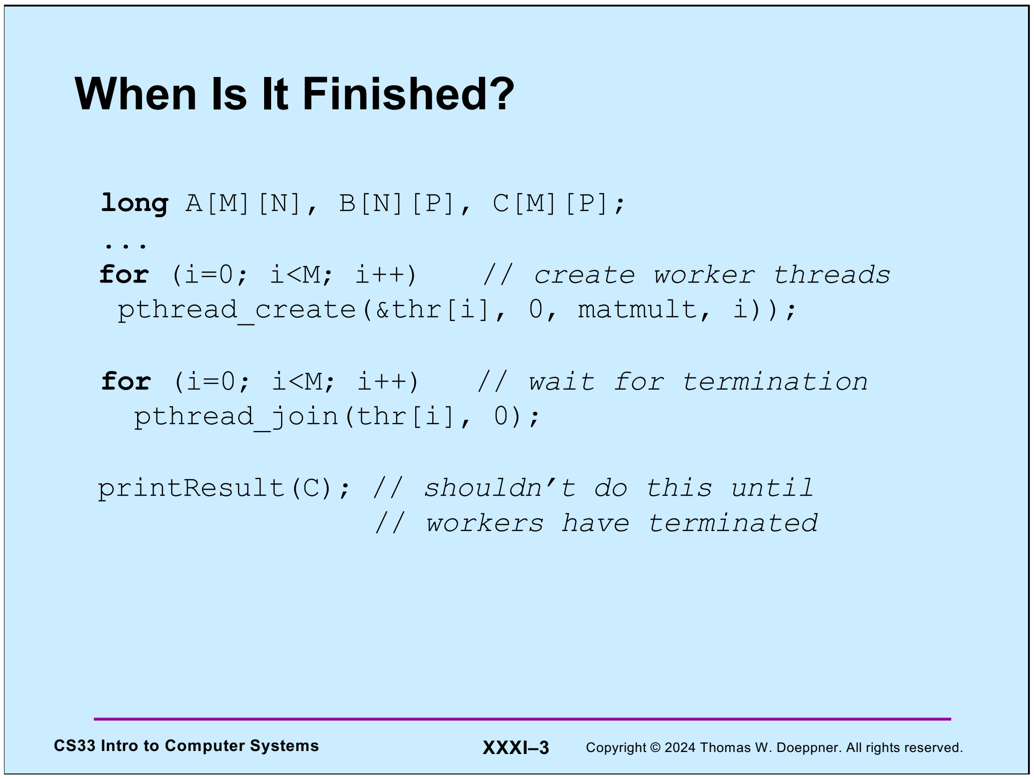
Creating Threads

```
  long A[M][N], B[N][P], C[M][P];
  ...
  for (i=0; i<M; i++)    // create worker threads
   pthread_create(&thr[i], 0, matmult, i);

  ...


void *matmult(void *arg) {
  long i = (long)arg;
  // compute row i of the product C of A and B
  ...
}
```

To create a thread, one calls the **pthread_create** function. This skeleton code for a server application creates a number of threads, each to handle client requests. If **pthread_create** returns successfully (i.e., returns 0), then a new thread has been created that is now executing independently of the caller. This new thread has an ID that is returned via the first parameter. The second parameter is a pointer to an **attributes structure** that defines various properties of the thread. Usually, we can get by with the default properties, which we specify by supplying a null pointer (we discuss this in more detail later). The third parameter is the address of the function in which our new thread should start its execution. The last parameter is the argument that is actually passed to the first function of the thread.

If **pthread_create** fails, it returns a code indicating the cause of the failure.

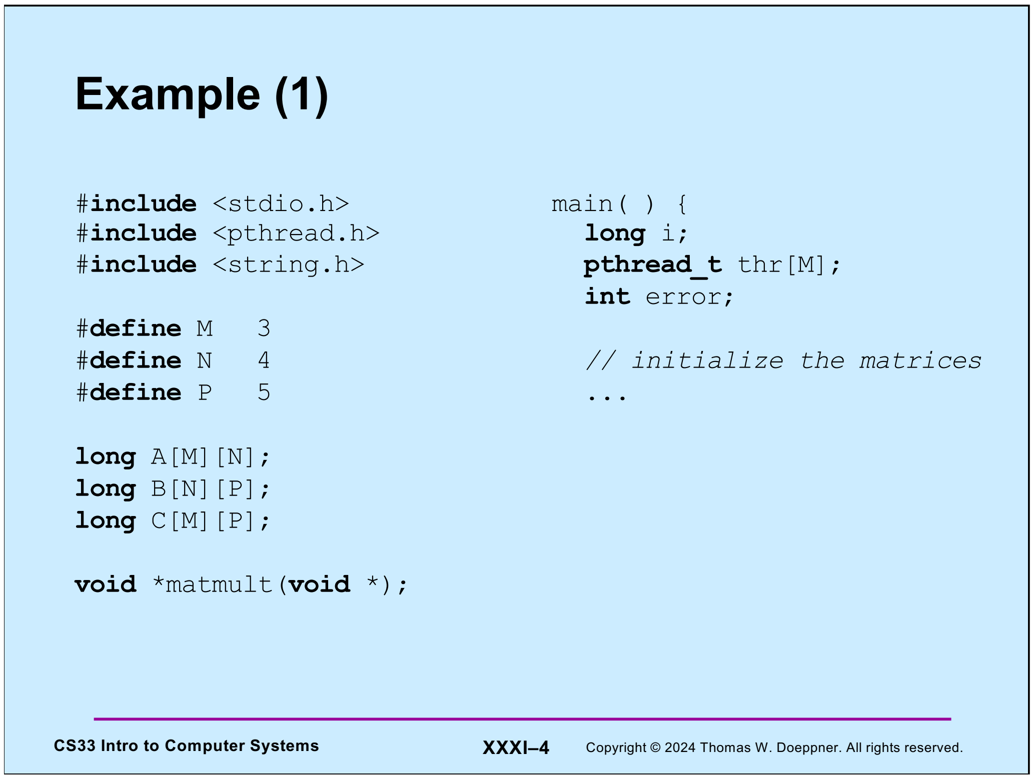
This example in the slide is a sketch of a multi-threaded matrix multiplication program in which we have one thread per row of the product matrix.



When Is It Finished?

```
long A[M][N], B[N][P], C[M][P];
...
for (i=0; i<M; i++)    // create worker threads
 pthread_create(&thr[i], 0, matmult, i));

for (i=0; i<M; i++)   // wait for termination
  pthread_join(thr[i], 0);

printResult(C); // shouldn't do this until
                // workers have terminated
```

We'd like the first thread to be able to print the resulting product matrix C, but it shouldn't attempt to do this until the worker threads have terminated. We have it call **pthread_join** for each of the worker threads, causing it to wait for each worker to terminate.



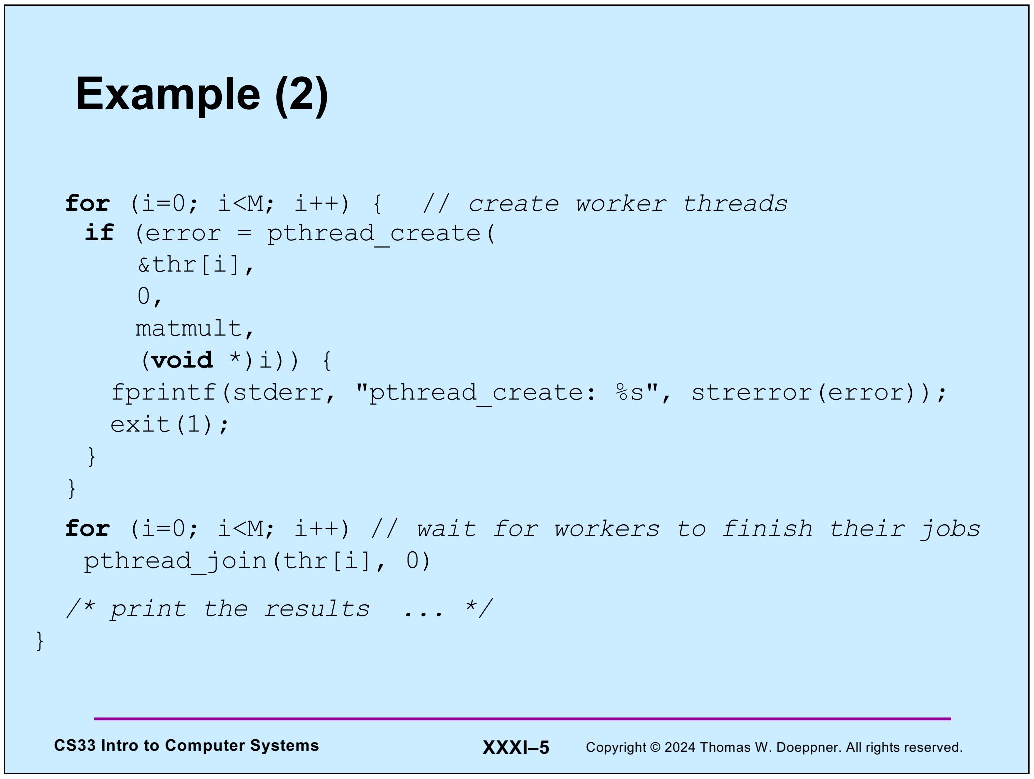
Example (1)

```
#include <stdio.h>
#include <pthread.h>
#include <string.h>

#define M   3
#define N   4
#define P   5

long A[M][N];
long B[N][P];
long C[M][P];

void *matmult(void *);
```

```
main( ) {
  long i;
  pthread_t thr[M];
  int error;

  // initialize the matrices
  ...
```

In this series of slide we show the complete matrix-multiplication program.

This slide shows the necessary includes, global declarations, and the beginning of the main function.

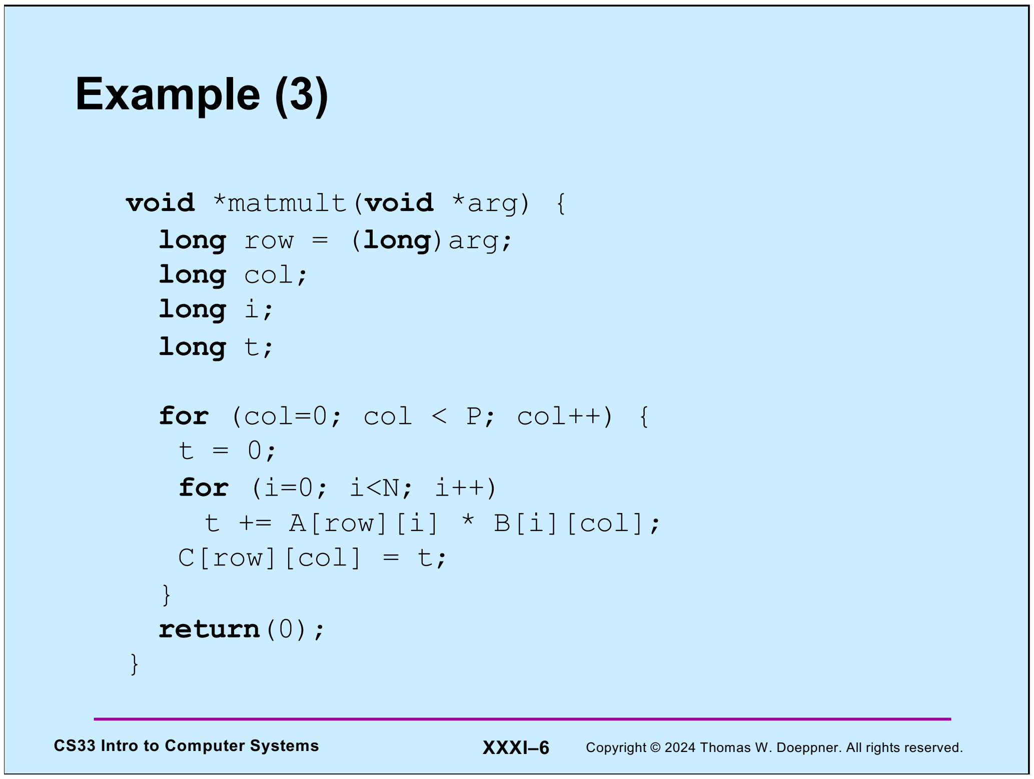


Example (2)

```
  for (i=0; i<M; i++) {   // create worker threads
   if (error = pthread_create(
       &thr[i],
       0,
       matmult,
       (void *)i)) {
     fprintf(stderr, "pthread_create: %s", strerror(error));
     exit(1);
   }
  }
  for (i=0; i<M; i++) // wait for workers to finish their jobs
   pthread_join(thr[i], 0)
  /* print the results  ... */
}
```

Here we have the remainder of **main**. It creates a number of threads, one for each row of the result matrix, waits for all of them to terminate, then prints the results (this last step is not spelled out). Note that we check for errors when calling **pthread_create**. (It is important to check for errors after calls to almost all of the pthread functions, but we normally omit it in the slides for lack of space.) For reasons discussed later, the pthread calls, unlike Unix system calls, do not return -1 if there is an error, but return the error code itself (and return zero on success). Thus one cannot use perror to print error messages (since perror uses errno to determine what the error was) Instead, one passes the error code to strerror, which returns a pointer to the error message.

