

CS 33

Multithreaded Programming II

Creating a Thread

```
pthread_create(pthread_t *thread,  
               pthread_attr_t *attr,  
               void *(start_routine) (void *)  
               void *arg)
```

One calls `pthread_create` to create a new thread. As discussed in the previous lecture, the first argument points to memory into which the new thread's ID will be stored. The second argument supplies attributes for the new thread, something we haven't yet discussed. For now (and for most most purposes) this is zero. The next argument is the address of the new thread's first function -- the function it calls when it starts execution. The final argument is the (single) argument passed to the new thread's first function.

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;
```

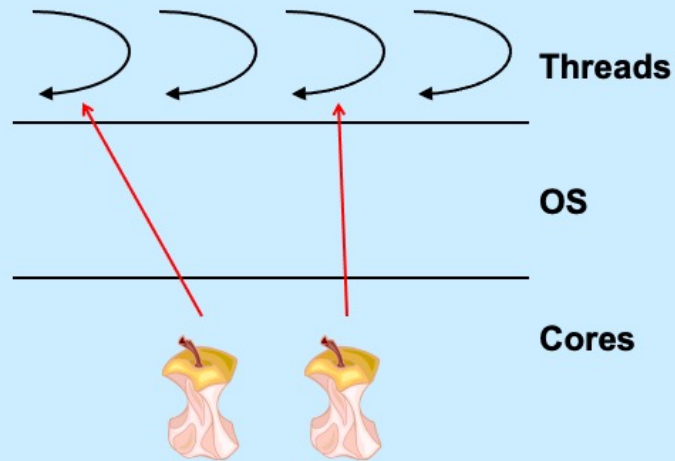
```
void relay(int left, int right) {  
    args_t LRargs, RLargs;  
    pthread_t LRthread, RLthread;  
    ...  
    pthread_create(&LRthread, 0, copy, &LRargs);  
    pthread_create(&RLthread, 0, copy, &RLargs);  
}
```

Quiz 1

Does this work?

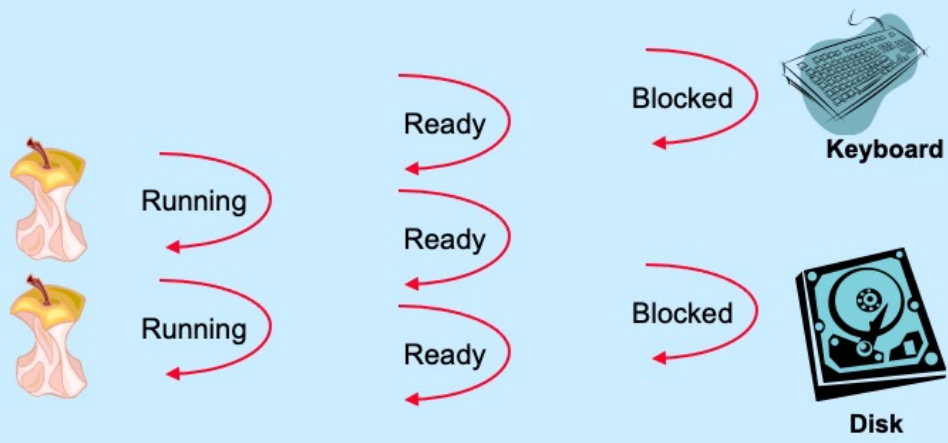
- a) yes
- b) no

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 processor cores. Periodically, say every millisecond, each processor is core and calls upon the OS to determine if another thread should run. If so, the current thread on the 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.

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%d\n", (long)arg);
    return 0;
}
```

In which order are things printed?

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

Cost of Threads

```
int main(int argc, char *argv[]) {
    ...
    int niters = SOME_LARGE_NUMBER;
    val = niters/nthreads;

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

void *work(void *arg) {
    long n = (long)arg; int i, j; volatile long x;

    for (i=0; i<n; i++) {
        x = 0;
        for (j=0; j<1000; j++)
            x = x*j;
    }
    return 0;
}
```

CS33 Intro to Computer Systems XXXII-9 Copyright © 2021 Thomas W. Doepfner. All rights reserved.

