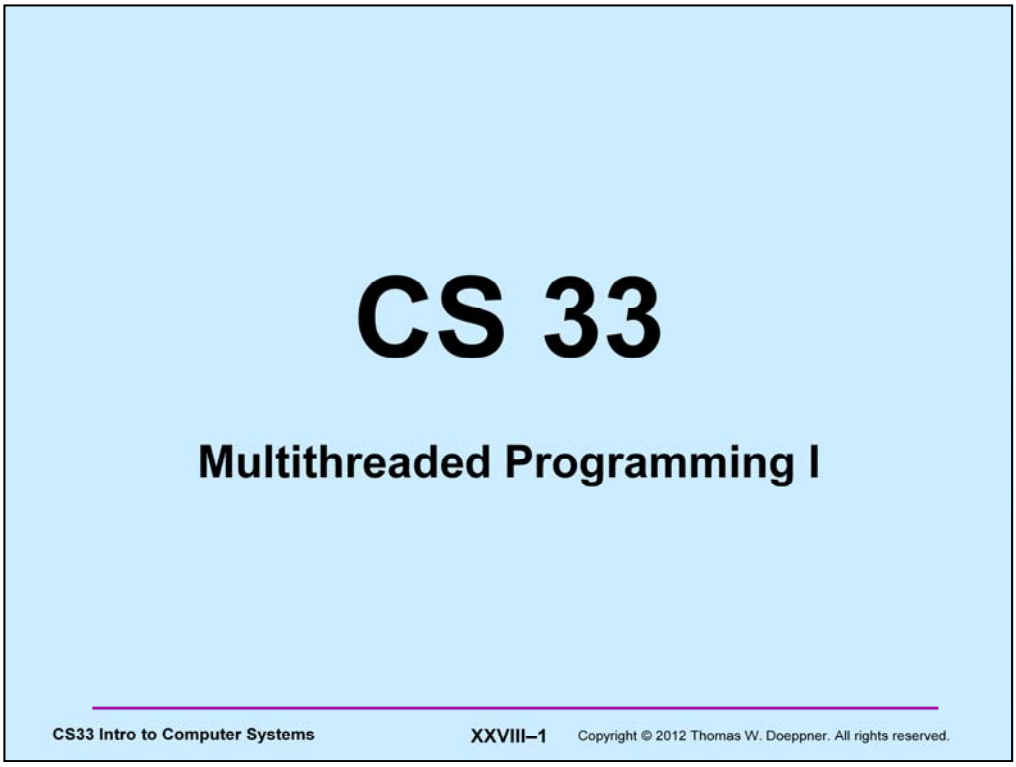
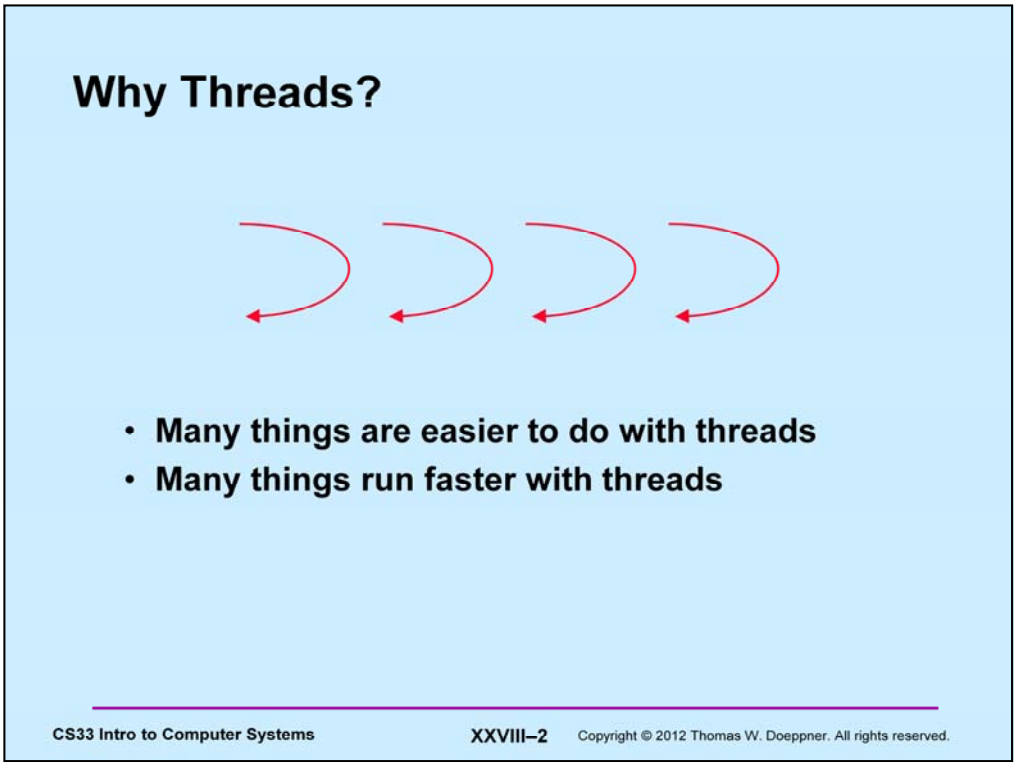
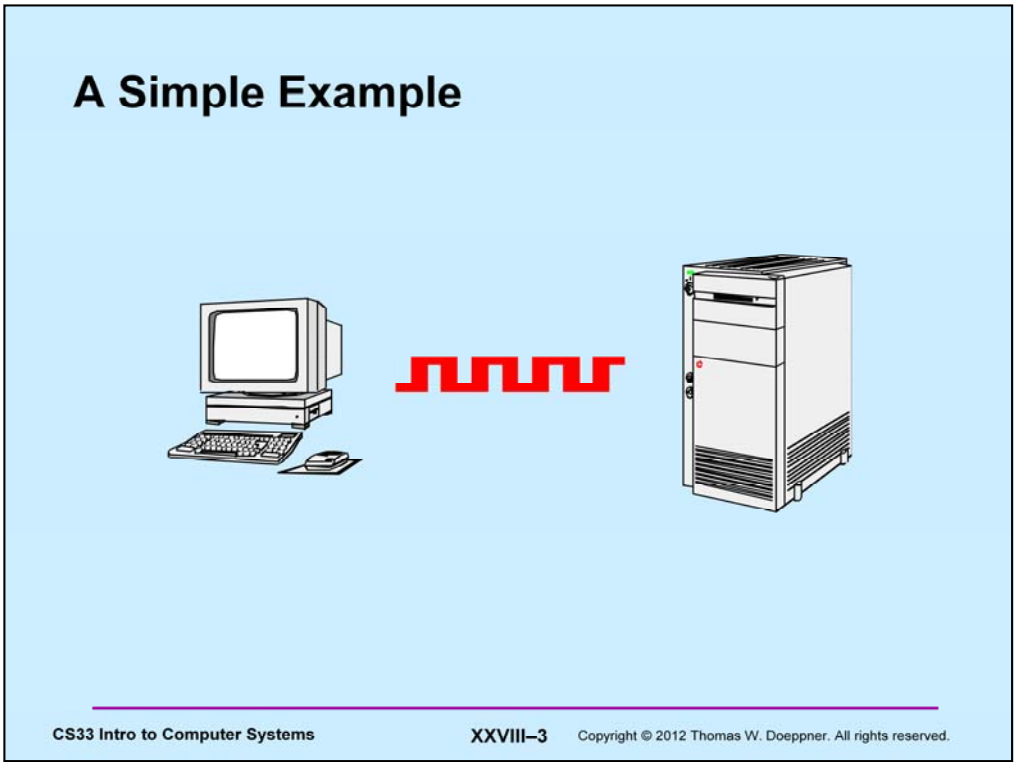


Source code used in this lecture is available on the course web page.

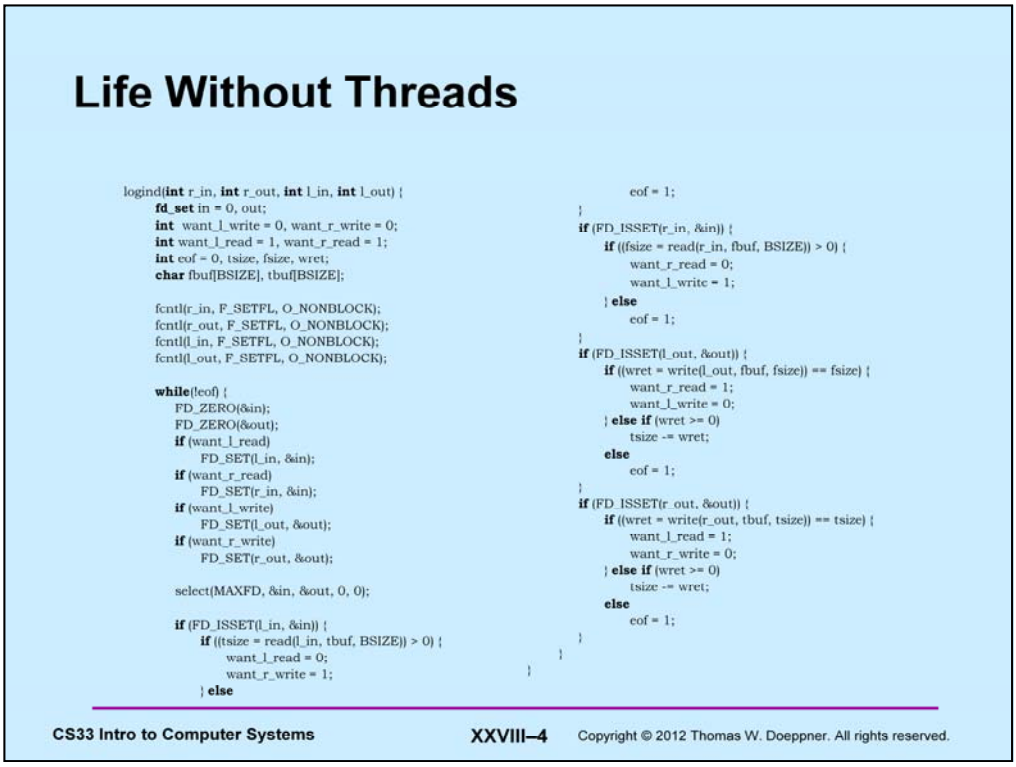


A *thread* is the abstraction of a processor — it is a *thread of control*. We are accustomed to writing single-threaded programs and to having multiple single-threaded programs running on our computers. Why does one want multiple threads running in the same program? Putting it only somewhat over-dramatically, programming with multiple threads is a powerful paradigm.