So that the first thread is certain that all the other threads have terminated, it must call **pthread_join** on each of them.



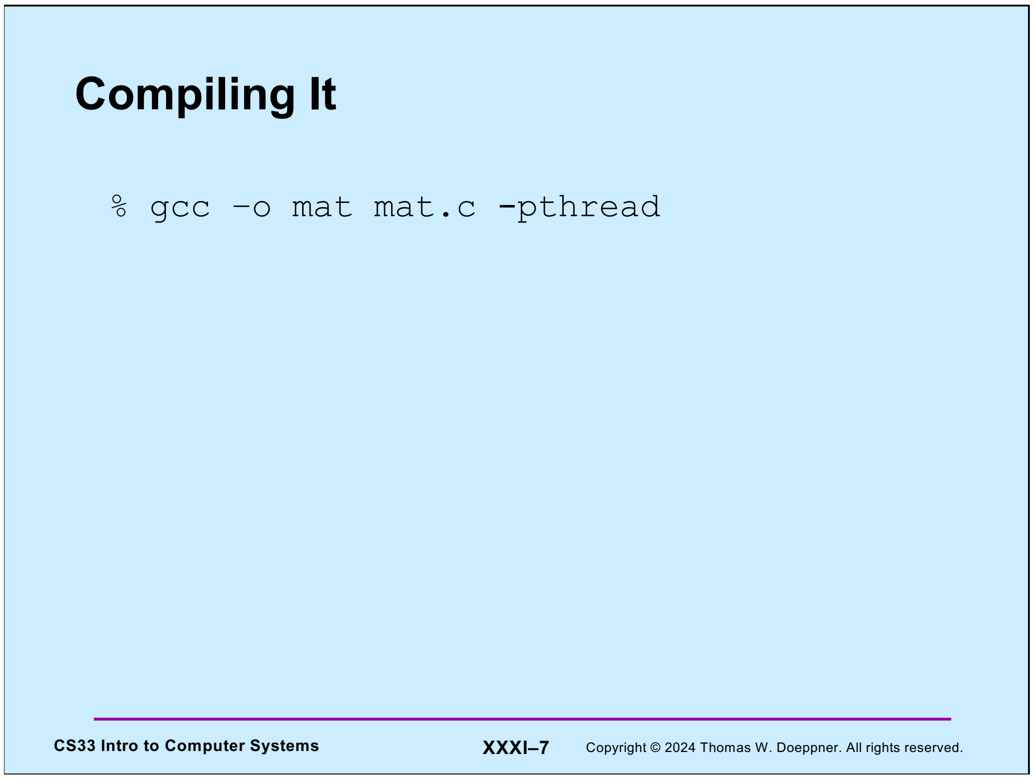
Example (3)

```
void *matmult(void *arg) {
  long row = (long)arg;
  long col;
  long i;
  long t;

  for (col=0; col < P; col++) {
   t = 0;
   for (i=0; i<N; i++)
     t += A[row][i] * B[i][col];
   C[row][col] = t;
  }
  return(0);
}
```

Here is the code executed by each of the threads. It’s pretty straightforward: it merely computes a row of the result matrix.

Note how the argument is explicitly converted from **void *** to **long**.

This code does not make optimal use of the cache. How can it be restructured so it does?

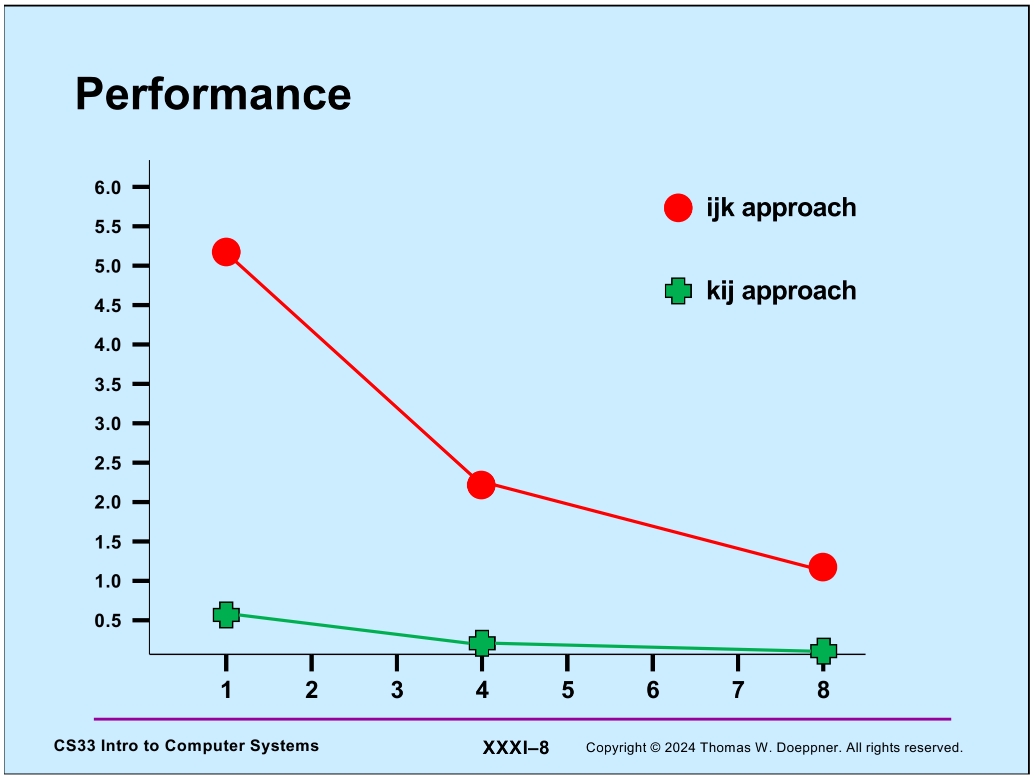


Compiling It

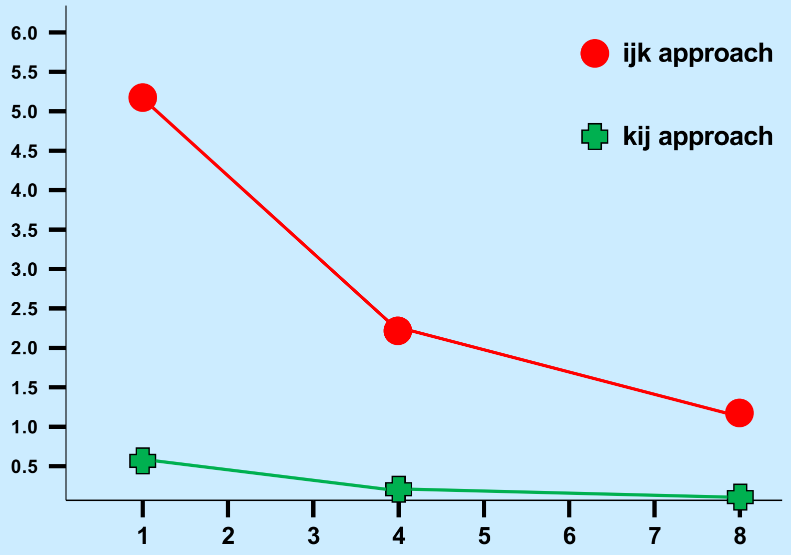
```
% gcc –o mat mat.c -pthread
```

Providing the –pthread flag to gcc is equivalent to providing all the following flags:

- -lpthread: include libpthread.so — the POSIX threads library
- -D_REENTRANT: defines certain things relevant to threads in stdio.h — we cover this later.
- -Dotherstuff, where "otherstuff" is a variety of flags required to get the correct versions of declarations for POSIX threads in a number of header (.h) files.

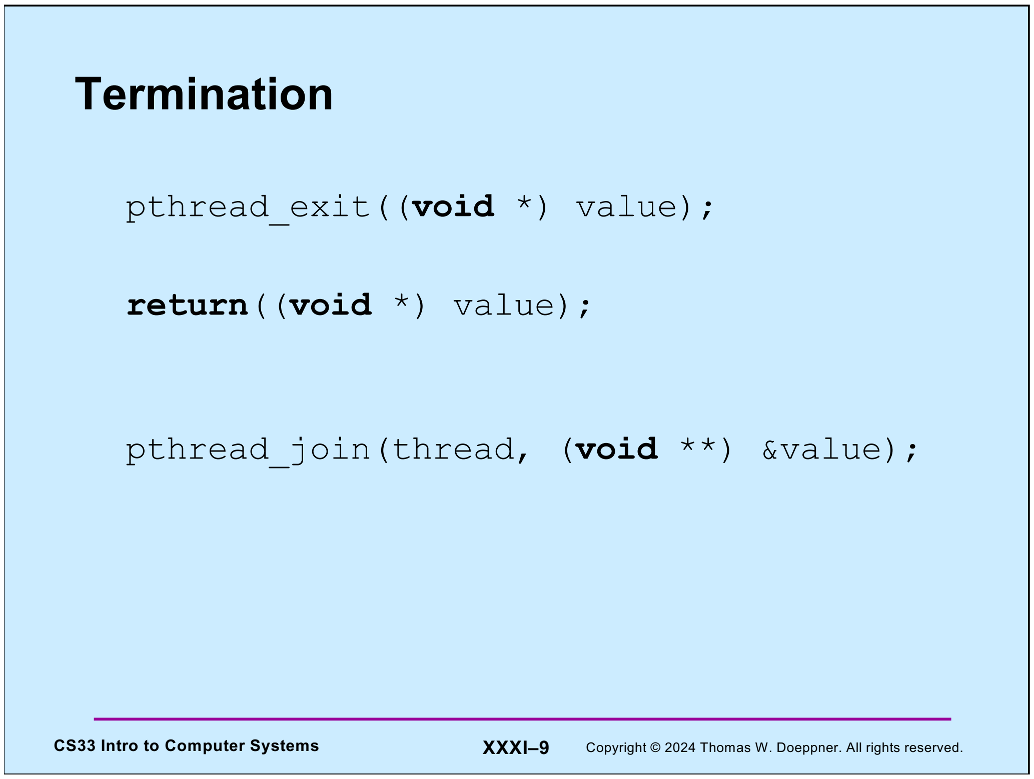


Performance



The slide shows the performance of multithreaded integer matrix multiplication on a Mac Studio with a 10-core Apple M1 Max chip. Two 2048x2048 matrices were multiplied. The x axis shows the number of threads used, the y axis shows the time in minutes. The algorithms are from lecture 17. The ijk approach results in 1.125 cache misses per iteration, while the kij approach results in .25 cache misses per iteration. What's interesting is that the speedup from using eight cores rather than just one is not as great as the speedup from having substantially fewer cache misses per iteration.

The results from using 16 and 32 threads are about the same as from using 8 threads.



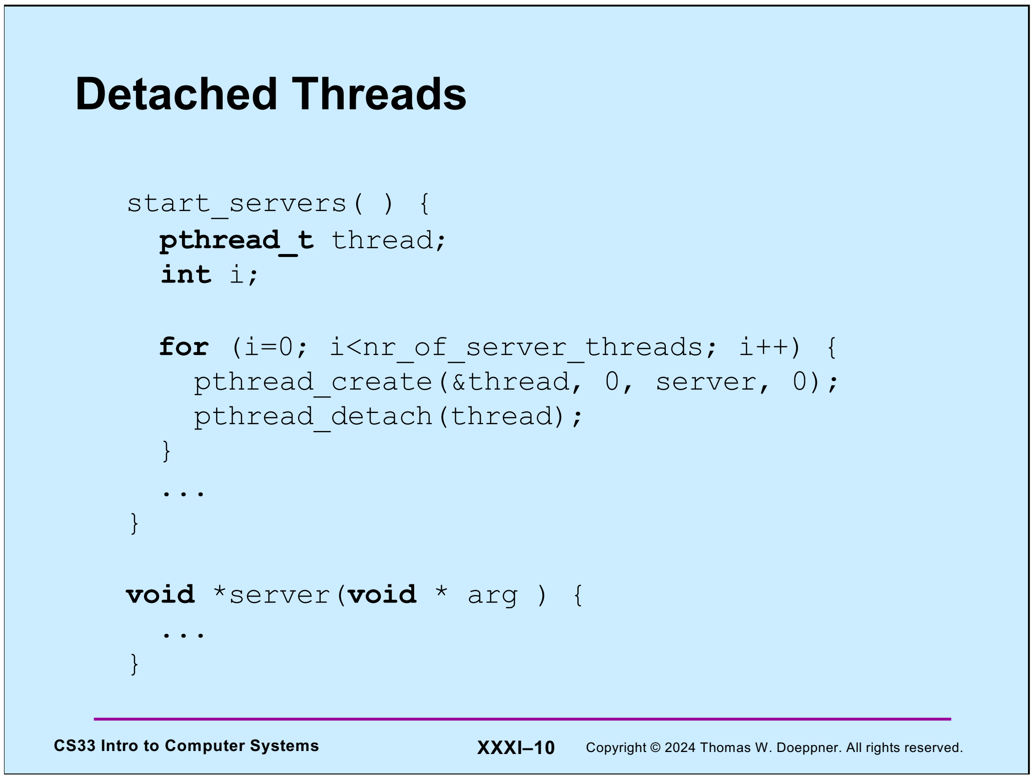
Termination

```
pthread_exit((void *) value);

return((void *) value);


pthread_join(thread, (void **) &value);
```

A thread terminates either by calling ***pthread_exit*** or by returning from its first function. 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 among 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** on the same target thread 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 functions, they merely terminate themselves.