We'd like to determine what the actual cost of multithreading is. If we do a certain amount of work using one thread, then do the same amount of work but with it evenly apportioned among a number of threads on a single-core processor, how much longer will it take to complete the work? If the cost of multithreading is zero, then it should take the same amount of time when done with multiple threads than with one thread. (The total amount of work done is the same in the single-thread version as it is in the multi-thread version.)

The idea behind this code is that we compute how long it takes one thread to compute $\text{work}(N)$. We then compute how long it takes for M threads to each compute $\text{work}(N/M)$. The total amount of computation done by these M threads is the same as done by one thread calling $\text{work}(N)$. If we run this on a one-core computer, then the ratio of the time for M threads each computing $\text{work}(N/M)$ to the time of one thread computing $\text{work}(N)$ is the overhead of running M threads. While it's not clear what the function **work** actually does, its purpose is simply to occupy a processor for a fair amount of time, time directly proportional to its argument N .

Similarly, if we run this on a P -core processor, the ratio of the time for PM threads each computing $\text{work}(N/PM)$ to the time of P threads each computing $\text{work}(N/P)$ is the overhead of running PM threads on a P -core processor.

Cost of Threads

```
int main(int argc, char *argv[]) {  
    ...  
    int niters = SOME_LARGE_NUMBER;  
    val = niters/nthreads;  
  
    for (i=0; i<nthreads; i++)  
        pthread_create(&thread, 0, work, (void *)val);  
    pthread_exit(0);  
    return 0;  
}  
  
void *work(void *arg) {  
    long n = (long)arg; int i, j; volatile long x;  
  
    for (i=0; i<n; i++) {  
        x = 0;  
        for (j=0; j<1000; j++)  
            x = x*j;  
    }  
    return 0;  
}
```

Quiz 3

This code runs in time n on a 4-core processor when $nthreads$ is 8. It runs in time p on the same processor when $nthreads$ is 400.

- a) $n \ll p$ (slower)
- b) $n \approx p$ (same speed)
- c) $n \gg p$ (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 4

Suppose gcc produces the following code. 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);
```

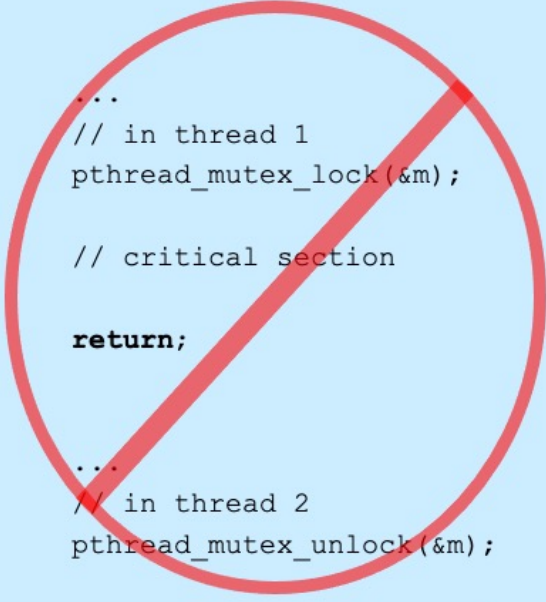
To solve our synchronization problem, we introduce mutexes — a synchronization construct providing mutual exclusion. A mutex is used to insure either that only one thread is executing a particular piece of code at once (code locking) or that only one thread is accessing a particular data structure at 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

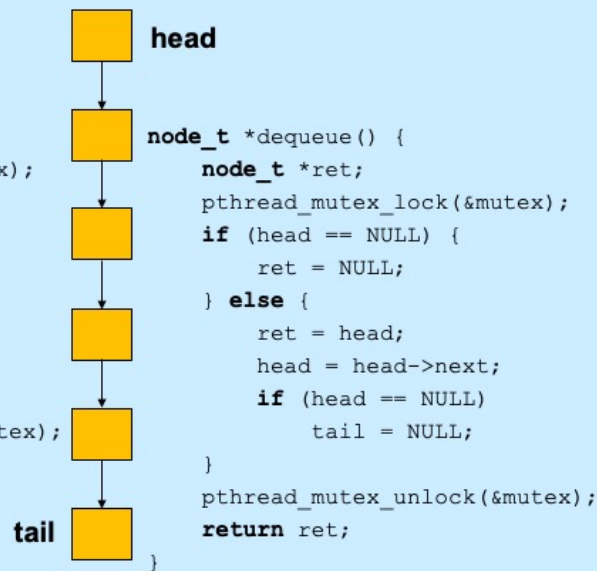
```
pthread_mutex_lock(&m);  
  
