# CS 33

## Multithreaded Programming II

## Example (1)

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

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

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

void *matmult(void *);
```

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

  // initialize the matrices
  ...
```

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

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

## Example (2)

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

Here we have the remainder of *main*. It creates a number of threads, one for each row of the result matrix, waits for all of them to terminate, then prints the results (this last step is not spelled out). Note that we check for errors when calling *pthread_create*. (It is important to check for errors after calls to almost all of the pthread 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)

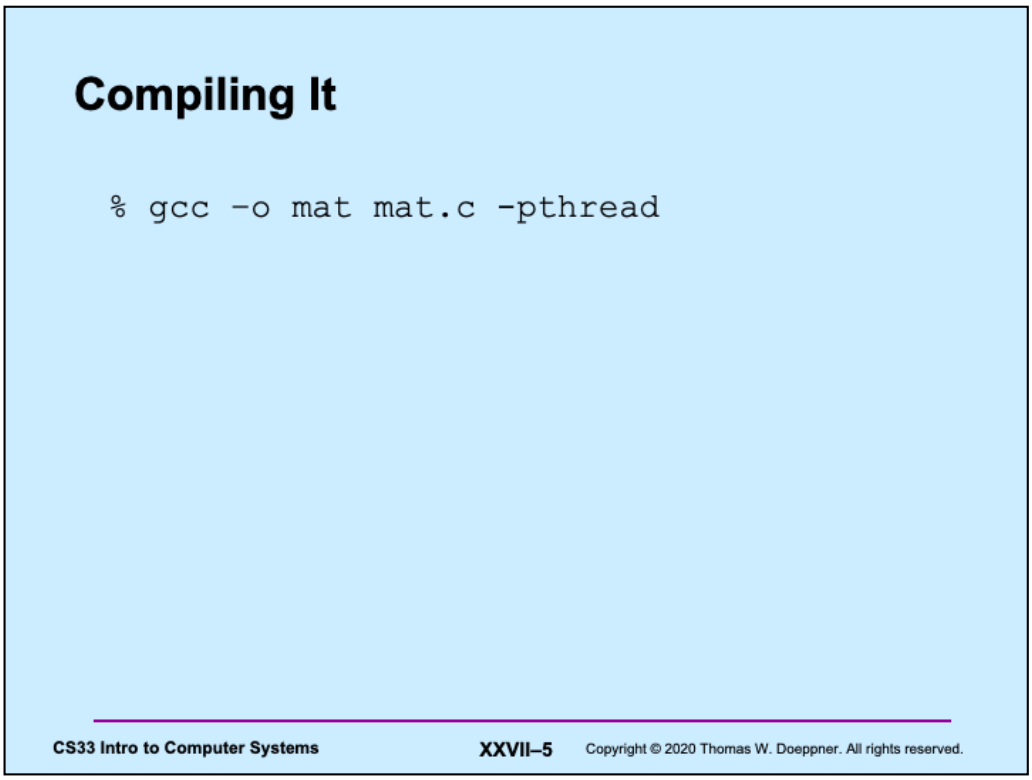