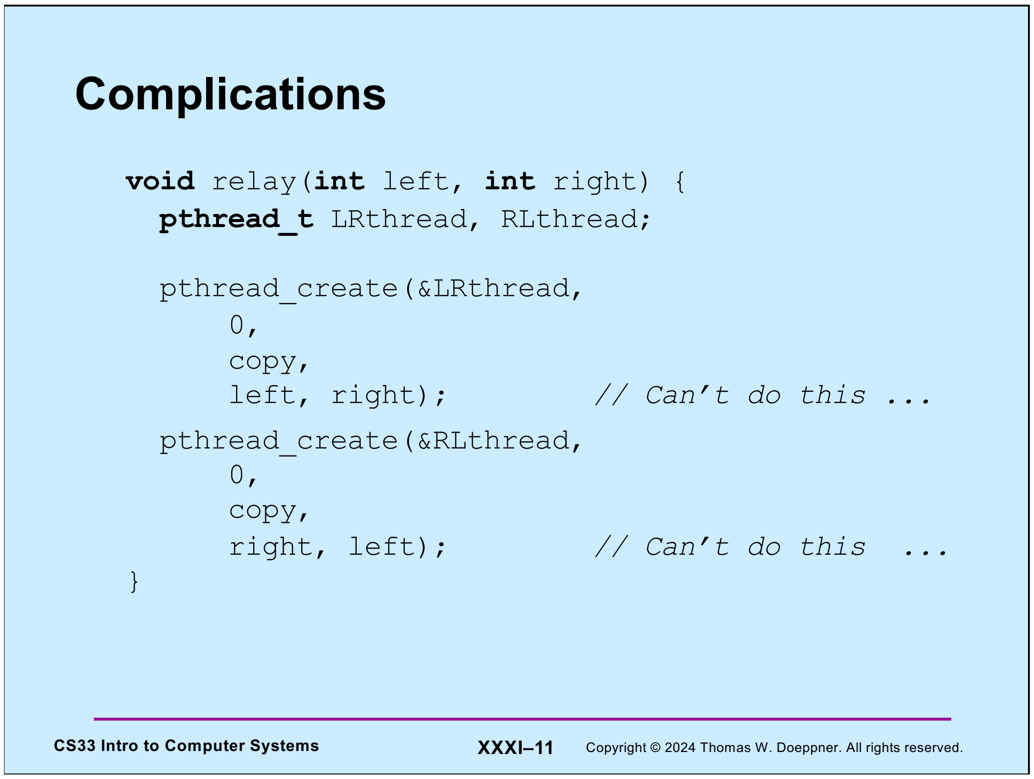
If no thread calls **exit** and no thread returns from main, then the process should terminate once all threads have terminated (i.e., have called **pthread_exit** or, for threads other than the first one, have returned from their first function). If the first thread calls **pthread_exit**, it self-destructs, but does not cause the process to terminate (unless no other threads are extant).



Detached Threads

```
start_servers( ) {
  pthread_t thread;
  int i;

  for (i=0; i<nr_of_server_threads; i++) {
    pthread_create(&thread, 0, server, 0);
    pthread_detach(thread);
  }
  ...
}

void *server(void * arg ) {
  ...
}
```

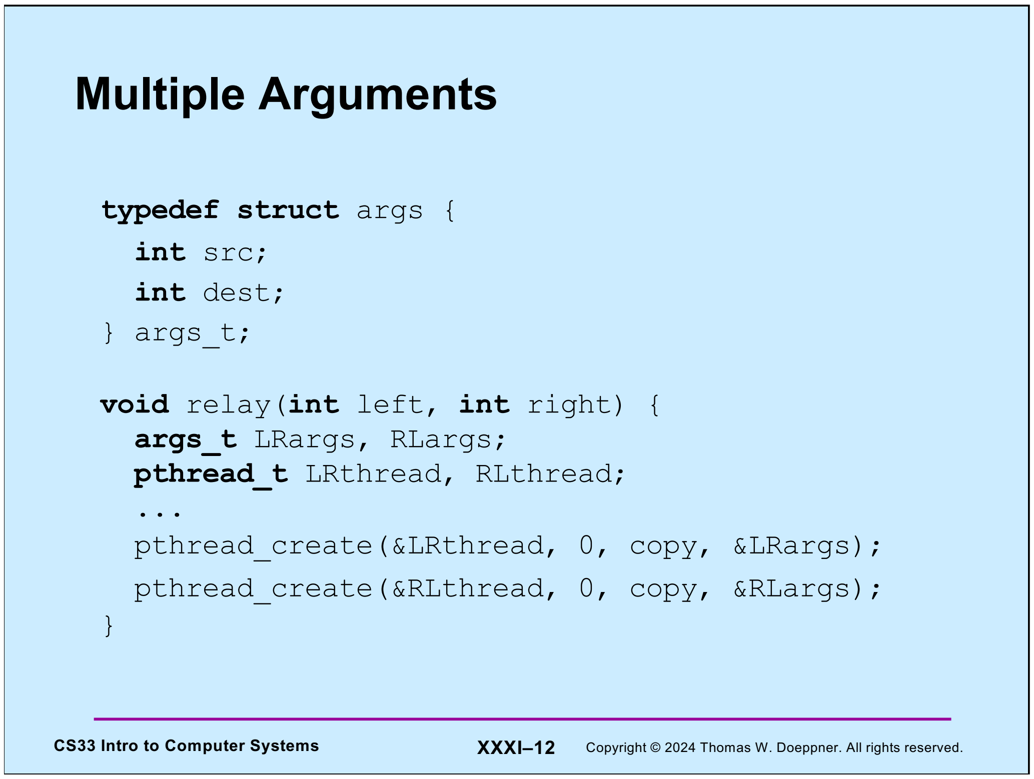
If there is no reason to synchronize with the termination of a thread, then it is rather a nuisance to have to call **pthread_join**. Instead, one can arrange for a thread to be **detached**. Such threads “vanish” when they terminate — not only do they not need to be joined, but they cannot be joined.



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 function 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);
}
```

To pass more than one argument to the first function 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

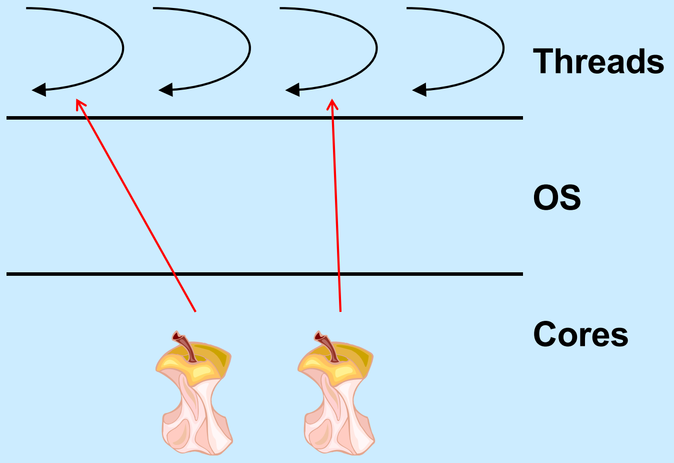
```
typedef struct args {
  int src;
  int dest;
} args_t;

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?

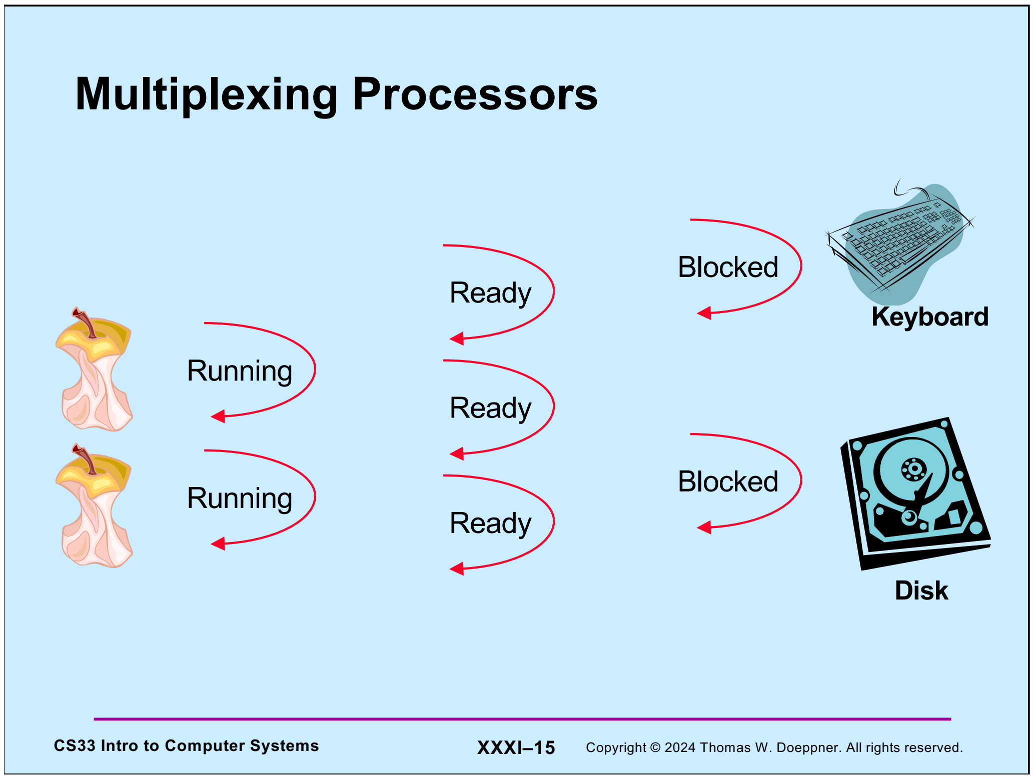
a) **no**
b) **yes**
c) **it depends upon the word size**

Execution



The operating system is responsible for multiplexing the execution of threads on the available processors. The OS's **scheduler** is responsible for assigning threads to processors (cores). Periodically, say every millisecond, each processor is interrupted and calls upon the OS to determine if another thread should run. If so, the current thread on the processor (core) is preempted in favor of the next thread. Assuming all threads are treated equally, over a sufficient period of time each thread gets its fair share of available processor time. Thus, even though a system may have only one core, all threads make progress and give the appearance of running simultaneously.

This notion of multiplexing threads on the available processors is known as **time slicing**. The amount of time that a thread runs before yielding to another is known as its **time slice**. The length of time slices is typically measured in milliseconds, while the time required for a processor to switch from one thread to another is typically measured in microseconds.



Multiplexing Processors

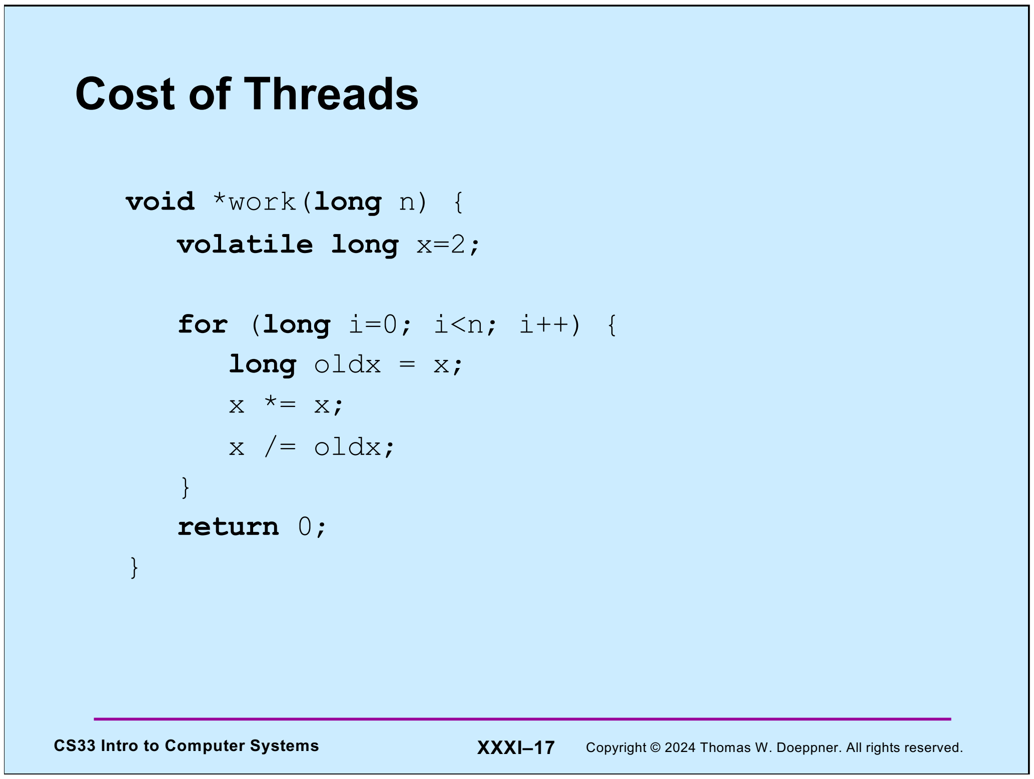
To be a bit more precise about scheduling, let's define some more (standard) terms. Threads are in either a **blocked** state or a **ready** state: in the former they cannot be assigned a core, in the latter they can. The scheduler determines which ready threads should be assigned cores. Ready threads that have been assigned cores are called **running** threads.