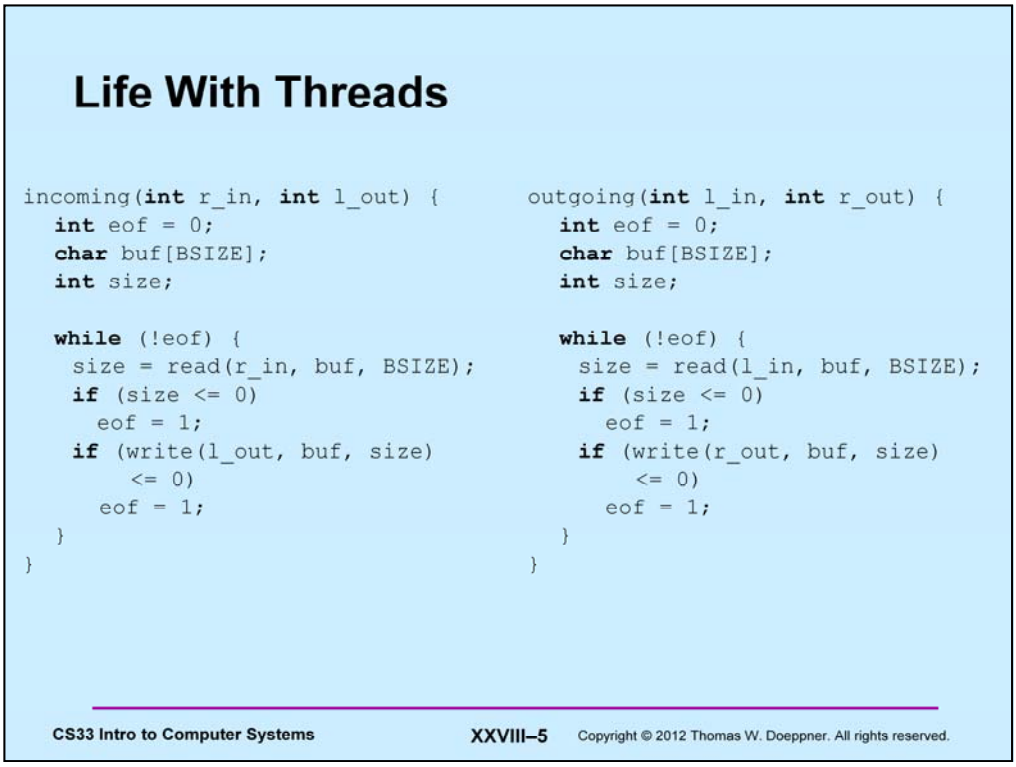
So, what is so special about this paradigm? Programming with threads is a natural means for dealing with *concurrency*. As we will see, concurrency comes up in numerous situations. A common misconception is that it is a useful concept only on multiprocessors. Threads do allow us to exploit the features of a multiprocessor, but they are equally useful on uniprocessors — in many instances a multithreaded solution to a problem is simpler to write, simpler to understand, and simpler to debug than a single-threaded solution to the same problem.



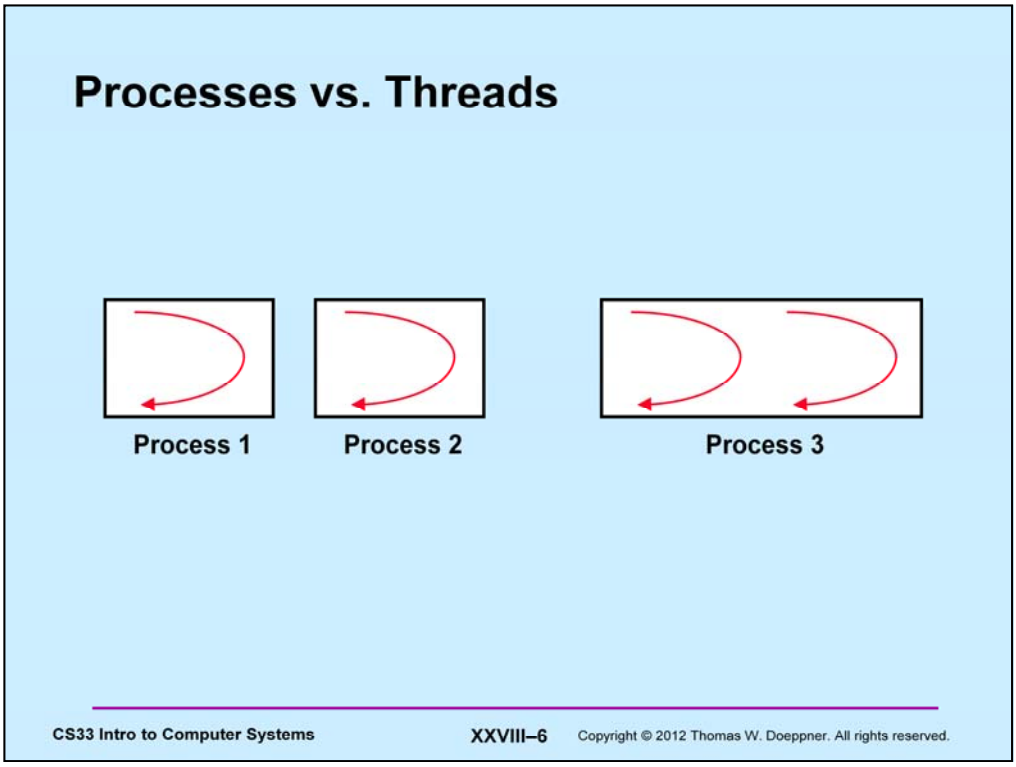
For a simple example of a problem that is more easily solved with threads than without, let's look at a server application, in particular, a remote login server. The service discussed here, rlogin, has since been replaced with ssh, but what ssh does includes what rlogin does. (ssh is much more secure, among other features.)



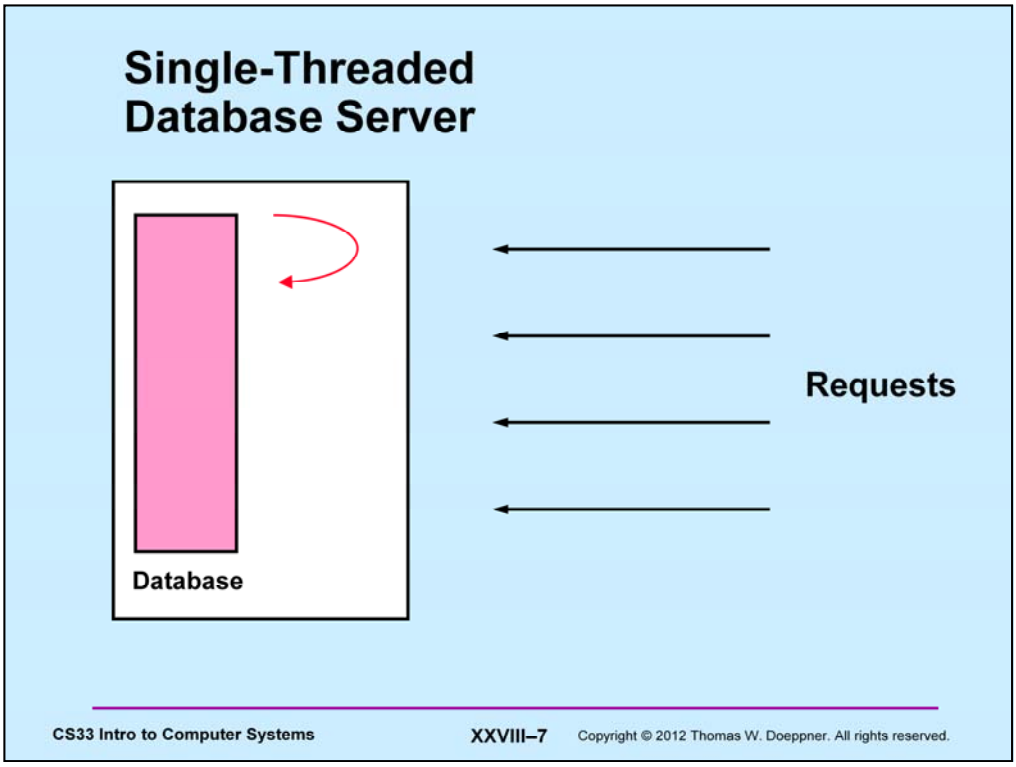
This slide shows a simplified excerpt from the *rlogind* program of Unix, the server portion of the rlogin service. It is not immediately apparent what it does (that's the point): it reads characters typed by the remote user (arriving on *r_in*), outputs these characters to local applications (via *l_out*), reads characters output by local applications (arriving on *l_in*), and sends them to the remote user (via *r_out*). To make sure that we never block indefinitely waiting for I/O on any particular one of the four file descriptors, we make complicated use of non-blocking I/O and the *select* system call.



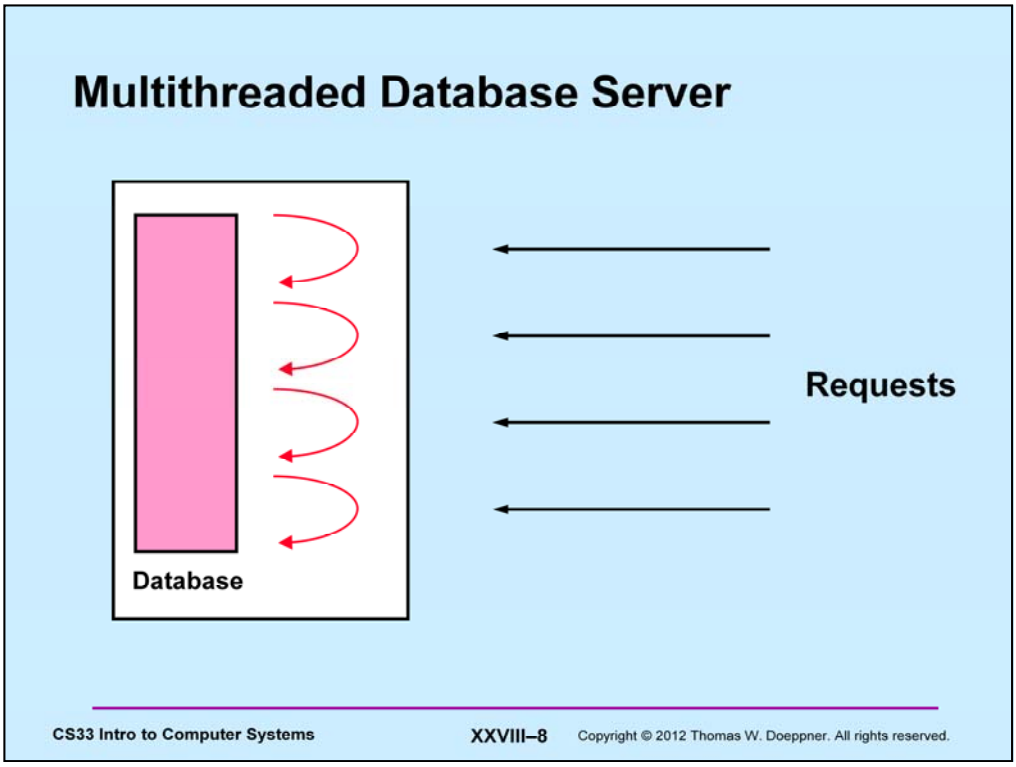
Here is a two-threaded implementation of *rlogind*. One thread simply reads from *r_in* and writes to *l_out*, while the other reads from *l_in* and writes to *r_out*. To an external observer, this solution and the previous one are equivalent, but with this implementation it is apparent how simple-minded the solution is. (In fact, were it not for other obscure things done in *rlogind*, what is shown here could just as well be written as two single-threaded processes as one two-threaded process.)