// critical section  
  
pthread_mutex_unlock(&m);  
  
...  
// in thread 1  
pthread_mutex_lock(&m);  
  
// critical section  
  
return;  
  
...  
// in thread 2  
pthread_mutex_unlock(&m);
```



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);
}
```



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.

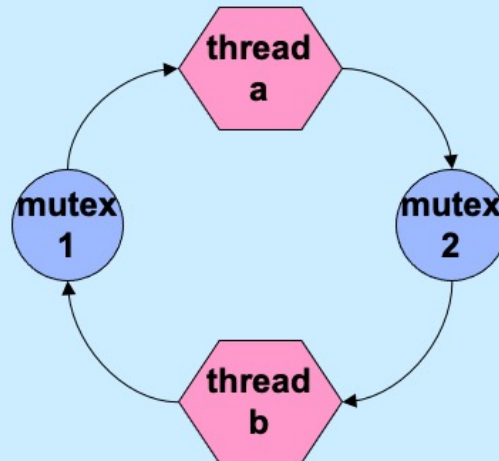
Since enqueue modifies one end of the queue and dequeue modifies the other, one might think that we could have a two-mutex solution, with one mutex protecting one end of the queue and another mutex protecting the other. But this becomes difficult in certain edge conditions. For example, suppose the queue 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).

Taking Multiple Locks

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

Preventing Deadlock



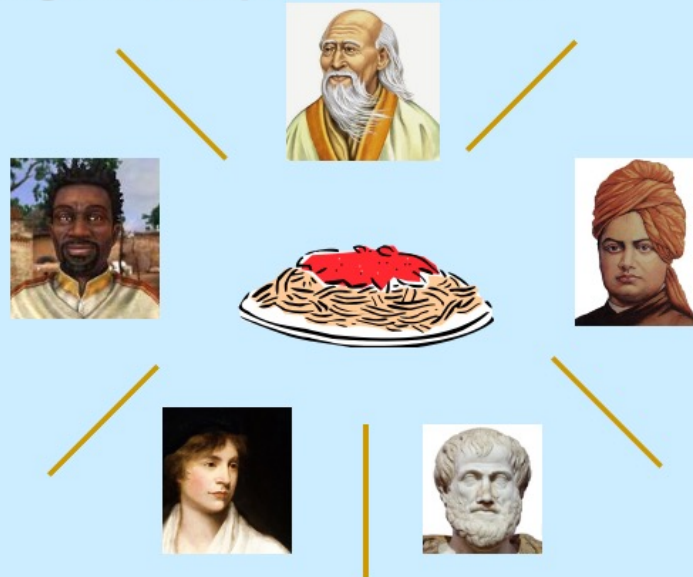
Deadlock results when there are circularities in dependencies. In the slide, mutex 1 is held by thread a, which is waiting to take mutex 2. However, thread b is holding mutex 2, waiting to take mutex 1. If we can make certain that such circularities never happen, there can't possibly be deadlock.

Taking Multiple Locks, Safely

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

If all threads take locks in the same order, deadlock cannot happen.