Quiz 2

```
  pthread_create(&tid, 0, tproc, (void *)1);
  pthread_create(&tid, 0, tproc, (void *)2);

  printf("T0\n");

  ...

void *tproc(void *arg) {
  printf("T%dl\n", (long)arg);
  return 0;
}
```

In which order are things printed?

- **a) indeterminate**
- **b) T2, T1, T0**
- **c) T0, T1, T2**
- **d) T1, T2, T0**



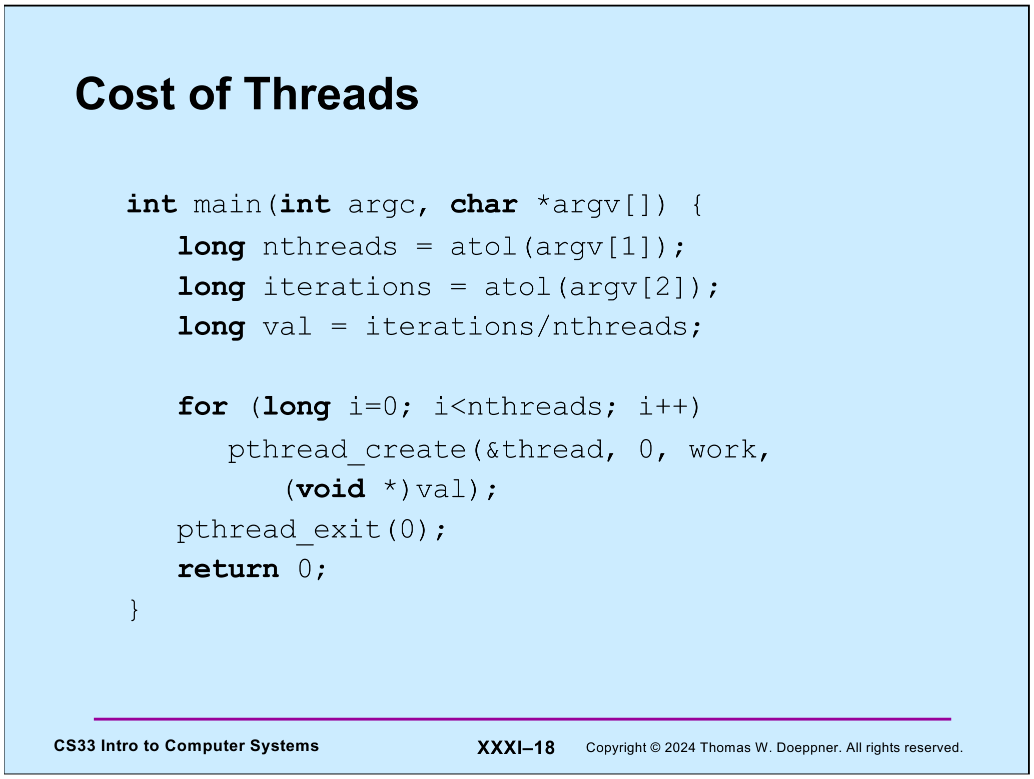
Cost of Threads

```
void *work(long n) {
   volatile long x=2;

   for (long i=0; i<n; i++) {
      long oldx = x;
      x *= x;
      x /= oldx;
   }
   return 0;
}
```

This function, called **work**, does nothing useful other than consuming cpu cycles. It executes for a period of time that's proportional to its argument. We use it to get an idea of the performance cost of using threads.

If work is called with an argument of W, its loop executes W times. Suppose work is called by each of T threads, each supplying an argument of W/T. If we sum the number of executions of the loop over all T threads, we find that there are a total of $(W/T)\cdot T = W$ executions of the loop. Thus comparing the time it takes one thread to do W iterations with the time it takes T threads to do a total of W iterations gives us a means for determining the cost of using threads. If, on a single core, one thread doing W iterations takes w_1 seconds, but T threads each doing W/T iterations takes w_2 seconds, and $w_2 > w_1$, then their difference is the additional time required due to the use of multiple threads.

Recall that the **volatile** attribute ensures that a variable will not be stored in a register, but that all references to it will be to memory. The reason for using the attribute here is to make certain that the function is accessing memory in each iteration, so that we can see the effect of this access on the execution of other threads.



Cost of Threads

```
int main(int argc, char *argv[]) {
   long nthreads = atol(argv[1]);
   long iterations = atol(argv[2]);
   long val = iterations/nthreads;

   for (long i=0; i<nthreads; i++)
      pthread_create(&thread, 0, work,
          (void *)val);
   pthread_exit(0);
   return 0;
}
```

Here’s our main function for doing the experiment described in the previous slide.

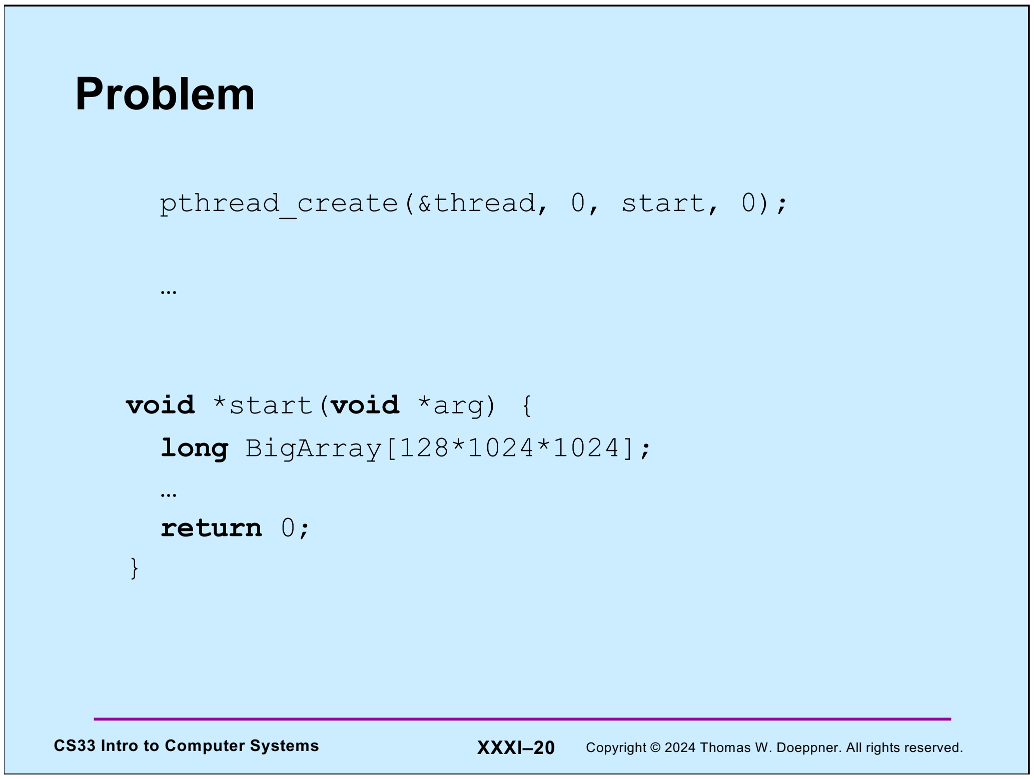
Cost of Threads

```
void *work(long n) {
   volatile long x=2;

   for (long i=0; i<n; i++) {
      long oldx = x;
      x *= x;
      x /= oldx;
   }
   return 0;
}
```

Not a Quiz

This code runs in time *n* on a 6-core processor when *nthreads* is 6. It runs in time *p* on the same processor when *nthreads* is 1000.

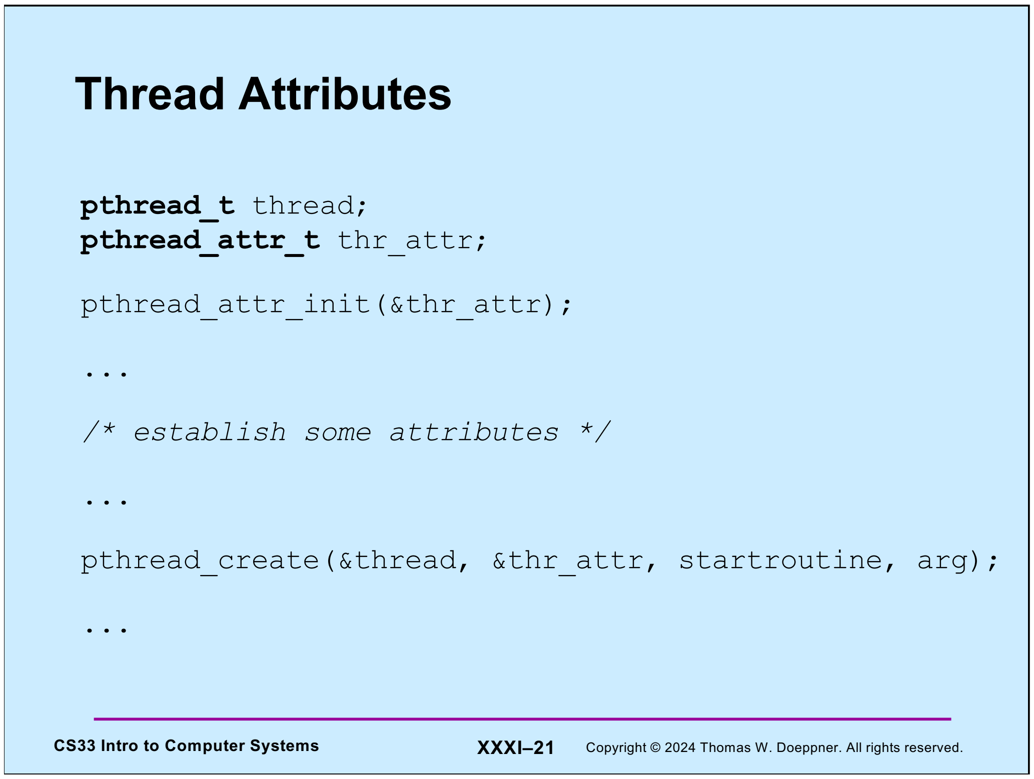
a) ***n* << *p* (slower)**
b) ***n* ≈ *p* (same speed)**
c) ***n* >> *p* (faster)**



Problem

```
  pthread_create(&thread, 0, start, 0);

  …


void *start(void *arg) {
  long BigArray[128*1024*1024];
  …
  return 0;
}
```

Here we are creating a thread that has a very large local variable that, of course, is allocated on the thread’s stack. How can we be sure that the thread’s stack is actually big enough? As it turns out, the default stack size for threads in Linux is two megabytes.