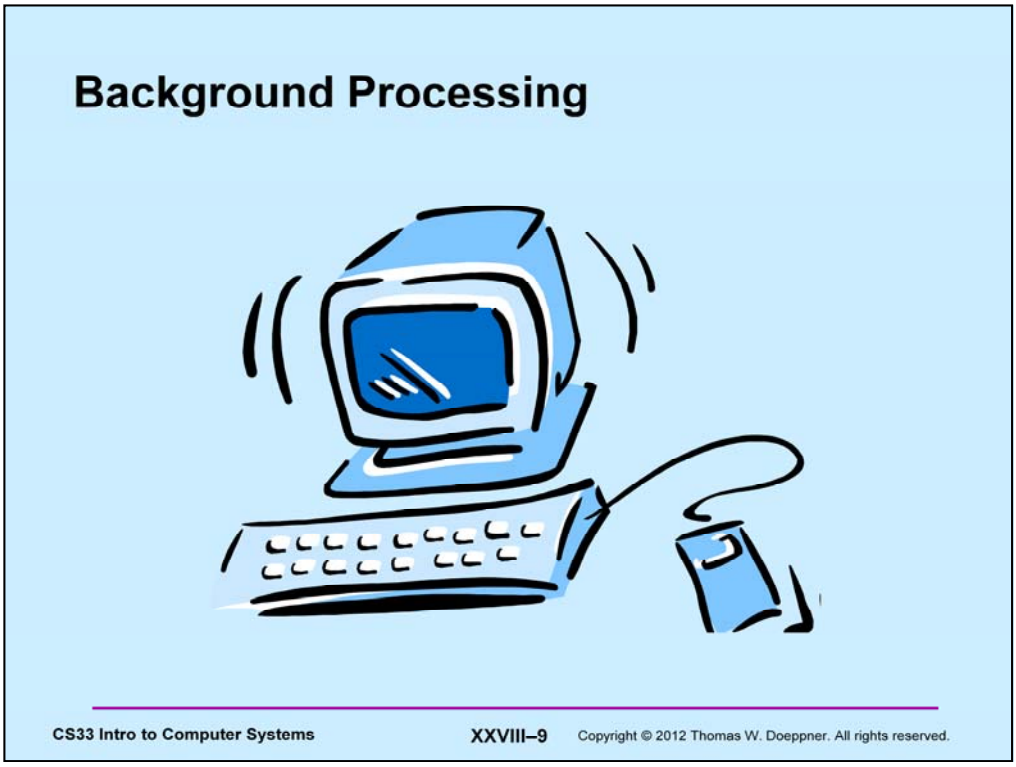
Speaking of processes, what is the difference between two single-threaded processes and one two-threaded process? First of all, if one process already exists, it is much cheaper to create another thread in the existing process than to create a new process. Switching between the contexts of two threads in the same process is also often cheaper than switching between the contexts of two threads in different processes. Finally, two threads in one process share everything—both address space and open files; the two can communicate without having to copy data. Though two different processes can share memory in modern Unix systems, the most common forms of interprocess communication are far more expensive.



Here is another server example, a database server handling multiple clients. The single-threaded approach to dealing with these requests is to handle them sequentially or to multiplex them explicitly. The former approach would be unfair to quick requests occurring behind lengthy requests, and the latter would require fairly complex and error-prone code.

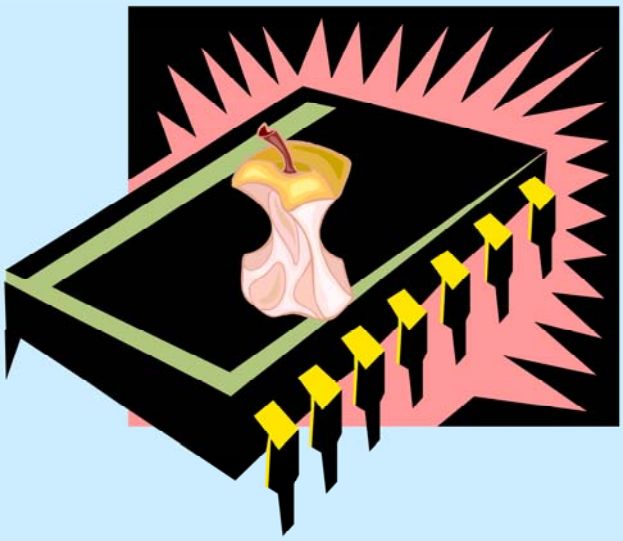


We now rearchitect our server to be multithreaded, assigning a separate thread to each request. The code is as simple as in the sequential approach and as fair as in the multiplexed approach. Some synchronization of access to the database is required, a topic we will discuss soon.

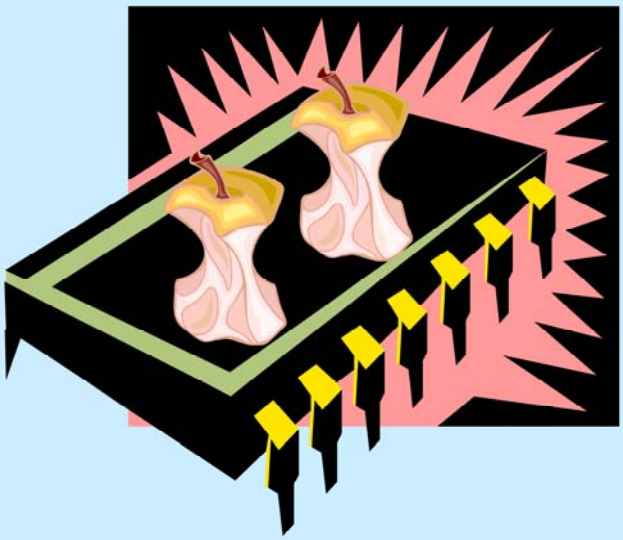


Another situation in which multithreading is a good paradigm is handling background activities. Consider writing a GUI-based application that uses both mouse and keyboard input and handles various clock- and counter-based events such as those resulting in the checkpointing of the application. In many of today's applications, if the application is busy with one activity, such as checkpointing, it cannot respond (quickly enough) to other events, such as mouse or keyboard input. Handling such concurrency is difficult and complex within a single-threaded process, but simple in a multithreaded process.