```
void *matmult(void *arg) {
  long row = (long)arg;
  long col;
  long i;
  long t;

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

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

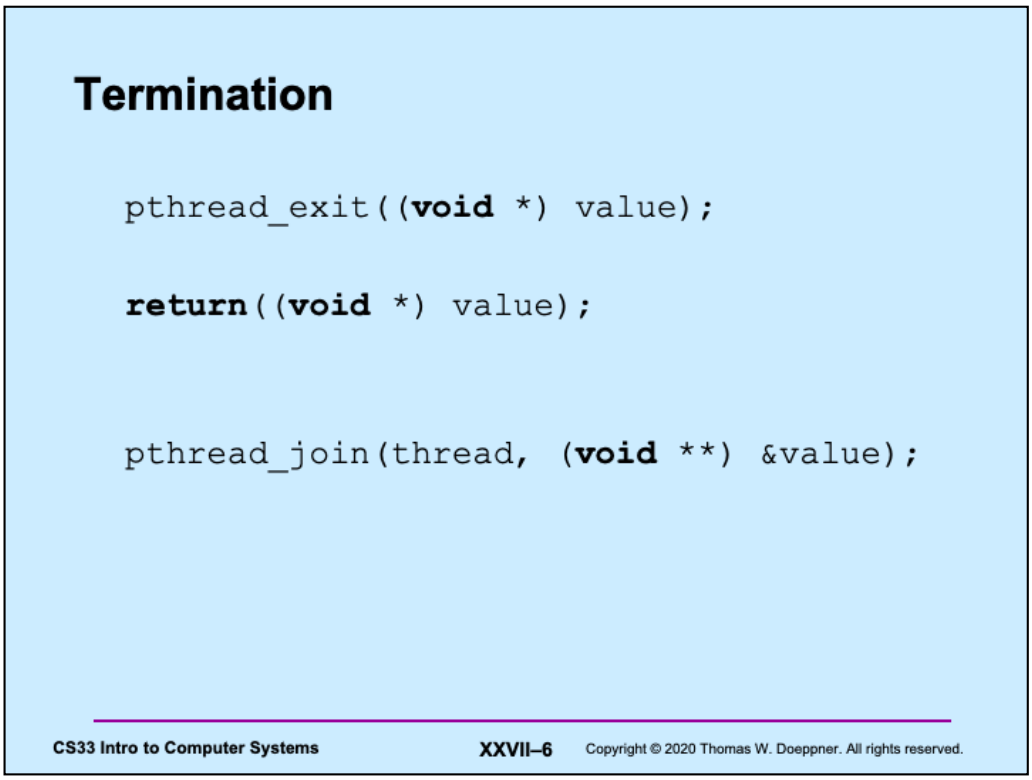
Note how the argument is explicitly converted from *void** to *long*.

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




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

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




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);
  }
  ...
}

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

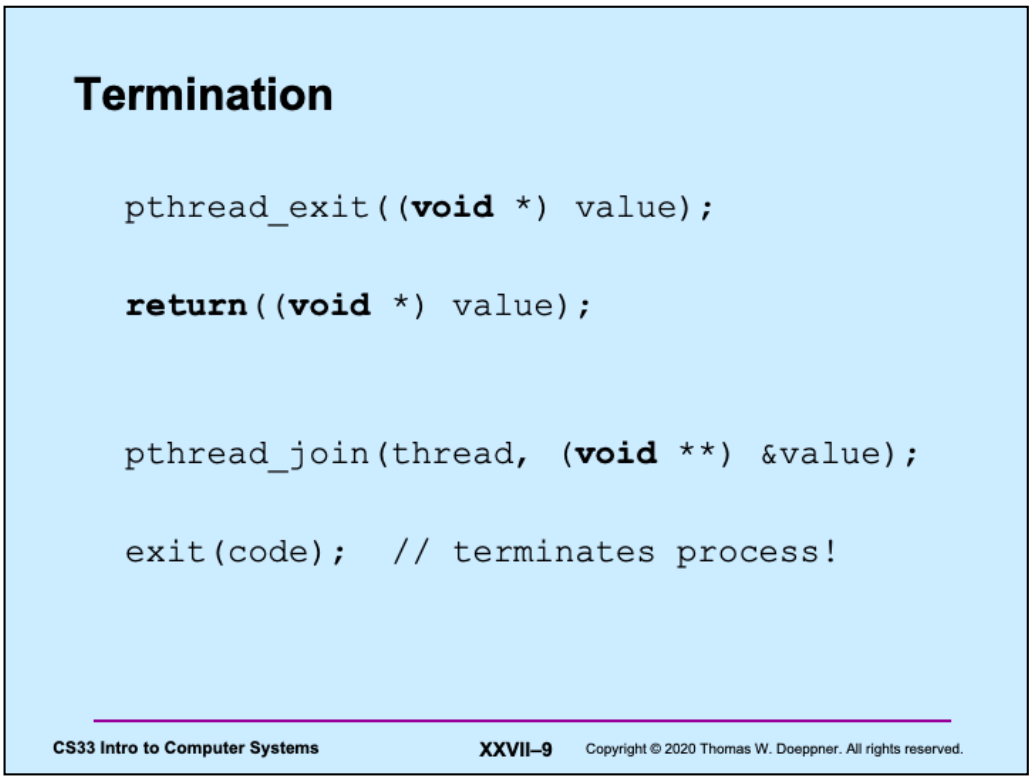
If there is no reason to synchronize with the termination of a thread, then it is rather a nuisance to have to call *pthread_join*. Instead, one can arrange for a thread to be *detached*. Such threads "vanish" when they terminate — not only do they not need to be joined, but they cannot be joined.

## Worker Threads