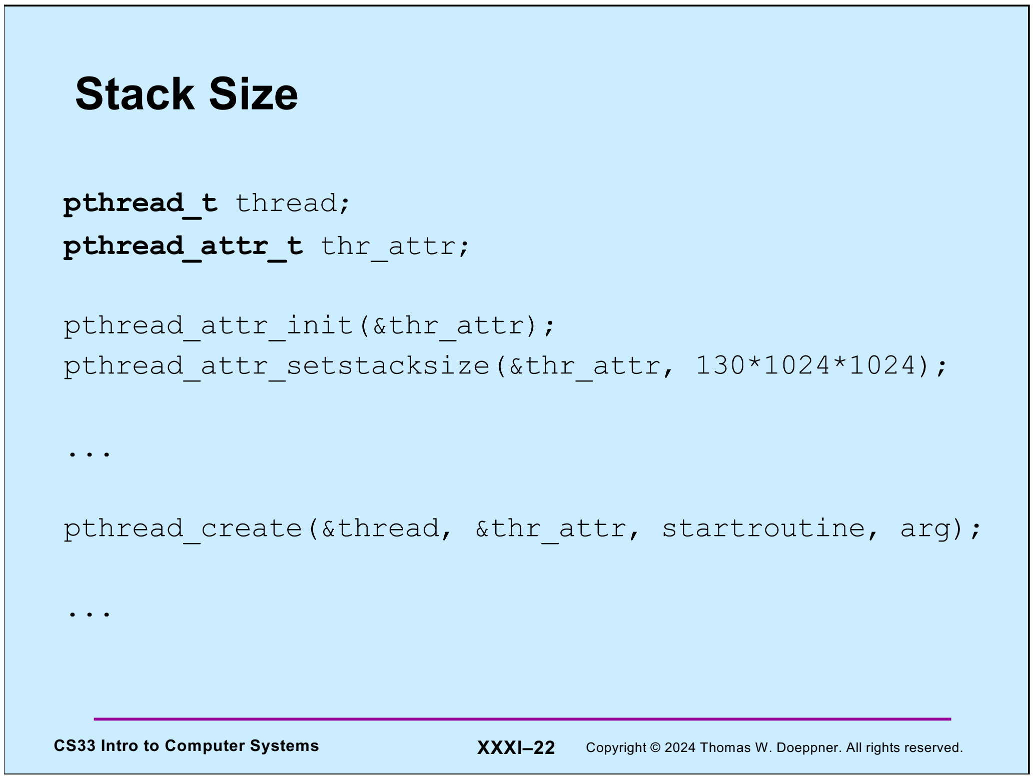
Thread Attributes

```
pthread_t thread;
pthread_attr_t thr_attr;

pthread_attr_init(&thr_attr);

...

/* establish some attributes */

...

pthread_create(&thread, &thr_attr, startroutine, arg);

...
```

A number of properties of a thread can be specified via the **attributes** argument when the thread is created. Some of these properties are specified as part of the POSIX specification, others are left up to the implementation. By burying them inside the attributes structure, we make it straightforward to add new types of properties to threads without having to complicate the parameter list of **pthread_create**. To set up an attributes structure, one must call **pthread_attr_init**. As seen in the next slide, one then specifies certain properties, or attributes, of threads. One can then use the attributes structure as an argument to the creation of any number of threads.

Note that the attributes structure only affects the thread when it is created. Modifying an attributes structure has no effect on already-created threads, but only on threads created subsequently with this structure as the attributes argument.

Storage may be allocated as a side effect of calling **pthread_attr_init**. To ensure that it is freed, call **pthread_attr_destroy** with the attributes structure as argument. Note that if the attributes structure goes out of scope, not all storage associated with it is necessarily released — to release this storage you must call **pthread_attr_destroy**.

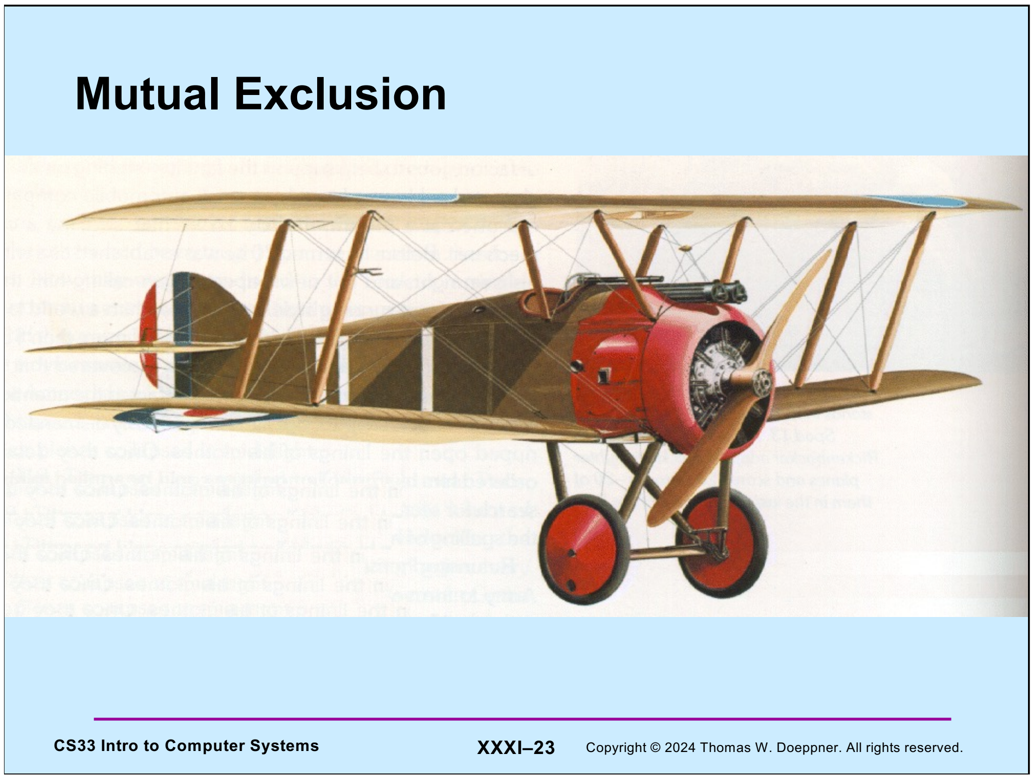

Stack Size

```
pthread_t thread;
pthread_attr_t thr_attr;

pthread_attr_init(&thr_attr);
pthread_attr_setstacksize(&thr_attr, 130*1024*1024);

...

pthread_create(&thread, &thr_attr, startroutine, arg);

...
```

Among the attributes that can be specified is a thread's **stack size**. The default attributes structure specifies a stack size that is probably good enough for "most" applications. How big is it? While the default stack size is not mandated by POSIX, in Linux it is two megabytes. To establish a different stack size, use the **pthread_attr_setstacksize** function, as shown in the slide.

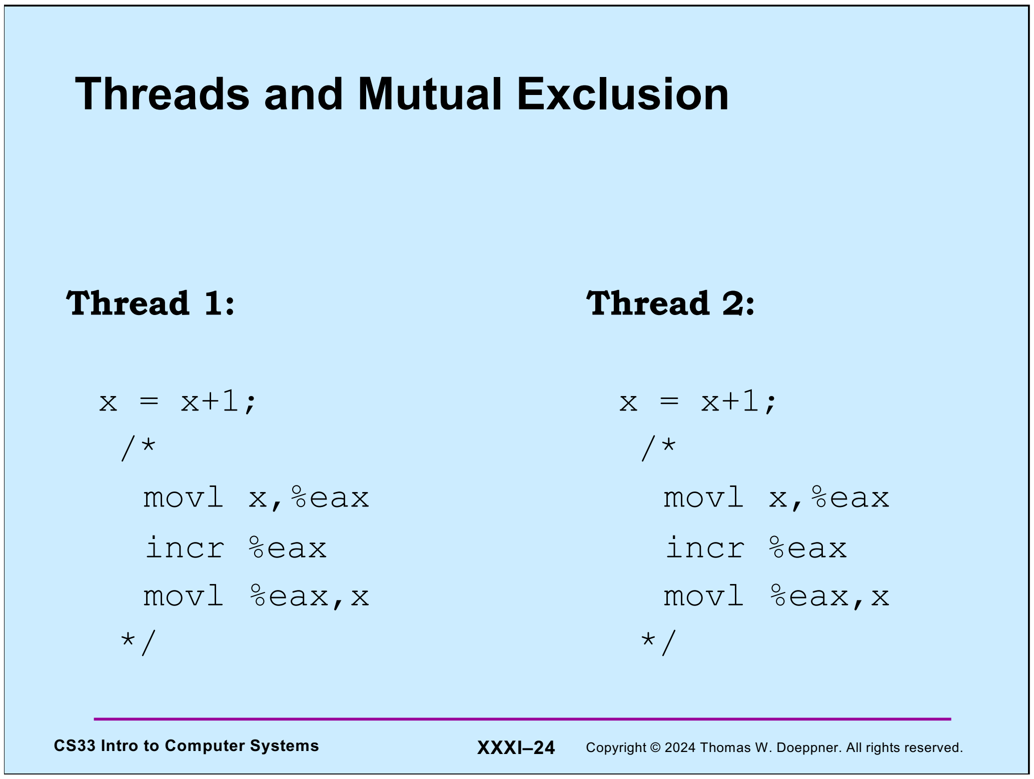
How large a stack is necessary? The answer, of course, is that it depends. If the stack size is too small, there is the danger that a thread will attempt to overwrite the end of its stack. There is no problem with specifying too large a stack, except that, on a 32-bit machine, one should be careful about using up too much address space (one thousand threads, each with a one-megabyte stack, use a fair portion of the address space).

What happens if a thread uses more stack space than was allotted to it? It would probably clobber memory holding another thread's stack, which could lead to some rather difficult to debug problems. To guard against such happenings, The lowest-address page of a thread's stack (recall that stacks grow downwards) is made inaccessible, meaning that any reference to it will generate a fault. Thus, if the thread references just beyond its allotted stack, there will be a fault which, though not good, makes it clear that this thread has exceeded its stack space.



Mutual Exclusion

The mutual-exclusion problem involves making certain that two things don’t happen at once. A non-computer example arose in the fighter aircraft of World War I (pictured is a Sopwith Camel). Due to a number of constraints (e.g., machine guns tended to jam frequently and thus had to be close to people who could unjam them), machine guns were mounted directly in front of the pilot. However, blindly shooting a machine gun through the whirling propeller was not a good idea — one was apt to shoot oneself down. At the beginning of the war, pilots politely refrained from attacking fellow pilots. A bit later in the war, however, the Germans developed the tactic of gaining altitude on an opponent, diving at him, turning off the engine, then firing without hitting the now-stationary propeller. Today, this would be called **coarse-grained synchronization**. Later, the Germans developed technology that synchronized the firing of the gun with the whirling of the propeller, so that shots were fired only when the propeller blades would not be in the way. This is perhaps the first example of a mutual-exclusion mechanism providing **fine-grained synchronization**.



Threads and Mutual Exclusion

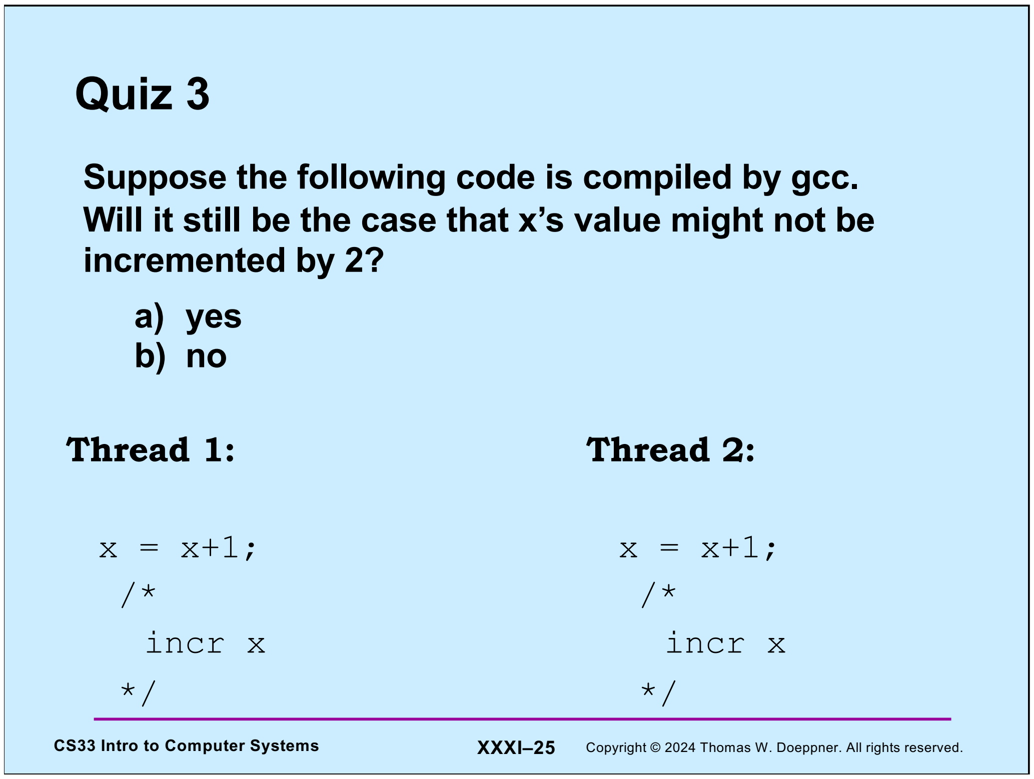
Thread 1:

```
x = x+1;
 /*
  movl x,%eax
  incr %eax
  movl %eax,x
 */
```

Thread 2:

```
x = x+1;
 /*
  movl x,%eax
  incr %eax
  movl %eax,x
 */
```

Here we have two threads that are reading and modifying the same variable: both are adding one to *x*. Although the operation is written as a single step in terms of C code, it might take three machine instructions, as shown in the slide. If the initial value of *x* is 0 and the two threads execute the code shown in the slide, we might expect that the final value of *x* is 2. However, suppose the two threads execute the machine code at roughly the same time: each loads the value of *x* into its register, each adds one to the contents of the register, and each stores the result into *x*. The final result, of course, is that *x* is 1, not 2.



