Source code used in this lecture is available on the course web page.
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 “\textit{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 “\textit{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`.)
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 locked by one thread and then unlocked by another.
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.
If all threads take locks in the same order, deadlock cannot happen.
We have a singly linked list structure to which we’d like multiple threads to be able to add nodes. The `add_after` function, shown in the slide, adds the node given as its second argument after the node given as its first argument. The calls to `pthread_mutex_lock` and `pthread_mutex_unlock` provide appropriate synchronization in case two threads concurrently attempt to add a node after the same node. In this example, we have one mutex protecting the entire list.
We’d now like finer-grained access so that multiple threads may be inserting items into the list at the same time (assuming they’re not all trying to insert after the same node). To do this, we give each node its own mutex.
Now we have a doubly linked list. We modify our fine-grained `add_after` code to work with it. We add a `delete` routine to remove a node from the list. Assume the node being removed is at neither of the two ends of the list.
If we add a new node after B, while at the same time delete node C, we can end up with a list that doesn’t have C completely removed and doesn’t have the new node completed added.
In this version of the code, we recognize that if we are to use or modify either of the pointer fields of a node, we must first lock its mutex.
However, we have a deadlock if one thread locks B’s mutex in `add_after`, while at the same time another thread locks C’s mutex in `delete`. 
In this version of the code, we switch the order of the two calls to `pthread_mutex_lock` in `delete` so as to prevent deadlock.
Doubly Linked List, Attempt 3

```c
void add_after(node_t *after, node_t *new) {
    pthread_mutex_lock(&after->mutex);
    pthread_mutex_lock(&after->next->mutex);
    after->next->prev = new;
    new->next = after->next;
    new->prev = after;
    after->next = new;
    pthread_mutex_unlock(&new->next->mutex);
    pthread_mutex_unlock(&after->mutex);
}

void delete(node_t *old) {
    pthread_mutex_lock(&old->prev->mutex);
    pthread_mutex_lock(&old->mutex);
    pthread_mutex_lock(&old->next->mutex);
    old->prev->next = old->next;
    old->next->prev = old->prev;
    pthread_mutex_unlock(&old->next->mutex);
    pthread_mutex_unlock(&old->mutex);
    pthread_mutex_unlock(&old->prev->mutex);
}
```

Quiz 1

Does this work?

a) yes
b) no
Doubly Linked List

A → B → C → D → E

new
There are sometimes situations in which it’s not possible for all threads to lock mutexes in the same order. 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.
The problem we’re looking at is a special case of what’s known as the “dining philosophers problem”, posed by Edsger Dijkstra in EWD310, first published as Hierarchical Ordering of Sequential Processes in Operating Systems Techniques, C.A.R. Hoare and R.H. Perrot, Eds., Academic Press, New York, 1972. The idea is that we have five philosophers sitting around a table. At the center of the table is a plate of spaghetti. Between each pair of philosophers is a single chopstick (Dijkstra’s original formulation used forks, but chopsticks make more sense). The algorithm of a philosopher is:

```plaintext
while (1) {
    think();
    when available
        grab chopstick from one side();
    when available
        grab chopstick from the other side();
    eat some spaghetti();
    put chopsticks down();
}
```

How long each operation takes varies. Which chopstick is grabbed first is not specified, but if each philosopher grabs his right chopstick first, they may starve to death.