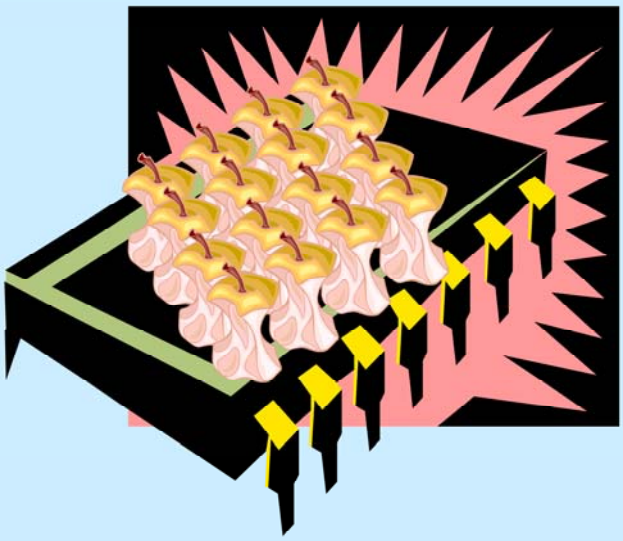
Single-Core Chips



Dual-Core Chips

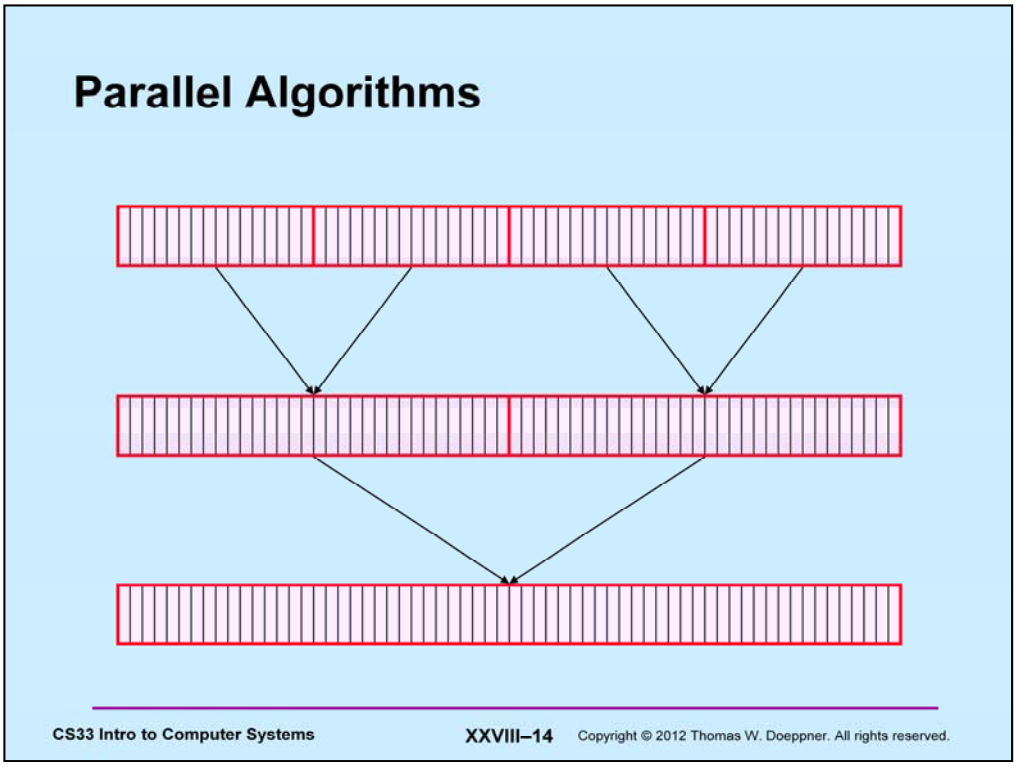


Multi-Core Chips

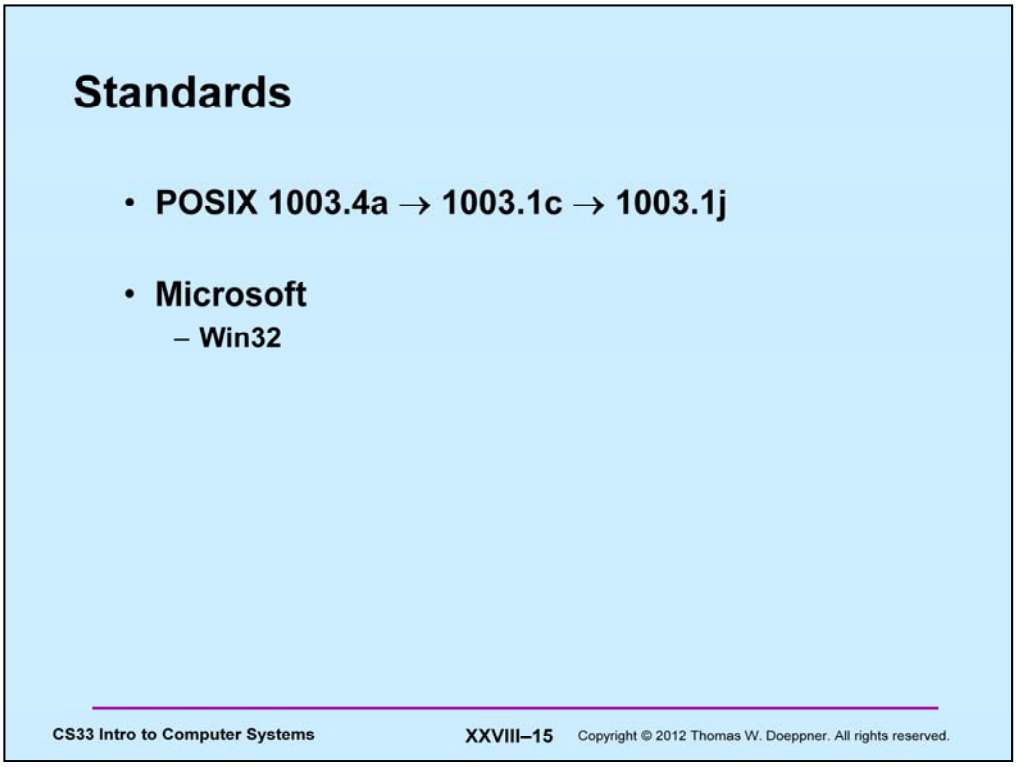


Good News/Bad News

- ☺ Good news
  - multi-threaded programs can take advantage of multi-core chips (single-threaded programs cannot)
- ☹ Bad news
  - it's not easy
    - » must have parallel algorithm
      - employing at least as many threads as processors
      - threads must keep processors busy
        - doing useful work



Now suppose we want to sort some data items on a shared-memory parallel computer. We can match our use of the thread paradigm to the architecture by creating one thread per processor (four in this example). We rely on the operating system to schedule threads to processors (more on this later); our task is merely to implement a multithreaded sorting algorithm. The picture illustrates the application of *mergesort*, which is particularly easy to parallelize with such small numbers of processors. We divide the data into four pieces and have each processor sort a different piece. Then two processors each merge two pieces and one processor merges the final two combined pieces. On one processor, this requires on the order of $n \cdot \log n + n$ comparisons, while on four processors it requires on the order of $(n/4) \cdot \log n + 3n/4$ comparisons. (A speedup proportional to the number of processors is not possible with this form of sorting.)



Despite the advantages of programming with threads, only recently have standard APIs for multithreaded programming been developed. The most important of these APIs, at least in the Unix world, is the one developed by the group known as POSIX 1003.4a. This effort took a number of years and in the summer of '95 resulted in an approved standard, which is now known by the number 1003.1c. In 2000, the POSIX advanced realtime standard, 1003.1j, was approved. It contains a number of new features added to POSIX threads.

Microsoft, characteristically, has produced a threads package whose interface has little in common with those of the Unix world. Moreover, there are significant differences between the Microsoft and POSIX approaches—some of the constructs of one cannot be easily implemented in terms of the constructs of the other, and vice versa. Despite this, both approaches are equally useful for multithreaded programming.

Creating a Thread

```
start_servers( ) {
  pthread_t thread;
  for (int i=0; i<nr_of_server_threads; i++)
    pthread_create(&thread,   // thread ID
      0,                      // default attributes
      server,                 // start routine
      argument);              // argument
  ...
}

void *server(void *arg) {
  while(1) {
    /* get and handle request */
  }
  return(0);
}
```


To create a thread, one calls the *pthread_create* routine. 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 routine in which our new thread should start its execution. The last argument is the argument that is actually passed to the first procedure of the thread.

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

Complications

```
rlogind(int r_in, int r_out, int l_in, int l_out) {
  pthread_t in_thread, out_thread;

  pthread_create(&in_thread,
      0,
      incoming,
      r_in, l_out);          // Can't do this ...
  pthread_create(&out_thread,
      0,
      outgoing,
      l_in, r_out);          // Can't do this  ...

  /* How do we wait till they're done? */
}
```


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 *rlogind* example, but we run into a problem when we try to pass two parameters to each of the two threads.

A further issue is synchronization with the termination of a thread. For a number of reasons we'll find it necessary for a thread to be able to suspend its execution until another thread has terminated.

Multiple Arguments

```
typedef struct {
  int first, second;
} two_ints_t;

rlogind(int r_in, int r_out, int l_in, int l_out) {
  pthread_t in_thread, out_thread;
  two_ints_t in={r_in, l_out}, out={l_in, r_out};
  pthread_create(&in_thread,
      0,
      incoming,
      &in);
  ...

}
```

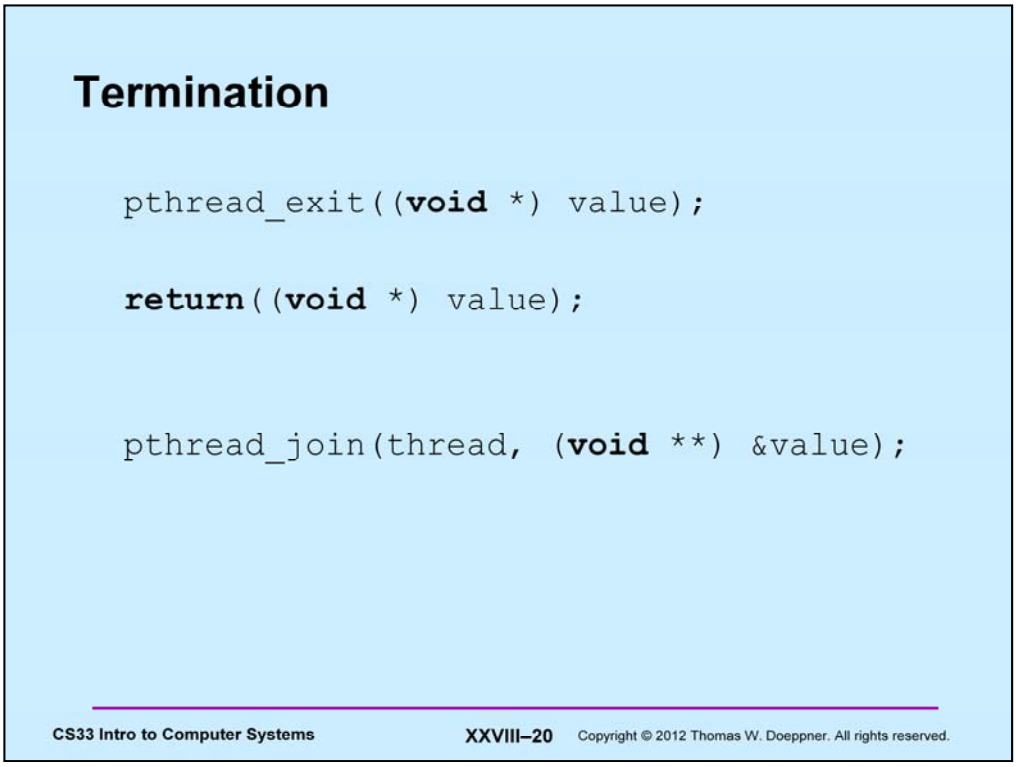

To pass more than one argument to the first procedure of a thread, we must somehow encode multiple arguments as one. Here we pack two arguments into a structure, then pass the pointer to the structure. This technique at least does not produce any compile-time errors, but it has a potential serious problem.

When Is It Done?

```
rlogind(int r_in, int r_out, int l_in, int l_out) {
  pthread_t in_thread, out_thread;
  two_ints_t in={r_in, l_out}, out={l_in, r_out};

  pthread_create(&in_thread, 0, incoming, &in);
  pthread_create(&out_thread, 0, outgoing, &out);

  pthread_join(in_thread, 0);
  pthread_join(out_thread, 0);
}
```


In the previous example, the *in* and *out* structures are local variables of the "mainline" thread. Their addresses are passed to new threads. If the mainline thread returns from its *rlogind* procedure before the new threads terminate, there is the danger that the new threads may dereference their argument pointers into storage locations that are no longer active. This would cause a serious runtime problem that might be difficult to track down.

However, if we can guarantee that the mainline thread does not return from *rlogind* until after the new threads have terminated, then our means for passing multiple arguments is safe. In the example above, the mainline thread places calls to *pthread_join*, which does not return until the thread mentioned as its first argument has terminated. Thus the mainline thread waits until the new threads terminate and then returns from *rlogind*.



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).