Quiz 3

Suppose the following code is compiled by gcc. Will it still be the case that x's value might not be incremented by 2?

a) yes
b) no

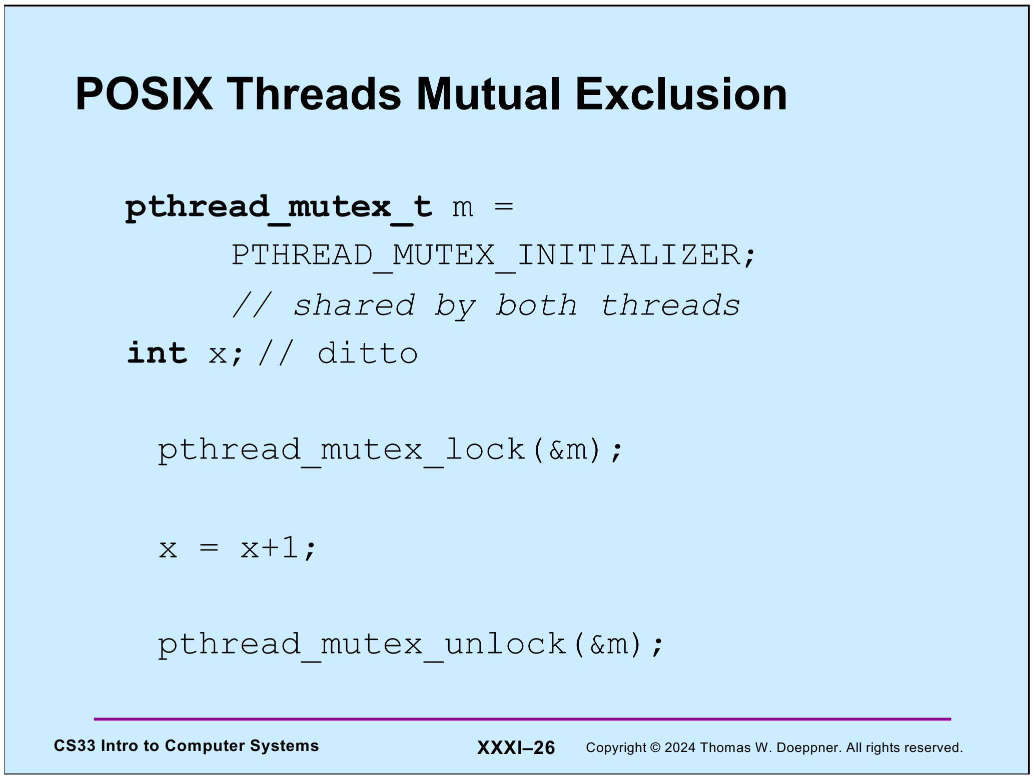
Thread 1:

```
x = x+1;
 /*
   incr x
 */
```

Thread 2:

```
x = x+1;
 /*
   incr x
 */
```

In this example, gcc generates an instruction that directly increments the memory location holding x.



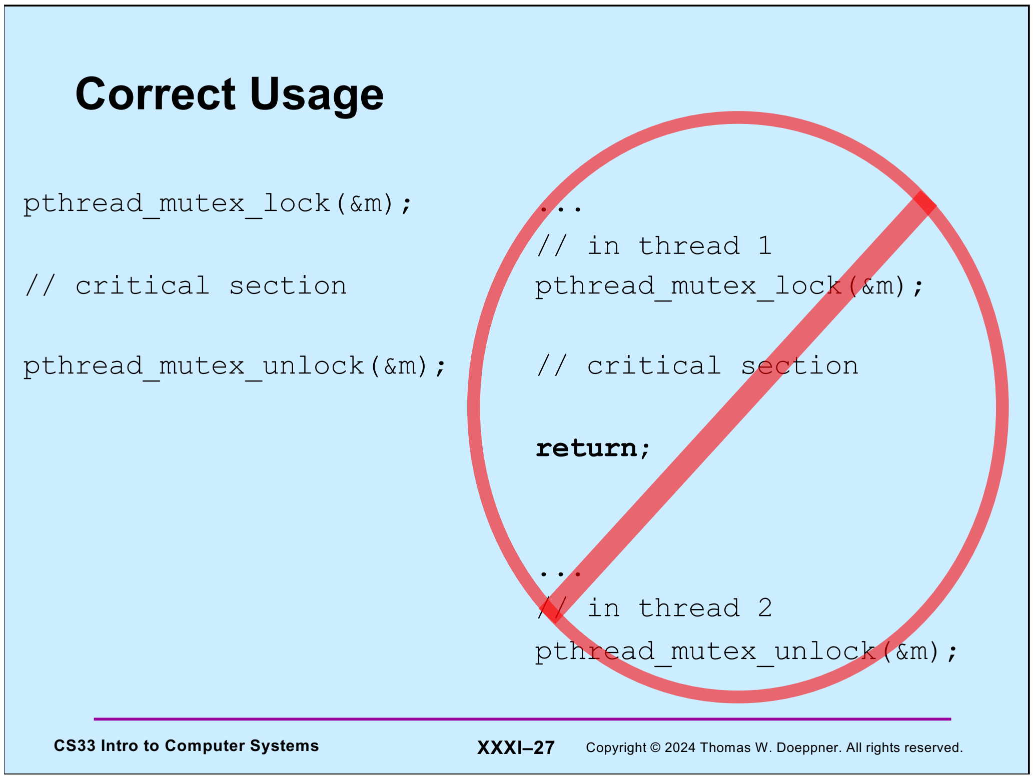
POSIX Threads Mutual Exclusion

```
pthread_mutex_t m =
     PTHREAD_MUTEX_INITIALIZER;
     // shared by both threads
int x; // ditto

  pthread_mutex_lock(&m);

  x = x+1;

  pthread_mutex_unlock(&m);
```

To solve our synchronization problem, we introduce **mutexes** — a synchronization construct providing mutual exclusion. A mutex is used to insure either that only one thread is executing a particular piece of code at once (code locking) or that only one thread is accessing a particular data structure at once (data locking). A mutex belongs either to a particular thread or to no thread (i.e., it is either locked or unlocked). A thread may lock a mutex by calling **pthread_mutex_lock**. If no other thread has the mutex locked, then the calling thread obtains the lock on the mutex and returns. Otherwise, it waits until no other thread has the mutex, and finally returns with the mutex locked. There may of course be multiple threads waiting for the mutex to be unlocked. Only one thread can lock the mutex at a time; there is no specified order for who gets the mutex next, though the ordering is assumed to be at least somewhat "fair."

To unlock a mutex, a thread calls **pthread_mutex_unlock**. It is considered incorrect to unlock a mutex that is not held by the caller (i.e., to unlock someone else's mutex). However, it is somewhat costly to check for this, so most implementations, if they check at all, do so only when certain degrees of debugging are turned on.

Like any other data structure, mutexes must be initialized. This can be done via a call to **pthread_mutex_init** or can be done statically by assigning PTHREAD_MUTEX_INITIALIZER to a mutex. The initial state of such initialized mutexes is unlocked. Of course, a mutex should be initialized only once! (I.e., make certain that, for each mutex, no more than one thread calls **pthread_mutex_init**.)

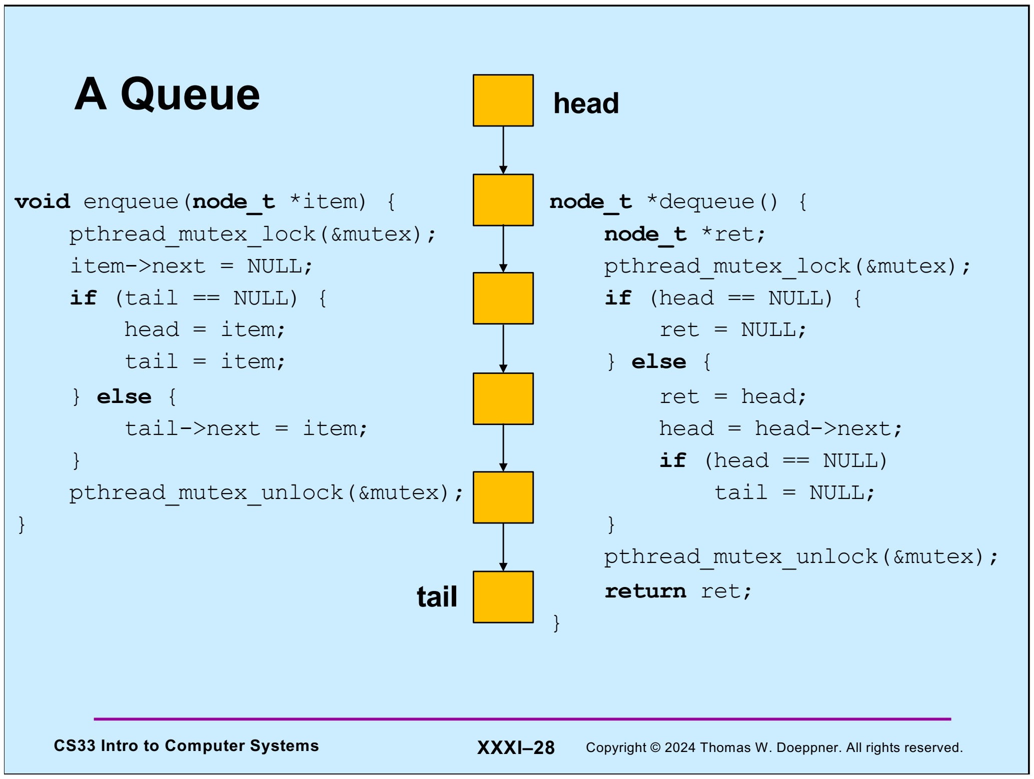


Correct Usage

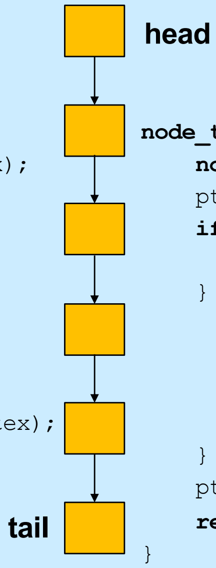
```
pthread_mutex_lock(&m);

// critical section

pthread_mutex_unlock(&m);
```

```
...
// in thread 1
pthread_mutex_lock(&m);

// critical section

return;


...
// in thread 2
pthread_mutex_unlock(&m);
```

An important restriction on the use of mutexes is that the thread that locked a mutex should be the thread that unlocks it. For a number of reasons, not the least of which is readability and correctness, it is not good for a mutex to be locked by one thread and then unlocked by another.



A Queue

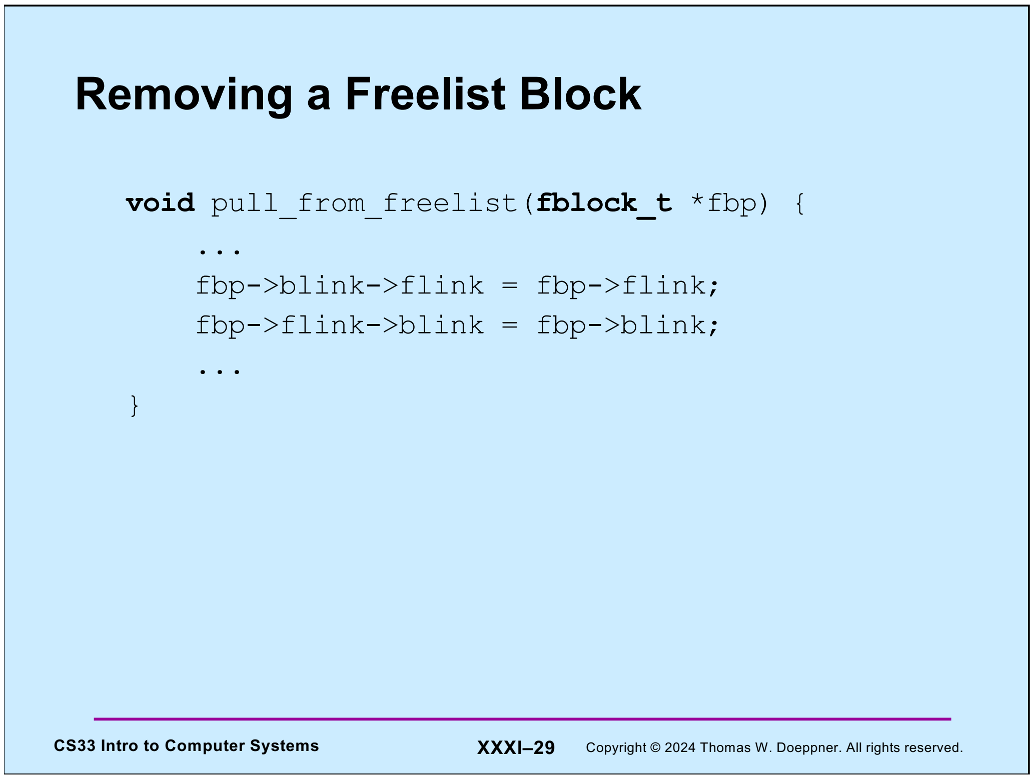


```
void enqueue(node_t *item) {
    pthread_mutex_lock(&mutex);
    item->next = NULL;
    if (tail == NULL) {
        head = item;
        tail = item;
    } else {
        tail->next = item;
    }
    pthread_mutex_unlock(&mutex);
}
```

```
node_t *dequeue() {
    node_t *ret;
    pthread_mutex_lock(&mutex);
    if (head == NULL) {
        ret = NULL;
    } else {
        ret = head;
        head = head->next;
        if (head == NULL)
            tail = NULL;
    }
    pthread_mutex_unlock(&mutex);
    return ret;
}
```

Here we have **enqueue** and **dequeue** functions that can be called by multiple threads to add and remove items from a queue. We employ a single mutex to make certain that at most one thread is performing a queue operation at a time. This could result in a bottleneck if there are lots of threads calling both of the functions. Thus, we might seek a solution that employs separate mutexes for callers to enqueue and for callers to dequeue.