Philosophers clockwise from top: Laozi, Swami Vivekananda, Aristotle, Mary
In this version, which actually works, we use `pthread_mutex_trylock` to avoid deadlock when we take locks out of order. We also make sure that if we use a field within a node, we must have the node locked.

```c
void delete(node_t *old) {
    while(1) {
        pthread_mutex_lock(&old->mutex);
        if (pthread_mutex_trylock(&old->prev->mutex) != 0) {
            pthread_mutex_unlock(&old->mutex);
            continue;
        }
        break;
    }
    pthread_mutex_lock(&old->next->mutex);
    old->prev->next = old->next;
    old->next->prev = old->prev;
    pthread_mutex_unlock(&old->next->mutex);
    pthread_mutex_unlock(&old->mutex);
    pthread_mutex_unlock(&old->prev->mutex);
}
```
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. For most purposes, the default attributes are fine and a NULL `attrp` can be provided to the `pthread_mutex_init` routine.
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.
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.
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.
In the *producer-consumer problem* we have two classes of threads, producers and consumers, and a buffer containing a fixed number of slots. A producer thread attempts to put something into the next empty buffer slot, a consumer thread attempts to take something out of the next occupied buffer slot. The synchronization conditions are that producers cannot proceed unless there are empty slots and consumers cannot proceed unless there are occupied slots.

This is a classic, but frequently occurring synchronization problem. For example, the heart of the implementation of UNIX pipes is an instance of this problem.
Illustrated in the slide is a simple pseudocode construct, the *guarded command*, that we use to describe how various synchronization operations work. The idea is that the code within the square brackets is executed only when the guard (which could be some arbitrary boolean expression) evaluates to true. Furthermore, this code within the square brackets is executed atomically, i.e., the effect is that nothing else happens in the program while the code is executed. Note that the code is not necessarily executed as soon as the guard evaluates to true: we are assured only that when execution of the code begins, the guard is true.

Keep in mind that this is strictly pseudocode: it’s not part of POSIX threads and is not necessarily even implementable (at least not for the general case).
Another synchronization construct is the semaphore, designed by Edsger Dijkstra in the 1960s. A semaphore behaves as if it were a nonnegative integer, but it can be operated on only by the semaphore operations. Dijkstra defined two of these: P (for *prolagen*, a made-up word derived from *proberen te verlagen*, which means “try to decrease” in Dutch) and V (for *verhogen*, “increase” in Dutch). Their semantics are shown in the slide.

We think of operations on semaphores as being a special case of guarded commands — a special case that occurs frequently enough to warrant a highly optimized implementation.
Quiz 2

```
semaphore S = 1;
int count = 0;

void proc( ) {
    P(S);
    count++;
    ...
    count--;
    V(S);
}
```

The function proc is called concurrently by \( n \) threads. What's the maximum value that count will take on?

a) 1  
b) 2  
c) \( n \)  
d) indeterminate

- P(S) operation:
  
  ```
  when (S > 0) [  
      S = S - 1;
  ]
  ```

- V(S) operation:
  
  ```
  [S = S + 1;]
  ```
Here’s a solution for the producer/consumer problem using semaphores — note that it works only with a single producer and a single consumer.
Here is the POSIX interface for operations on semaphores. (This is not a typo — the blasted “pthread_” prefix really is not used here, since the semaphore operations come from a different POSIX specification — 1003.1b. Note also the need for the header file, semaphore.h) When creating a semaphore (sem_init), rather than supplying an attributes structure, one supplies a single integer argument, pshared, which indicates whether the semaphore is to be used only by threads of one process (pshared = 0) or by multiple processes (pshared = 1). The third argument to sem_init is the semaphore’s initial value. All the semaphore operations return zero if successful; otherwise they return an error code.

POSIX Semaphores

```
#include <semaphore.h>

int sem_init(sem_t *semaphore, int pshared, int init);
int sem_destroy(sem_t *semaphore);
int sem_wait(sem_t *semaphore);
    /* P operation */
int sem_trywait(sem_t *semaphore);
    /* conditional P operation */
int sem_post(sem_t *semaphore);
    /* V operation */
```
Here is the producer-consumer solution implemented with POSIX semaphores.

```c
void produce(char item) {
    sem_wait(&empty);
    buf[nextin] = item;
    if (++nextin >= BSIZE)
        nextin = 0;
    sem_post(&occupied);
}

char consume() {
    char item;
    sem_wait(&occupied);
    item = buf[nextout];
    if (++nextout >= BSIZE)
        nextout = 0;
    sem_post(&empty);
    return item;
}
```