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);
  }
  ...
}

server( ) {
  ...
}
```


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 with, but they cannot be joined with.

Types

```
pthread_create(&tid, 0, (void *(*)(void *))func, (void *)1);
  …

int func = 4;              // func definition 1

void func(int i) {         // func definition 2
  …
}

void *func(void *arg) {    // func definition 3
  int i = (int)arg;
  …
  return 0;
}
```


This slide shows some problems that can occur because of confusion about types. Suppose *pthread_create* is called, as shown in the slide. The third argument, *func*, has been carefully cast so that the compiler will not issue warning messages. However, if the definition of *func* is the first one, there will be a serious problem when the program is run! The issue here is that the C compiler trusts you when you give it a cast — it's up to you to make certain that the trust is warranted.

Suppose the second definition of *func* is used. At least this one is a function. However, there are two potential problems. The given definition returns nothing, though *pthread_create* assumes it returns a *void **. On most, if not all of today's systems, this probably is not a problem, though it is conceivable that the calling sequence for a function that returns an argument is different from that for a function that doesn't (e.g., space for the return value might be allocated on the stack).

The second problem can definitely occur on some of today's architectures. The argument being passed to *func* is supplied as an *int*, though it will be passed as a *void **. The routine *func*, however, expects an *int*, though receives a *void **. If there is no difference in size between an *int* and a *void **, this is not likely to be a problem. However, consider a 64-bit machine on which a *void ** occupies 64 bits and an *int* occupies 32 bits. In our example, the 32-bit *int* would be passed within a 64-bit *void **, probably as the least-significant bits; *func* would receive a 64-bit quantity, but would use only 32 bits of it. Which 32 bits? On little-endian architectures, *func* would use the least significant 32 bits and things would work, but there would be problems on big-endian architectures.

The correct way of doing things is shown in the third definition, in which *func* does return something and there is an explicit (and legal) conversion from *void ** to *int*.

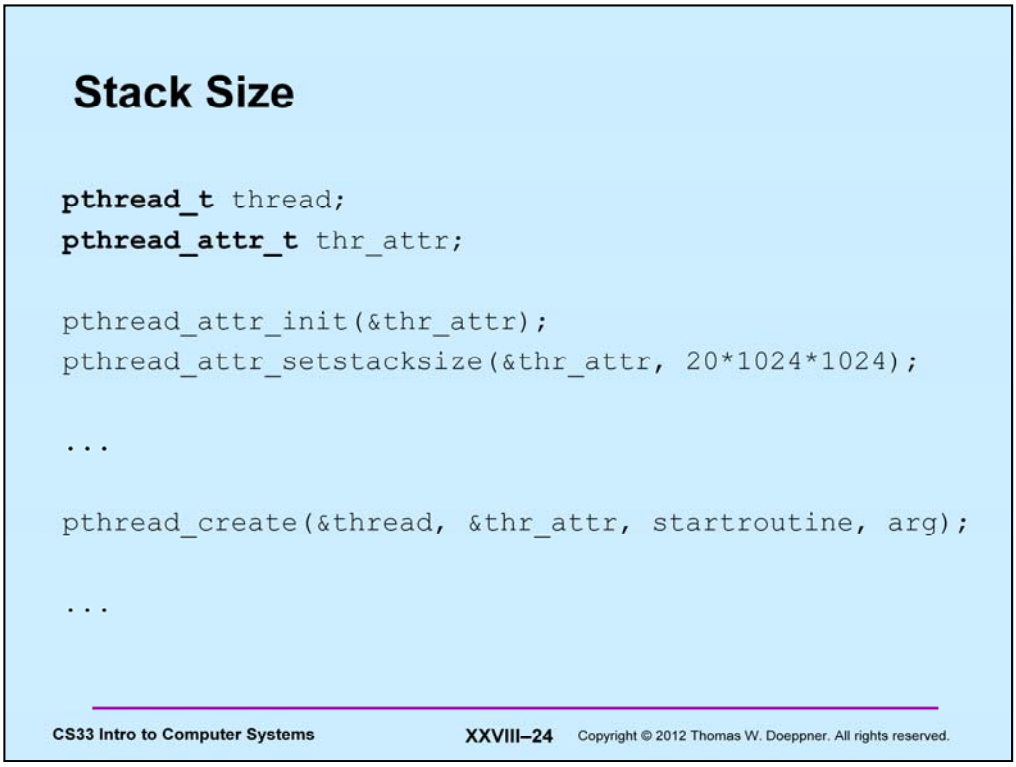
Thread Attributes

```
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*.



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* routine, 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).

Example (1)

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

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

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

void *matmult(void *);
```

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

  // initialize the matrices
  ...
```


In this series of slides we present a simple example of an (almost) complete program—one that multiplies two matrices using a number of threads. Our algorithm is not an example of an efficient matrix-multiplication algorithm, but it shows us everything that must be included to make a multithreaded C program compile and run. Our approach is to use the most straightforward technique for multiplying two matrices: each element of the product is formed by directly taking the inner product of a row of the multiplier and a column of the multiplicand. We employ multiple threads by assigning each row of the product a thread to compute it.

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

Example (2)

```
for (i=0; i<M; i++) {   /* create the 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 routines, 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). However, the text associated with error codes is matched with error codes, just as for Unix-system-call error codes.

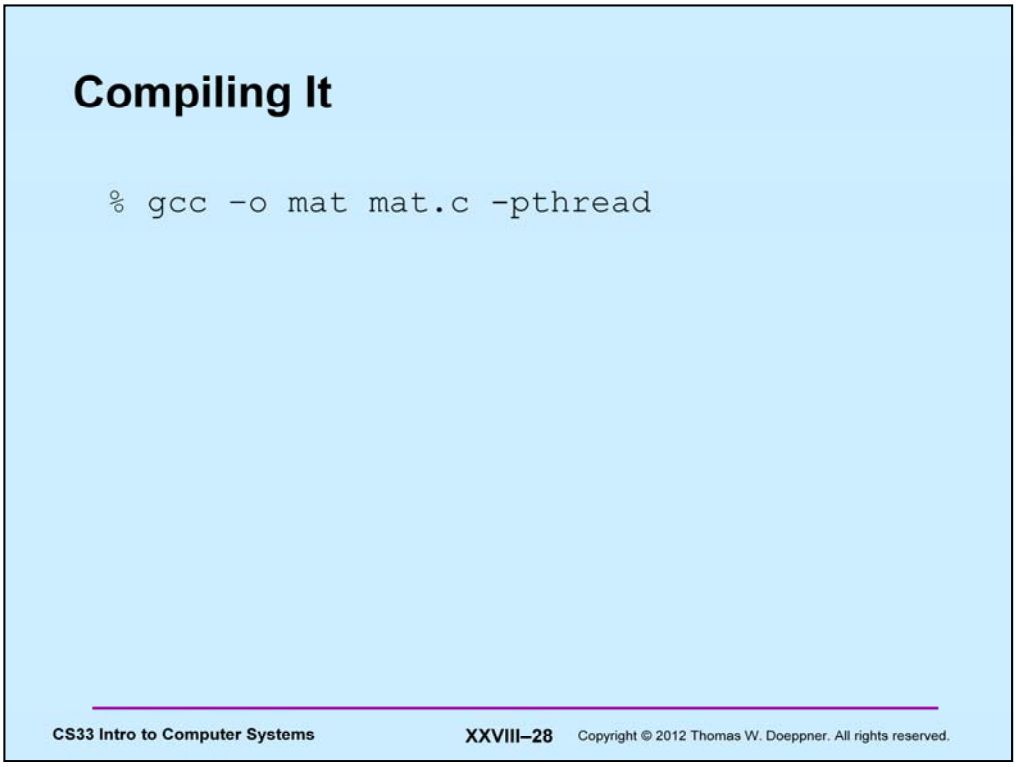
So that the first thread is certain that all the other threads have terminated, it must call *pthread_join* on each of them.

Example (3)