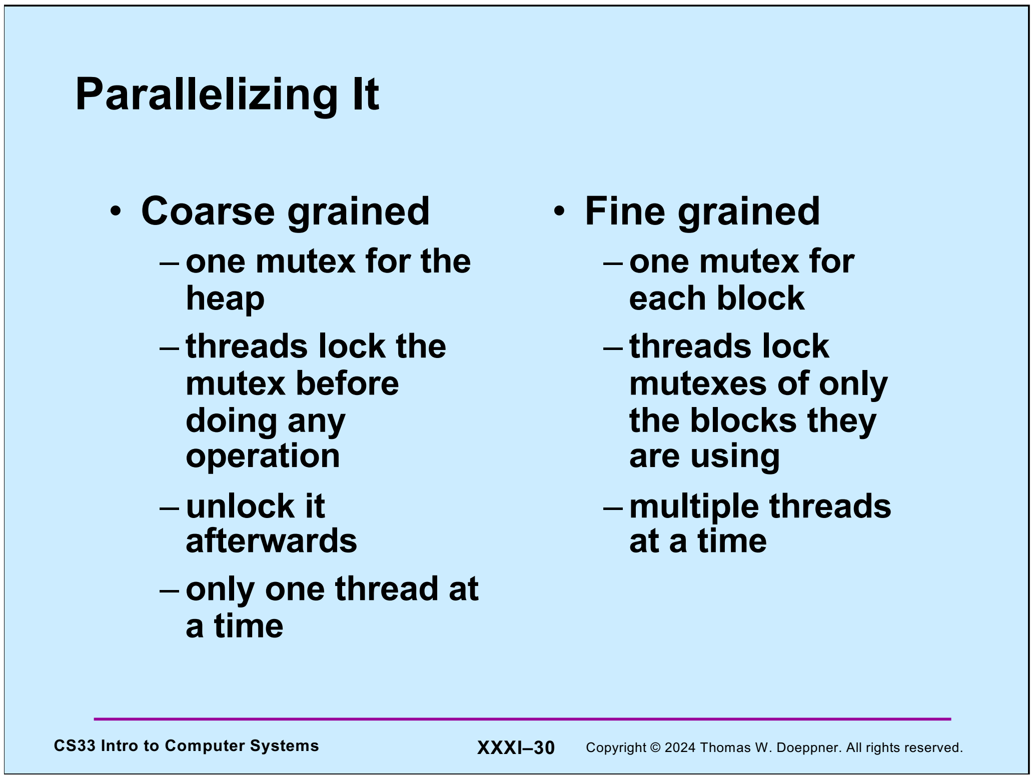
Since enqueue modifies one end of the queue and dequeue modifies the other, one might think that we could have a two-mutex solution, with one mutex protecting one end of the queue and another mutex protecting the other. But this becomes difficult in certain edge conditions. For example. suppose the queue contains just one item. A thread is calling dequeue to remove that item, but at the same time another thread is calling enqueue to add another item. With one mutex protecting the tail of the queue and another protecting the head, we could have a situation in which the next field of the node being enqueued refers to the node being dequeued. If we have a single mutex ensuring mutually exclusive access to the entire queue, this is easy to prevent (the situation can't happen in the code of the slide). But if callers to enqueue use a different mutex than callers to dequeue, we can't easily prevent the problem (If at all).



Removing a Freelist Block

```
void pull_from_freelist(fblock_t *fbp) {
    ...
    fbp->blink->flink = fbp->flink;
    fbp->flink->blink = fbp->blink;
    ...
}
```

We review some code that should be familiar to you from the malloc assignment. We assume, for the sake of the example, that there are at least three blocks in the freelist. (Why it's three and not two comes up later.)



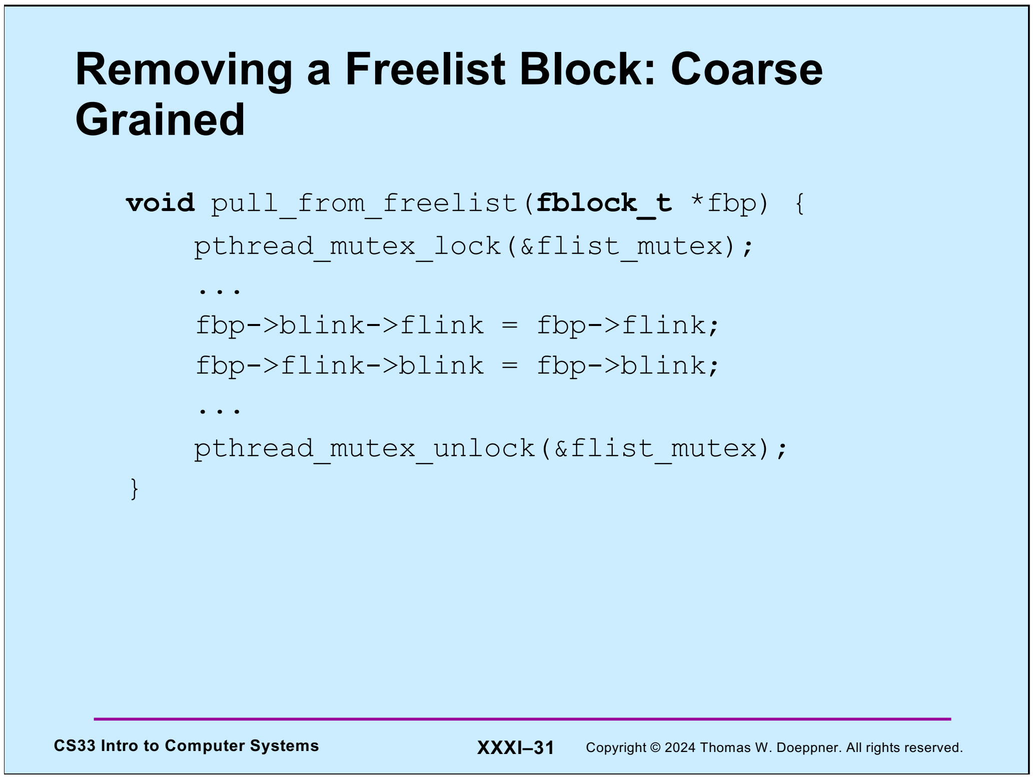
Parallelizing It

- Coarse grained
  - one mutex for the heap
  - threads lock the mutex before doing any operation
  - unlock it afterwards
  - only one thread at a time
- Fine grained
  - one mutex for each block
  - threads lock mutexes of only the blocks they are using
  - multiple threads at a time

However, we'd now like to modify all the malloc code so that multiple threads can use it at the same time.

One way of doing this, which is pretty straightforward, is to employ a single mutex, which threads must lock before doing any heap operation, and then unlock when they're finished. However, this approach allows only one thread to be manipulating the heap at a time. This approach to using mutexes is called **coarse-grained**.

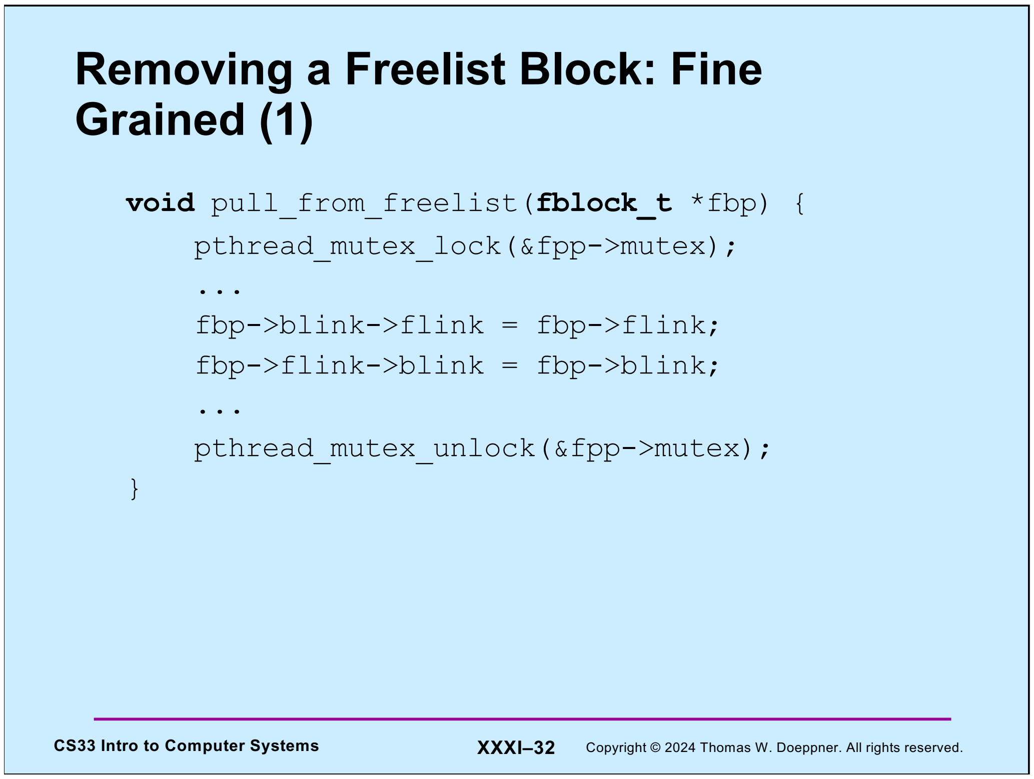
To get more advantage (in terms of speed) from using multiple threads, we'd like it to be possible for multiple threads to be manipulating the heap at once. To accomplish this, each block has a mutex inside of it. To do anything to this block (such as pull it from the free list or add it to the free list), the thread doing so must lock the block's mutex, then, of course, unlock it when done. This approach to using mutexes is called **fine-grained**.



Removing a Freelist Block: Coarse Grained

```
void pull_from_freelist(fblock_t *fbp) {
    pthread_mutex_lock(&flist_mutex);
    ...
    fbp->blink->flink = fbp->flink;
    fbp->flink->blink = fbp->blink;
    ...
    pthread_mutex_unlock(&flist_mutex);
}
```

Here’s the modified code for pull_from_freelist, using coarse-grained synchronization.