Dining Philosophers Problem



The problem we've been 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:

```
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 their right chopstick first, they may starve to death. There are many subtle issues involved in its solution. (It has many, none of which are as interesting as the problem itself.)

Philosophers clockwise from top: Laozi, Swami Vivekananda, Aristotle, Mary Wollstonecraft, Zara Yacob.

Practical Issues with Mutexes

- **Used a lot in multithreaded programs**
 - **speed is really important**
 - » shouldn't slow things down much in the success case
 - **checking for errors slows things down (a lot)**
 - » thus errors aren't checked by default

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 functions *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 functions *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.

Note that, as we've already seen, a mutex that's allocated statically may be initialized with `PTHREAD_MUTEX_INITIALIZER`.

Stupid (i.e., Common) 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);

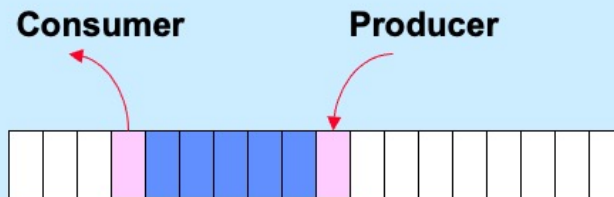
if (pthread_mutex_lock(&mut1) == EDEADLK)
    fprintf(stderr, "error caught at runtime\n");

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.)

Producer-Consumer Problem



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.

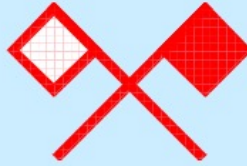
Guarded Commands

```
when (guard) [  
    /*  
        once the guard is true, execute this  
        code atomically  
    */  
  
    ...  
  
]
```

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).

Semaphores



- **P(S) operation:**

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

- **V(S) operation:**

```
[S = S + 1;]
```

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 5

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

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

- **P(S) operation:**

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

- **V(S) operation:**

```
[S = S + 1;]
```

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

- a) 1
- b) 2
- c) n
- d) indeterminate

Producer/Consumer with Semaphores

```
Semaphore empty = BSIZE;
Semaphore occupied = 0;
int nextin = 0;
int nextout = 0;

void Produce(char item) {
    P(empty);
    buf[nextin] = item;
    if (++nextin >= BSIZE)
        nextin = 0;
    V(occupied);
}

char Consume( ) {
    char item;
    P(occupied);
    item = buf[nextout];
    if (++nextout >= BSIZE)
        nextout = 0;
    V(empty);
    return item;
}
```

Here's a solution for the producer/consumer problem using semaphores — note that it works only with a single producer and a single consumer, though it can be generalized to work with multiple producers and consumers.

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 POSIX interface for operations on semaphores. (These operation names are not typos — the “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. The function **sem_trywait** is similar to **sem_wait** (and to the P operation) except that if the semaphore’s value cannot be decremented immediately, then rather than wait, it returns -1 and sets `errno` to `EAGAIN`.

Producer-Consumer with POSIX Semaphores

```
sem_init(&empty, 0, BSIZE);
sem_init(&occupied, 0, 0);
int nextin = 0;
int nextout = 0;

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;
}
```

Here is the producer-consumer solution implemented with POSIX semaphores.

Start/Stop



- Start/Stop interface

```
void wait_for_start(state_t *s);
```

```
void start(state_t *s);
```

```
void stop(state_t *s);
```

We'd like to design a “start-stop” interface. A thread calling **wait_for_start** waits for the **start** button to be pressed. Once it's been pressed, those waiting will be released and subsequent threads calling **wait_for_start** will return immediately. However, once the **stop** button is pressed, then all threads calling **wait_for_start** will wait until the **start** button is pressed again.

Start/Stop



- Start/Stop interface

```
void wait_for_start(state_t *s) {  
    if (s->state == stopped)  
        sleep();  
}  
  
void start(state_t *s) {  
    state = started;  
    wakeup_all();  
}  
  
void stop(state_t *s) {  
    state = stopped;  
}
```

Here's a possible implementation. Callers of **sleep** don't return from sleep until **wakeup_all** has been called.

However, calls to **wakeup_all** merely wakeup all who are currently in **sleep**. They have no effect on subsequent calls to sleep. Thus, there could be a problem in the above code if a thread calls start while another thread has just checked the state in **wait_for_start**, but hasn't yet called **sleep**.

Start/Stop



- Start/Stop interface

```
void wait_for_start(state_t *s) {  
    pthread_mutex_lock(&s->mutex);  
    if (s->state == stopped) {  
        pthread_mutex_unlock(&s->mutex);  
        sleep();  
    }  
    else pthread_mutex_unlock(&s->mutex);  
}  
  
void start(state_t *s) {  
    pthread_mutex_lock(&s->mutex);  
    state = started;  
    wakeup_all();  
    pthread_mutex_unlock(&s->mutex);  
}
```

Here's one attempt to fix the problem of the previous slide using mutexes. It clearly doesn't help – the thread calling start might get the mutex and call wakeup_all just before the other thread calls sleep.

Start/Stop



- Start/Stop interface