```
void *matmult(void *arg) {
  int row = (int)arg;
  int col;
  int i;
  int 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);
}
```

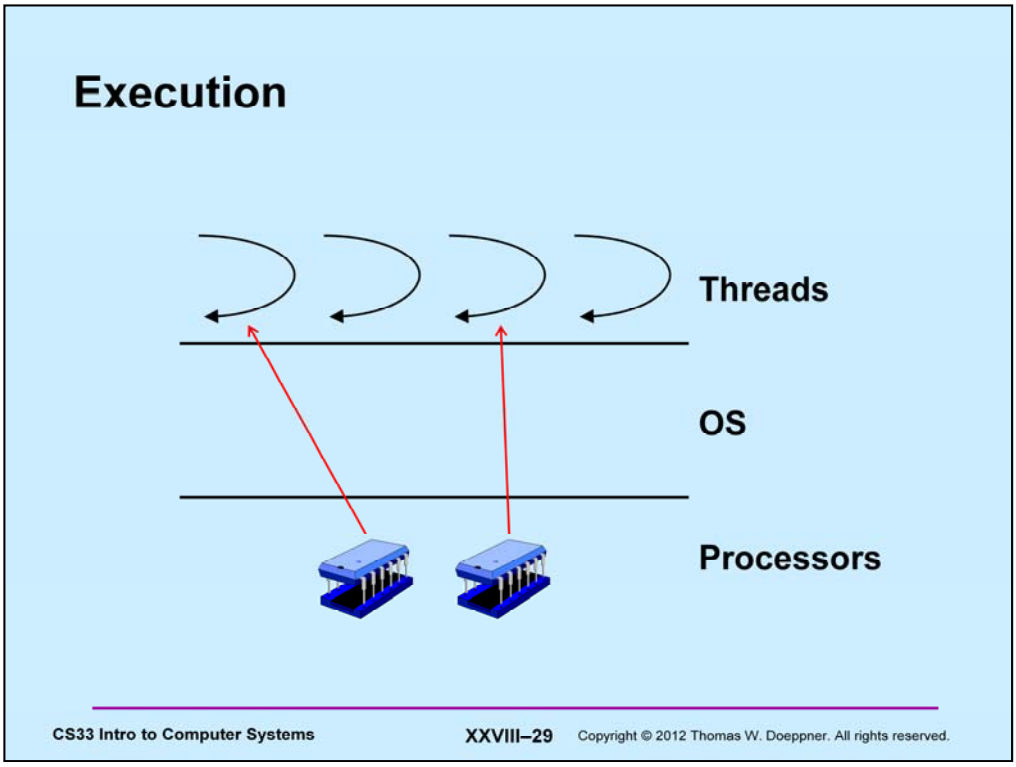

Here is the code executed by each of the threads. It's pretty straightforward: it merely computes a row of the result matrix.

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

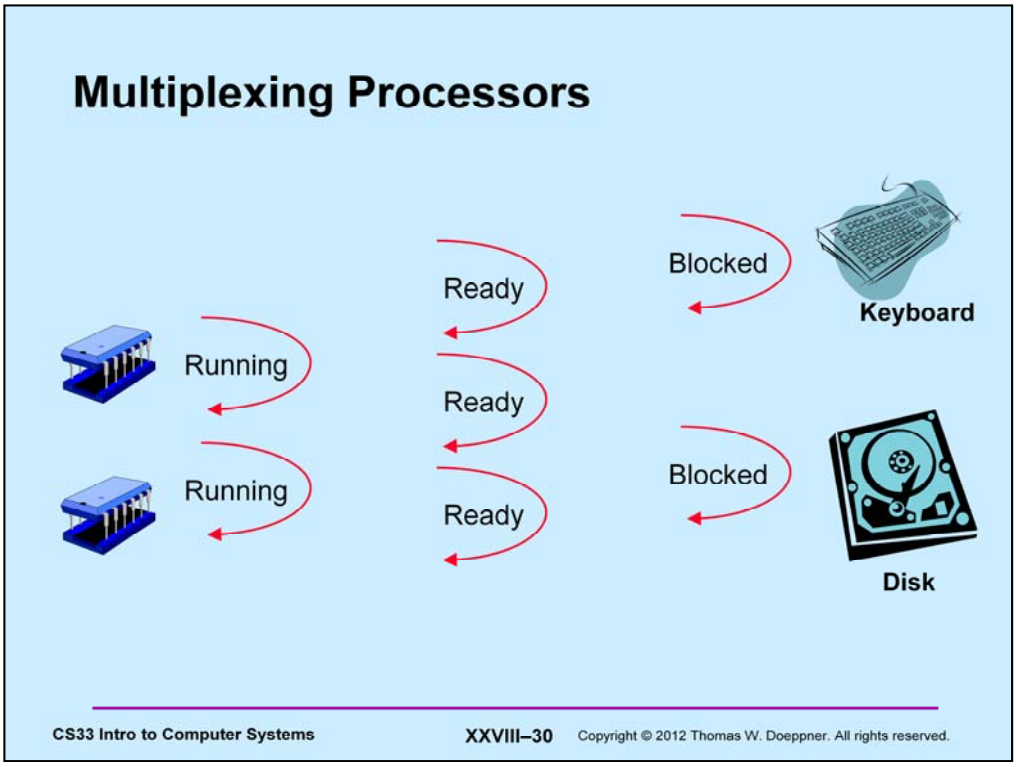


The –pthread flag to gcc is equivalent providing the following flags:

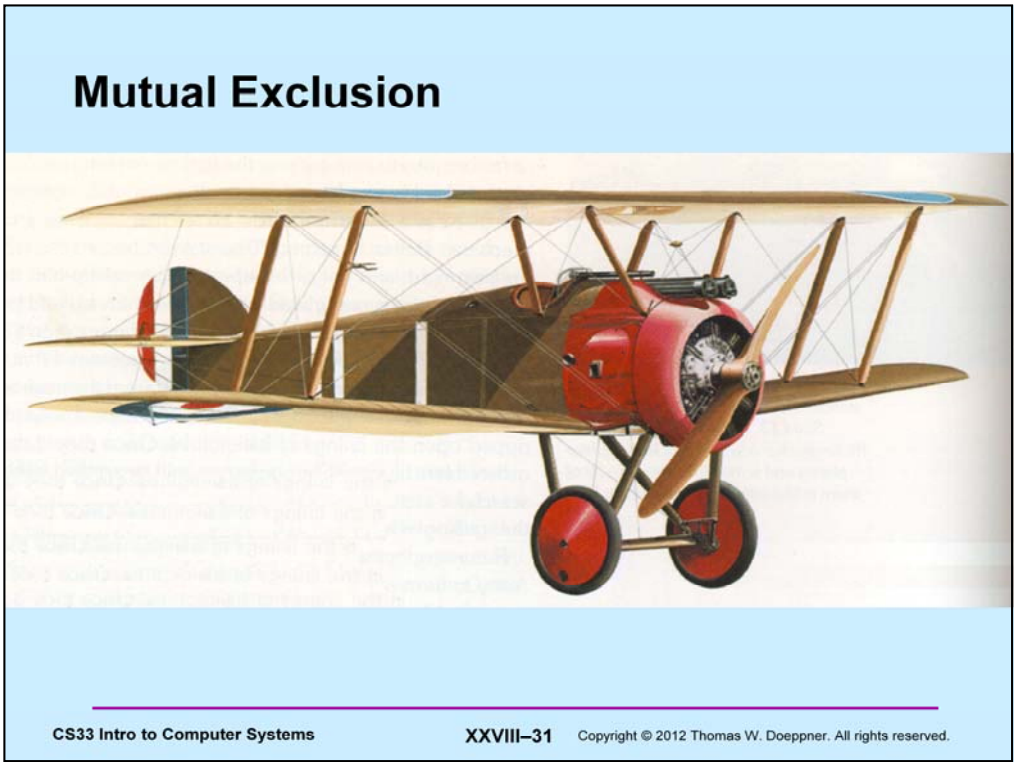
- -lpthread: include libpthread.so — the POSIX threads library
- -D_REENTRANT: defines certain things relevant to threads in stdio.h — we cover this later.
- -Dotherstuff, where “otherstuff” is a variety of flags required to get the current versions of declarations for POSIX threads in pthread.h.



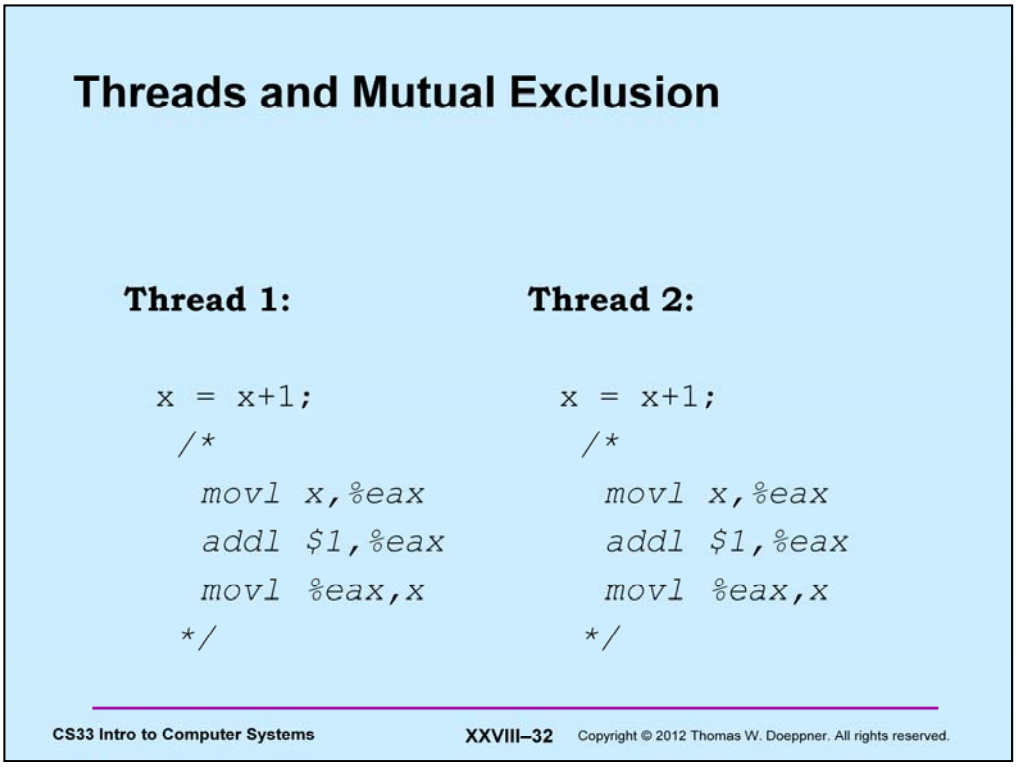
The operating system is responsible for multiplexing the execution of threads on the available processors. The OS's *scheduler* is responsible for assigning threads to processors. Periodically, say every millisecond, each processor is interrupted and calls upon the OS to determine if another thread should run. If so, the current thread on the processor 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 processor, all threads make process and give the appearance of running simultaneously.



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 processor, in the latter they can. The scheduler determines which ready threads should be assigned processors. Ready threads that have been assigned processors are called *running* threads.

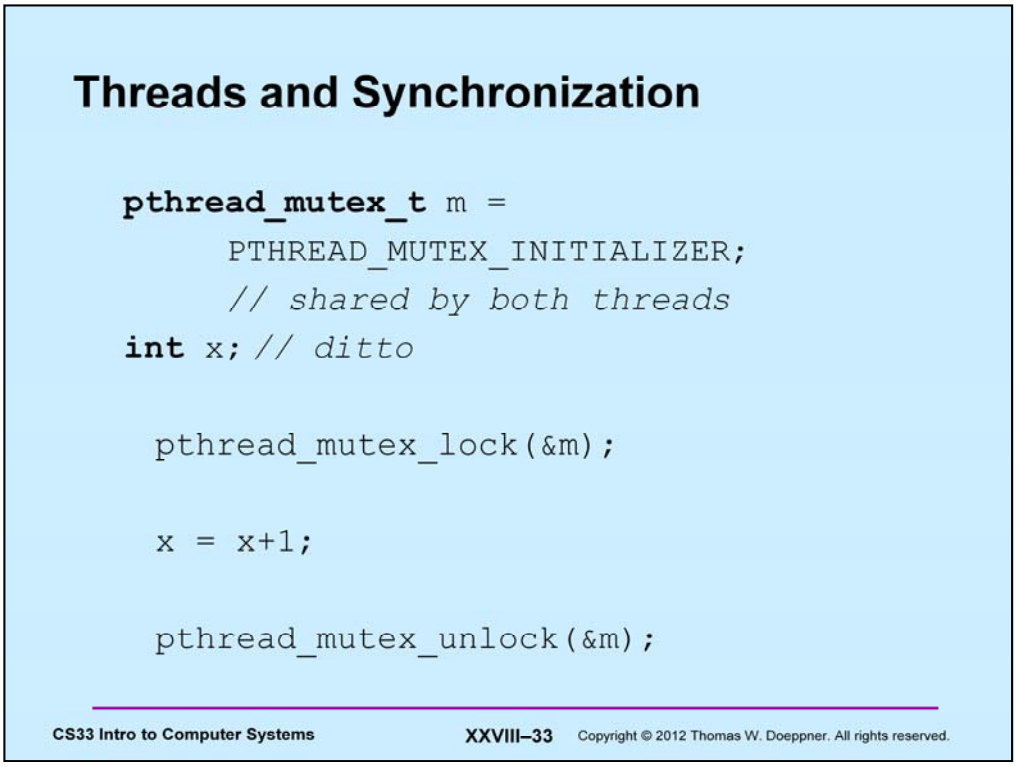


The mutual-exclusion problem involves making certain that two things don't happen at once. A non-computer example arose in the fighter aircraft of World War I (pictured is a Sopwith Camel). Due to a number of constraints (e.g., machine guns tended to jam frequently and thus had to be close to people 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.*



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.

Note that gcc will likely compile "*x = x+1;*" into simply "*addl $1, x*" — a single machine instruction. However, there is still a problem if two threads, each adding one to *x*, run concurrently. This is because, as we discussed earlier in the course, individual instructions are actually executed in multiple steps. In the case of "*addl $1,x*", first *x* must be fetched from memory, then *1* is added to the value, then the result is stored back into memory. If these steps are executed by two threads running simultaneously on different processors, it is likely that the net change to *x* will be that it becomes one greater rather than two greater than what it was.



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*.)

Set Up