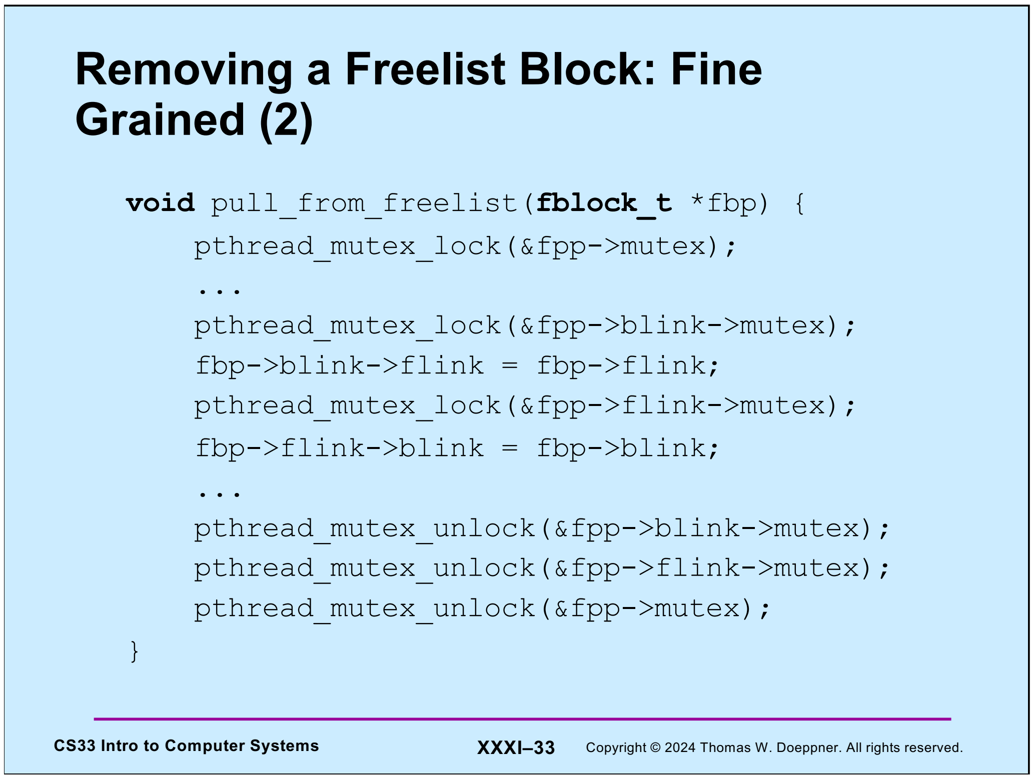


Removing a Freelist Block: Fine Grained (1)

```
void pull_from_freelist(fblock_t *fbp) {
    pthread_mutex_lock(&fpp->mutex);
    ...
    fbp->blink->flink = fbp->flink;
    fbp->flink->blink = fbp->blink;
    ...
    pthread_mutex_unlock(&fpp->mutex);
}
```

Let’s try a similar approach for the fine-grained approach.

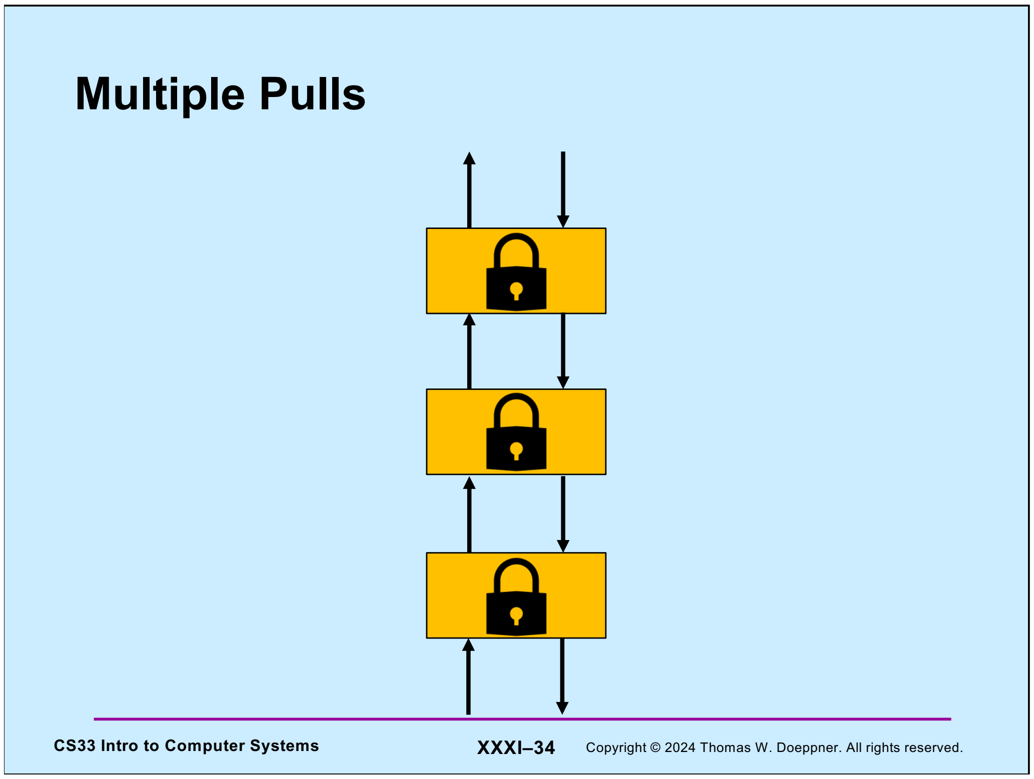
However, we can see an immediate problem: the thread executing this code will not only modify *fbp, but also its predecessor and successor blocks in the freelist.



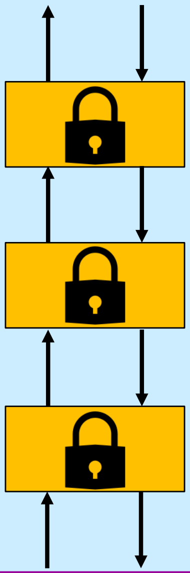
Removing a Freelist Block: Fine Grained (2)

```
void pull_from_freelist(fblock_t *fbp) {
    pthread_mutex_lock(&fpp->mutex);
    ...
    pthread_mutex_lock(&fpp->blink->mutex);
    fbp->blink->flink = fbp->flink;
    pthread_mutex_lock(&fpp->flink->mutex);
    fbp->flink->blink = fbp->blink;
    ...
    pthread_mutex_unlock(&fpp->blink->mutex);
    pthread_mutex_unlock(&fpp->flink->mutex);
    pthread_mutex_unlock(&fpp->mutex);
}
```

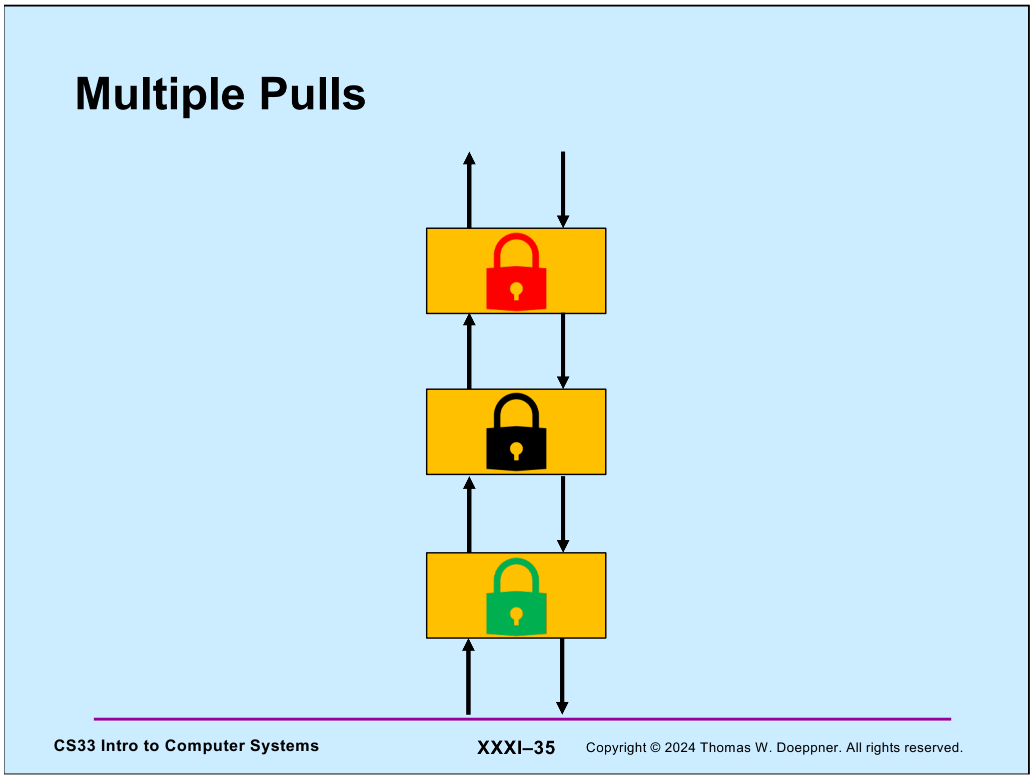
So, we add code to lock and unlock the mutexes of the adjacent blocks.



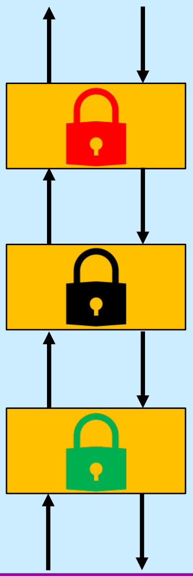
Multiple Pulls



Pictorially, our thread first locks the mutex on *fbp (the middle block), then its predecessor and then its successor.



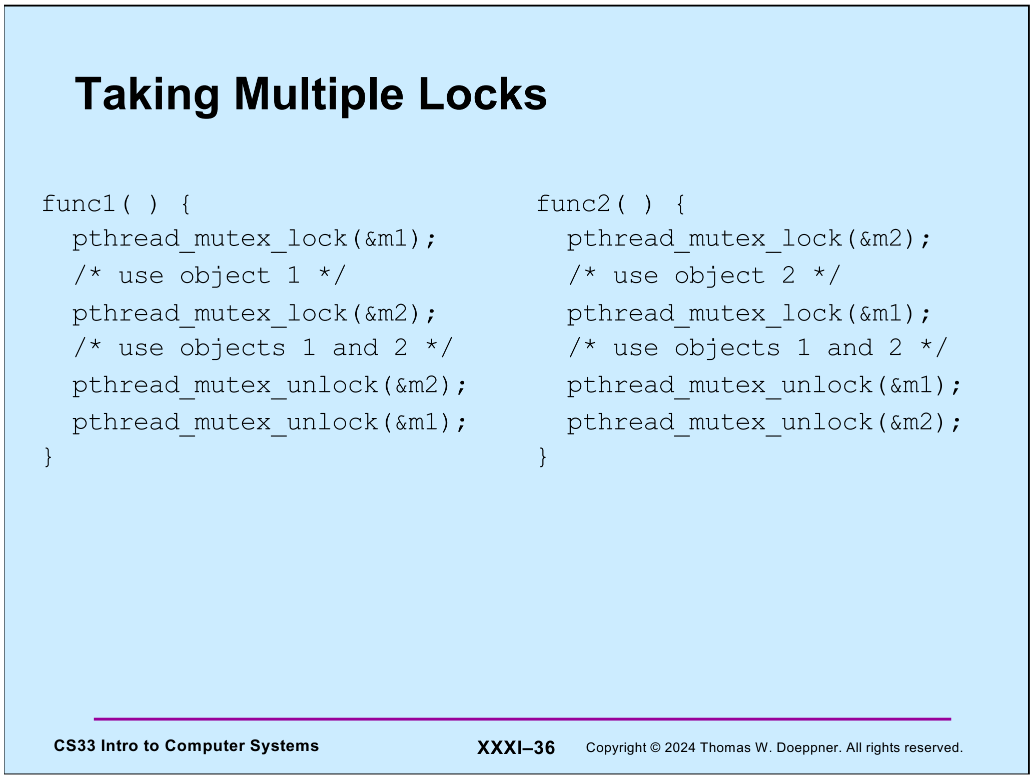
Multiple Pulls



But suppose other threads are calling pull_from_freelist at the same time. Let’s say that thread 1 is pulling the middle block, thread 2 is pulling the upper block, and thread 3 is pulling the lower block.

All three threads have locked the mutex on the block they’re pulling. Thread one tries to lock the mutex on the upper block (once it successfully locks it, then it will try to lock the mutex on the lower block). However, thread 2 has already locked the mutex on the upper block, and won’t unlock it until after it locks the mutex on the middle block. But thread 1 won’t unlock it until it locks the mutex on the upper block (not to mention the lower block).

So, we’re stuck. threads 1 and 2 (as well as 3) won’t ever be able to lock all the mutexes they need, and, because of this, can’t make any further progress. This phenomenon is called **deadlock**.



Taking Multiple Locks

```
func1( ) {                           func2( ) {
  pthread_mutex_lock(&m1);             pthread_mutex_lock(&m2);
  /* use object 1 */                   /* use object 2 */
  pthread_mutex_lock(&m2);             pthread_mutex_lock(&m1);
  /* use objects 1 and 2 */            /* use objects 1 and 2 */
  pthread_mutex_unlock(&m2);           pthread_mutex_unlock(&m1);
  pthread_mutex_unlock(&m1);           pthread_mutex_unlock(&m2);
}                                    }
```

In this example our threads are using two mutexes to control access to two different objects. Thread 1, executing **func1**, first takes mutex 1, then, while still holding mutex 1, obtains mutex 2. Thread 2, executing **func2**, first takes mutex 2, then, while still holding mutex 2, obtains mutex 1. However, things do not always work out as planned. If thread 1 obtains mutex 1 and, at about the same time, thread 2 obtains mutex 2, then if thread 1 attempts to take mutex 2 and thread 2 attempts to take mutex 1, we have a **deadlock**.