```
void wait_for_start(state_t *s) {  
    pthread_mutex_lock(&s->mutex);  
    if (s->state == stopped) {  
        sleep();  
    }  
    pthread_mutex_unlock(&s->mutex);  
}  
  
void start(state_t *s) {  
    pthread_mutex_lock(&s->mutex);  
    state = started;  
    wakeup_all();  
    pthread_mutex_unlock(&s->mutex);  
}
```

This code is perhaps worse, the thread waits in sleep with the mutex locked, preventing any thread from calling wakeup_all.

Start/Stop



- Start/Stop interface

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

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

This code actually works; it uses a POSIX threads construct known as the **condition variable**. The thread in `wait_for_start` first locks the mutex, then checks the state. If it's stopped, it calls **pthread_cond_wait**, which, all at once, put the calling thread to sleep, enqueues it on the queue (known as a condition variable, and unlocks the mutex.

A thread calling `start` can't proceed until it has locked the mutex, thus ensuring that no thread is in the midst of checking the state and then calling **pthread_cond_wait** in `wait_for_start`. Once the thread calling `start` has the mutex, it sets state to started and calls **pthread_broadcast**, waking up all threads who are waiting on the queue (the condition variable). It then unlocks the mutex.

The thread that was waiting within **pthread_cond_wait** is woken up, but it doesn't return from the call to **pthread_cond_wait** until it locks the mutex. Thus, it enters **pthread_cond_wait** with the mutex locked and exits it with the mutex locked. While its inside **pthread_cond_wait**, it does not have the lock on the mutex (though some other thread might).

Thus, this code is a correct implementation of the start/stop interface.

Condition Variables

```
when (guard) [  
    statement 1;  
    ...  
    statement n;  
]  
  
// code modifying the guard:  
...  
  
pthread_mutex_lock(&mutex);  
while (!guard)  
    pthread_cond_wait(  
        &cond_var, &mutex);  
statement 1;  
...  
statement n;  
pthread_mutex_unlock(&mutex);  
  
pthread_mutex_lock(&mutex);  
// code modifying the guard:  
...  
pthread_cond_broadcast(  
    &cond_var);  
pthread_mutex_unlock(&mutex);
```

Condition variables are another means for synchronization in POSIX; they represent queues of threads waiting to be woken by other threads and can be used to implement guarded commands, as shown in the slide. Though they are rather complicated at first glance, they are even more complicated when you really get into them.

A thread puts itself to sleep and joins the queue of threads associated with a condition variable by calling **pthread_cond_wait**. When it places this call, it must have some mutex locked, and it passes the mutex as the second argument. As part of the call, the mutex is unlocked and the thread is put to sleep, **all in a single atomic step**: i.e., nothing can happen that might affect the thread between the moments when the mutex is unlocked and when the thread goes to sleep. Threads queued on a condition variable are released in first-in-first-out order. They are released in response to calls to **pthread_cond_signal** (which releases the first thread in line) and **pthread_cond_broadcast** (which releases all threads). However, before a released thread may return from **pthread_cond_wait**, it first relocks the mutex. Thus, only one thread at a time actually returns from **pthread_cond_wait**. If a call to either function is made when no threads are queued on the condition variable, nothing happens — the fact that a call had been made is not remembered.

So far, though complicated, the description is rational. Now for the weird part: **a thread may be released from the condition-variable queue at any moment**, perhaps spontaneously, perhaps due to sun spots. Thus, it's extremely important that, after **pthread_cond_wait** returns, that the caller check to make sure that it really should have returned. The reason for this weirdness is that it allows a fair amount of latitude in implementations. However, the Linux implementation behaves rationally, i.e., as in the

first two paragraphs. (But don't depend on this behavior — it could change tomorrow!)

Set Up