```
int pthread_mutex_init(pthread_mutex_t *mutexp,
    pthread_mutexattr_t *attrp)

int pthread_mutex_destroy(pthread_mutex_t *mutexp)

int pthread_mutexattr_init(pthread_mutexattr_t *attrp)

int pthread_mutexattr_destroy(pthread_mutexattr_t *attrp)
```

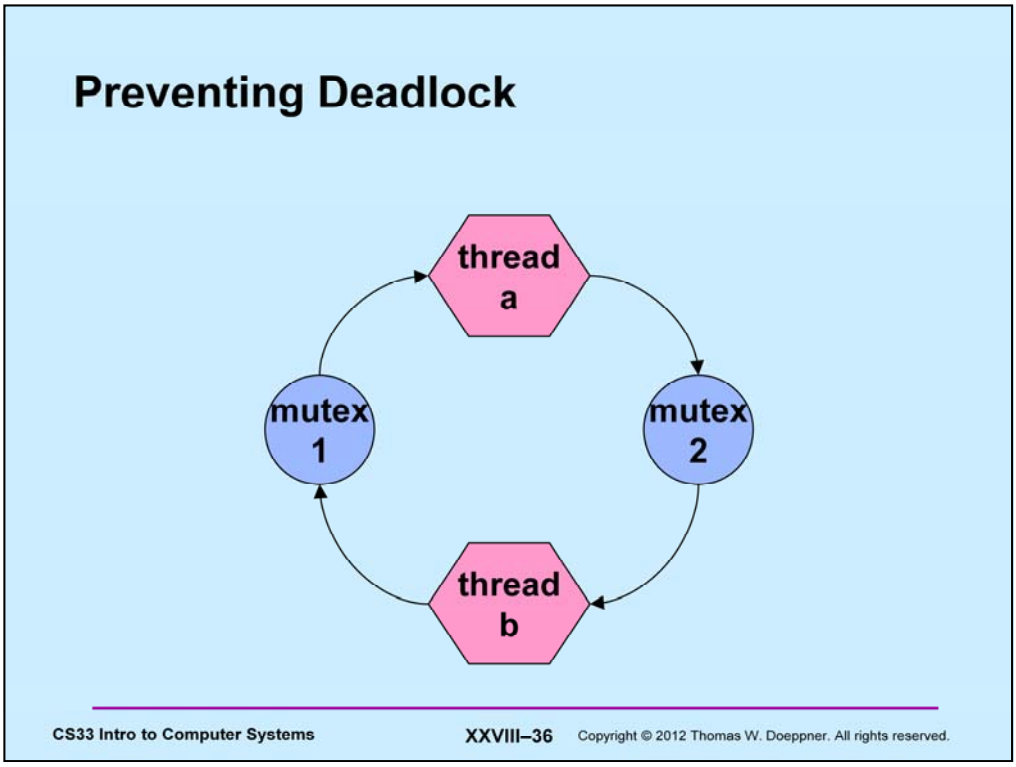

The routines *pthread_mutex_init* and *pthread_mutex_destroy* are supplied to initialize and to destroy a mutex. (They do not allocate or free the storage for the mutex data structure, but in some implementations they might allocate and free storage referred to by the mutex data structure.) As with threads, an attribute structure encapsulates the various parameters that might apply to the mutex. The routines *pthread_mutexattr_init* and *pthread_mutexattr_destroy* control the initialization and destruction of these attribute structures, as we see a few slides from now.

Taking Multiple Locks

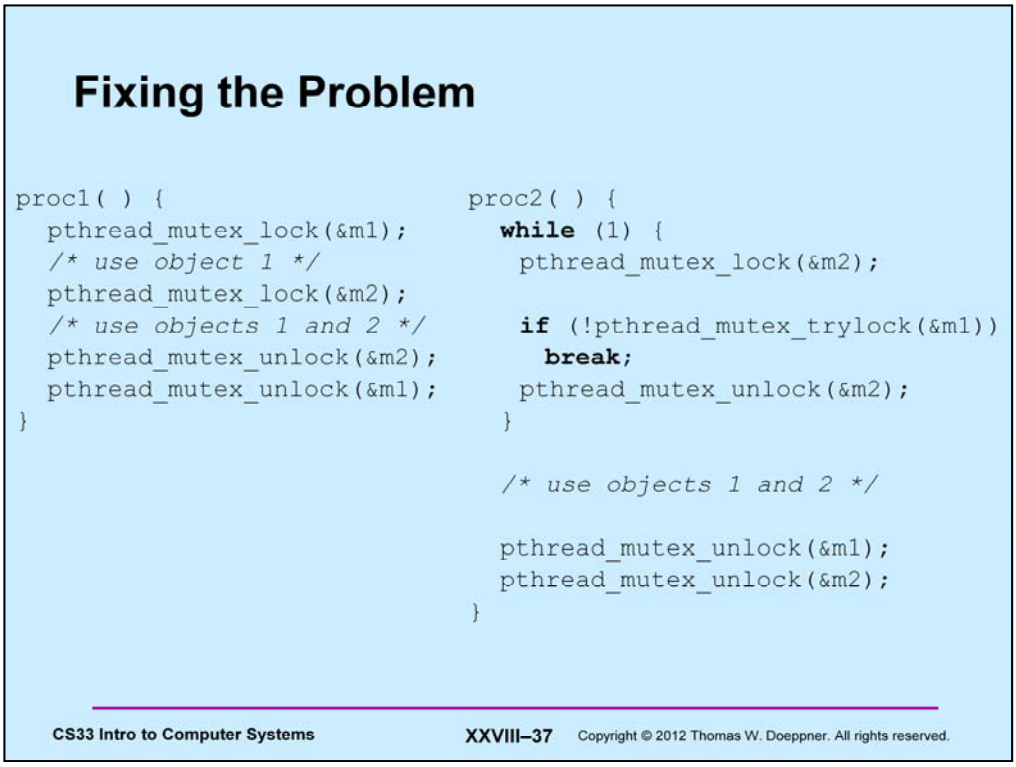
```
proc1( ) {
  pthread_mutex_lock(&m1);
  /* use object 1 */
  pthread_mutex_lock(&m2);
  /* use objects 1 and 2 */
  pthread_mutex_unlock(&m2);
  pthread_mutex_unlock(&m1);
}
```

```
proc2( ) {
  pthread_mutex_lock(&m2);
  /* use object 2 */
  pthread_mutex_lock(&m1);
  /* use objects 1 and 2 */
  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 *proc1*, first takes mutex 1, then, while still holding mutex 1, obtains mutex 2. Thread 2, executing *proc2*, 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*.



Deadlock results when there are circularities in dependencies. In the slide, mutex 1 is held by thread a, which is waiting to take mutex 2. However, thread b is holding mutex 2, waiting to take mutex 1. If we can make certain that such circularities never happen, there can't possibly be deadlock.



The simplest (and best) approach to avoiding deadlocks to make certain that all threads that will hold multiple mutexes simultaneously obtain these mutexes in a prescribed order. Ordinarily this can be done, but occasionally it might turn out to be impossible. For example, we might not know which mutex to take second until the first mutex has already been obtained. To avoid deadlock in such situations, we can use the approach shown in the slide. Here thread 1, executing *proc1*, obtains the mutexes in the correct order. Thread 2, executing *proc2*, must for some reason take the mutexes out of order. If it is holding mutex 2, it must be careful about taking mutex 1. So, rather than call *pthread_mutex_lock*, it calls *pthread_mutex_trylock*, which always returns without blocking. If the mutex is available, *pthread_mutex_trylock* locks the mutex and returns 0. If the mutex is not available (i.e., it is locked by another thread), then *pthread_mutex_trylock* returns a nonzero error code (EBUSY). In the example, if mutex 1 is not available, it is probably because it is currently held by thread 1. If thread 2 were to block waiting for the mutex, we have an excellent chance for deadlock. So, rather than block, thread 1 not only quits trying for mutex 1 but also unlocks mutex 2 (since thread 1 could well be waiting for it). It then starts all over again, first taking mutex 2, then mutex 1.

Stupid Mistakes ...

```
pthread_mutex_lock(&m1);
pthread_mutex_lock(&m1);
  // really meant to lock m2 ...



pthread_mutex_lock(&m1);
  ...
pthread_mutex_unlock(&m2);
  // really meant to unlock m1 ...
```


In the example at the top of the slide, we have mistyped the name of the mutex in the second call to pthread_mutex_lock. The result will be that when *pthread_mutex_lock* is called for the second time, there will be immediate deadlock, since the caller is attempting to lock a mutex that is already locked, but the only thread who can unlock that mutex is the caller.

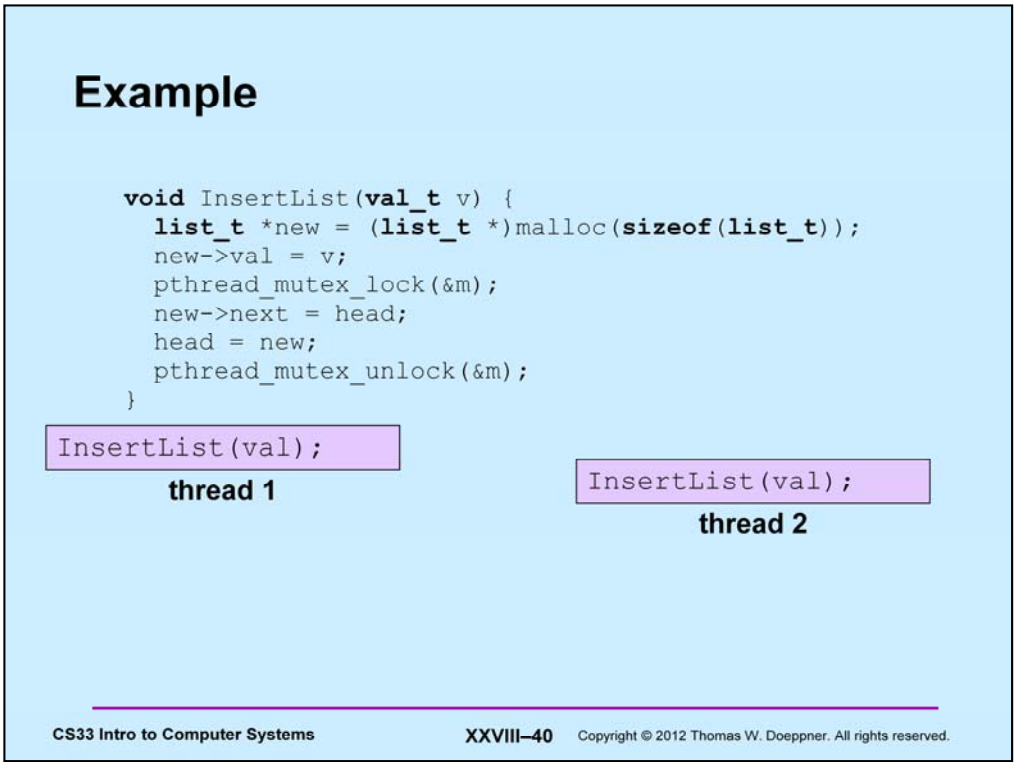
In the example at the bottom of the slide, we have again mistyped the name of a mutex, but this time for a *pthread_mutex_unlock* call. If m2 is not currently locked by some thread, unlocking will have unpredictable results, possibly fatal. If m2 is locked by some thread, again there will be unpredictable results, since a mutex that was thought to be locked (and protecting some data structure) is now unlocked. When the thread who locked it attempts to unlock it, the result will be even further unpredictability.

Runtime Error Checking

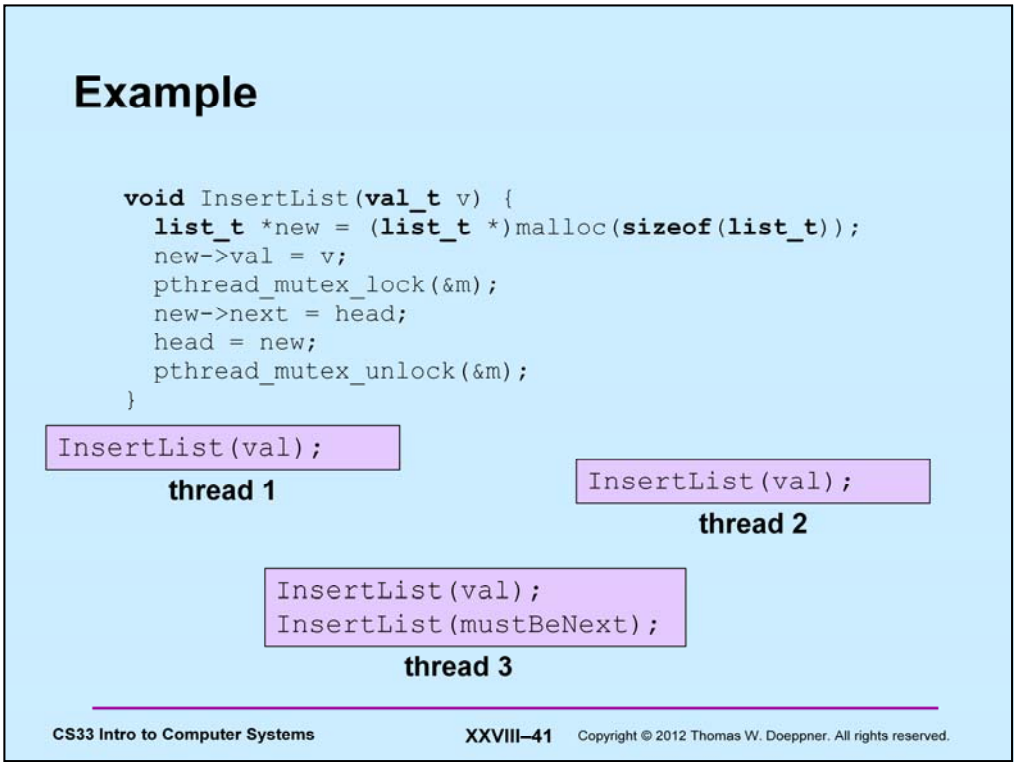
```
pthread_mutexattr_t err_chk_attr;
pthread_mutexattr_init(&err_chk_attr);
pthread_mutexattr_settype(&err_chk_attr,
       PTHREAD_MUTEX_ERRORCHECK);

pthread_mutex_t mut1;
pthread_mutex_init(&mut1, &err_chk_attr);
pthread_mutex_lock(&mut1);
if (pthread_mutex_lock(&mut1) == EDEADLK)
  fprintf(stderr, "error caught at runtime\n");
if (pthread_mutex_unlock(&mut2) == EPERM)
  fprintf(stderr, "another error: you didn't lock it!\n");
```


Checking for some sorts of mutex-related errors is relatively easy to do at runtime (though checking for all possible forms of deadlock is prohibitively expensive). However, since mutexes are used so frequently, even a little bit of extra overhead for runtime error checking is often thought to be too much. Thus, if done at all, runtime error checking is an optional feature. One "turns on" the feature for a particular mutex by initializing it to be of type "ERRORCHECK," as shown in the slide. For mutexes initialized in this way, *pthread_mutex_lock* checks to make certain that it is not attempting to lock a mutex that is already locked by the calling thread; *pthread_mutex_unlock* checks to make certain that the mutex being unlocked is currently locked by the calling thread.

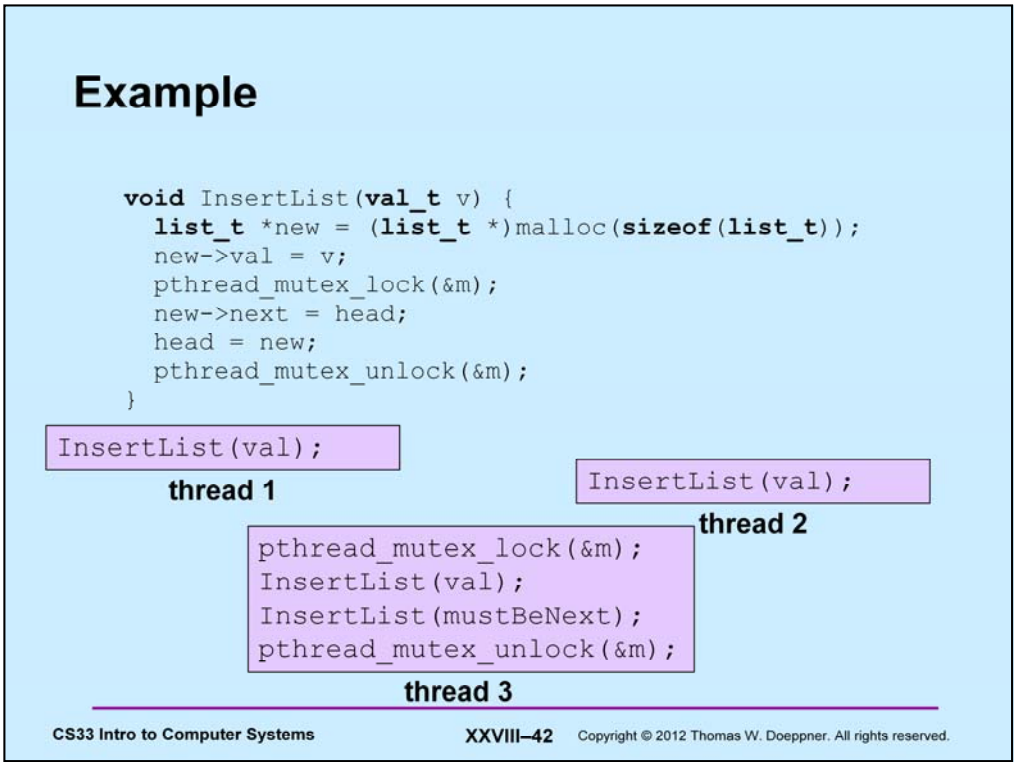
Note that mutexes with the error-check attribute are more expensive than normal mutexes, since they must keep track of which thread, if any, has the mutex locked. (For normal mutexes, just a single bit must be maintained for the state of the mutex, which is either locked or unlocked.)



Here we have a simple routine for inserting an item at the head of a singly linked list. So as to deal with multiple threads calling it concurrently, the routine employs a mutex.



A third thread is calling *InsertList*. However, its intent is that it not only call *InsertList* twice, but that the value inserted in the second call must end up adjacent to the item inserted in the first call.



So as to ensure that no other items are inserted between the two it is inserting, thread 3 locks the mutex that's protecting the list. However, this approach clearly has a problem: after explicitly locking the mutex, thread 3 calls InsertList, which attempts to lock it again. Thus there will be deadlock.

It's clear what the intent is here: thread 3 wants to make certain that the mutex is locked throughout the period from just before it calls InsertList the first time to just after its second call to InsertList returns. One solution, of course, is to remove the lock and unlock calls that are inside of InsertList and require all callers to explicitly lock the mutex before calling and to unlock it on return. This is unsatisfactory for two reasons: it makes all code that uses InsertList more complicated (and error-prone); furthermore, if the need to keep the list locked across two calls to InsertList is a new feature being added to an already working system, all previous calls to InsertList must be modified to accommodate the change. Again, this is error-prone.

A better solution is to somehow fix *pthread_mutex_lock* so that what is shown in the slide actually works.

This is done with the introduction of another mutex attribute, the *recursive* attribute: if a thread has a mutex locked, further calls to lock the same mutex by this thread will not block. Thus the implementation of such recursive mutexes entails maintaining a *mutex owner*, indicating which thread, if any, has the mutex locked, and a *lock count*, which is incremented by one each time the owning thread locks the mutex and decremented each time the thread unlocks it. The mutex is really unlocked (and the owning-thread field cleared) when the lock count is lowered to zero.

Creating Recursive Mutexes

```
pthread_mutexattr_t recursive_attr;
pthread_mutexattr_init(&recursive_attr);
pthread_mutexattr_settype(&recursive_attr,
      PTHREAD_MUTEX_RECURSIVE);

pthread_mutex_t mut;
pthread_mutex_init(&mut, &recursive_attr);
pthread_mutex_lock(&mut);
if (pthread_mutex_lock(&mut) == EDEADLK)
  // won't happen - mutex is recursive
  ;
```


Here we create a mutex of type RECURSIVE: the same thread can lock it any number of times without problems.

Note that recursive mutexes, due to the need to maintain a mutex owner and a lock count, are more expensive than normal mutexes and should be avoided unless really needed.