```
int pthread_cond_init(pthread_cond_t *cvp,  
    pthread_condattr_t *attrp)  
  
int pthread_cond_destroy(pthread_cond_t *cvp)  
  
int pthread_condattr_init(pthread_condattr_t *attrp)  
  
int pthread_condattr_destroy(pthread_condattr_t *attrp)
```

Setting up condition variables is done in a similar fashion as mutexes: The functions **pthread_cond_init** and **pthread_cond_destroy** are supplied to initialize and to destroy a condition variable. They may also be statically initialized by setting them to `PTHREAD_COND_INITIALIZER` in their declarations. As with mutexes and threads, default attributes may be specified by supplying a zero. The functions **pthread_condattr_init** and **pthread_condattr_destroy** control the initialization and destruction of their attribute structures.

PC with Condition Variables (1)

```
typedef struct buffer {  
    pthread_mutex_t m;  
    pthread_cond_t  more_space;  
    pthread_cond_t  more_items;  
    int             next_in;  
    int             next_out;  
    int             empty;  
    char            buf[BSIZE];  
} buffer_t;
```

Here we begin a producer-consumer solution using condition variables and mutexes; this solution, unlike the previous, allows multiple producers and consumers. We define a struct *buffer* to represent a buffer, associated synchronization variables, and other associated variables. In our example, producers wait for empty slots to become available, and consumers wait for occupied slots to become available. Waiting producers are queued on the condition variable **more_space** and waiting consumers are queued on the condition variable **more_items**.

PC with Condition Variables (2)

```
void produce(buffer_t *b,
             char item) {
    pthread_mutex_lock(&b->m);
    while (!(b->empty > 0))
        pthread_cond_wait(
            &b->more_space, &b->m);
    b->buf[b->nextin] = item;
    if (++(b->nextin) == BSIZE)
        b->nextin = 0;
    b->empty--;
    pthread_cond_signal(
        &b->more_items);
    pthread_mutex_unlock(&b->m);
}

char consume(buffer_t *b) {
    char item;
    pthread_mutex_lock(&b->m);
    while (!(b->empty < BSIZE))
        pthread_cond_wait(
            &b->more_items, &b->m);
    item = b->buf[b->nextout];
    if (++(b->nextout) == BSIZE)
        b->nextout = 0;
    b->empty++;
    pthread_cond_signal(
        &b->more_space);
    pthread_mutex_unlock(&b->m);
    return item;
}
```

Here we have the remaining code of our solution. A producer, if there is at least one empty slot, fills the one at location **nextin**, increments **nextin** (taking wraparound into account), calls **pthread_cond_signal** to notify any waiting consumers that there is now an occupied slot in the buffer, and releases the mutex. If there are no empty slots in the buffer, the producer calls **pthread_cond_wait** to wait for one.

As discussed previously, this call has a fairly complicated effect: it releases the mutex given as the second argument and puts its caller to sleep, after queuing it on the condition variable given as the first argument. At some point in the future, a consumer should call **pthread_cond_signal**, with **more_space** as the argument.

Note that we've used **pthread_cond_signal** rather than **pthread_cond_broadcast**. We can do this here since, if, for example, n threads are waiting within the call to **pthread_cond_wait** in the producer, then there must be n calls to *consume* to release them all. If we'd used **pthread_cond_broadcast** instead, the solution would still work, but would probably be less efficient, since in many cases waiting threads would return from **pthread_cond_wait**, discover that the guard is still false, and have to call **pthread_cond_wait** again.

If our producer is the first in the queue associated with **more_space**, it is released from the queue, but it does not yet return from **pthread_cond_wait**. Instead, it continues execution inside that routine, where it effectively makes a call to **pthread_mutex_lock** to reacquire the mutex it had when it entered **pthread_cond_wait** in the first place. Once it obtains the mutex, it then returns from **pthread_cond_wait**. Note that when the thread attempts to reacquire the mutex, other threads might be waiting for the mutex at the entrance of the producer code. One of these other threads might obtain the mutex first — thus there is no guarantee that callers of *produce* are served in FIFO order.

The order in which threads are released from a condition variable's queue is first-in-first-out within priority levels. Thus, waiting high-priority threads are released before waiting low-priority threads; threads of the same priority are released in the order in which they called **pthread_cond_wait**.