```
int main() {
  pthread_t thread[10];
  for (int i=0; i<10; i++)
    pthread_create(&thread[i], 0,
        worker, (void *)i);
  return 0;
}

void *worker(...) {...}
```

This program will probably do nothing! This is because the first thread returns from main right away, which is equivalent to calling exit. Calling exit terminates all the threads in the process.




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

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

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

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

## Complications

```
void relay(int left, int right) {
  pthread_t LRthread, RLthread;

  pthread_create(&LRthread,
      0,
      copy,
      left, right);          // Can't do this ...
  pthread_create(&RLthread,
      0,
      copy,
      right, left);          // Can't do this  ...
}
```

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

## Multiple Arguments

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

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

To pass more than one argument to the first procedure of a thread, we must somehow encode multiple arguments as one. Here we pack two arguments into a structure, then pass the pointer to the structure.

# Multiple Arguments

```
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);
}
```

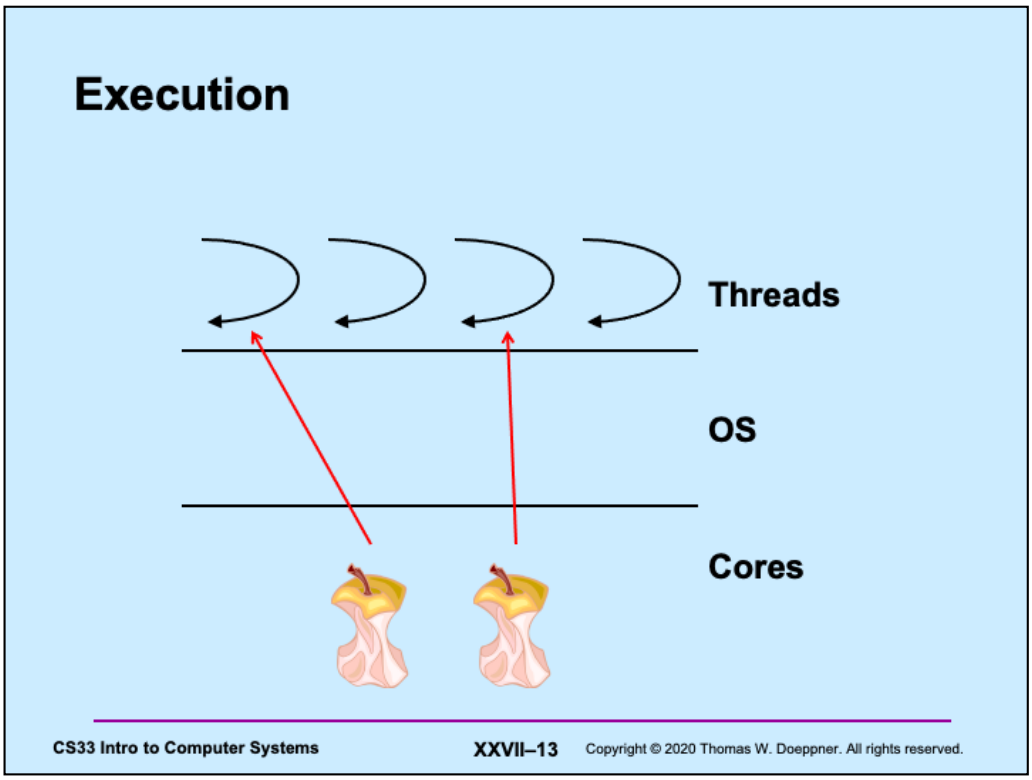
**Quiz 1**

**Does this work?**

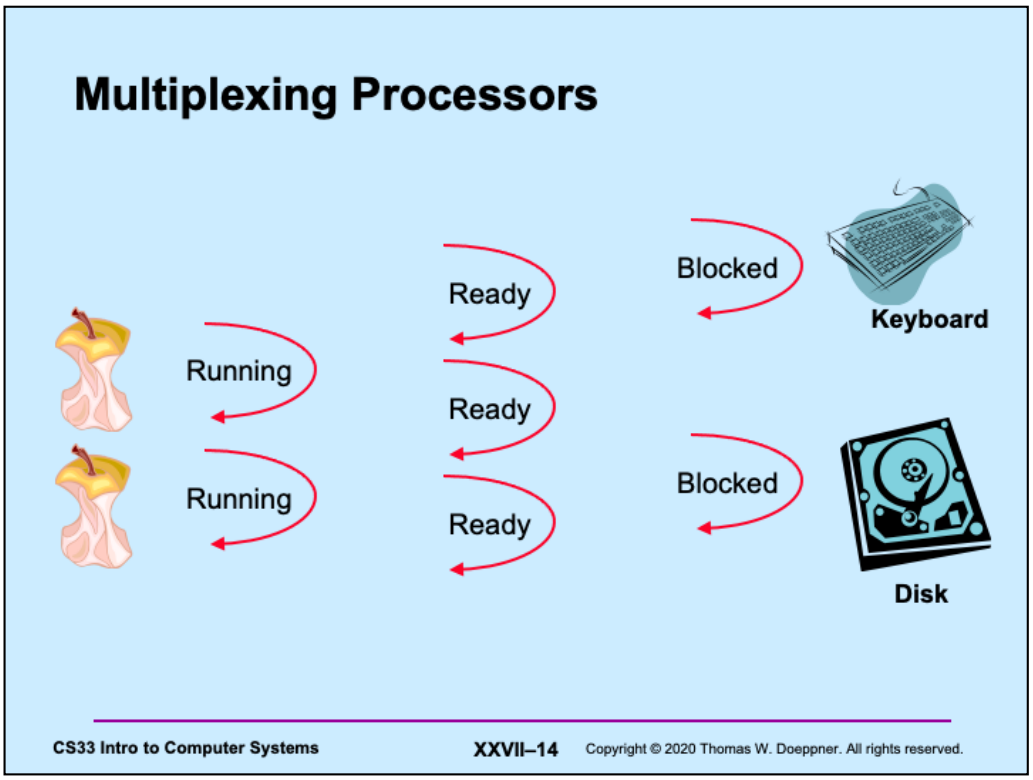
a) **yes**

b) **no**




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.




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

# Quiz 2

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

  printf("T0\n");

  ...

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

**In which order are things printed?**

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

## Cost of Threads

```
int main(int argc, char *argv[]) {
   ...
   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;
}
```

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. The idea behind this code is that we compute how long it takes one thread to compute work(N). We then compute how long it takes for M threads to each compute work(N/M). The total amount of computation done by these M threads is the same is done by one thread calling work(N). If we run this on a one-core computer, then the ratio of the time for M threads each computing work(N/M) to the time of one thread computing work(N) is the the overhead of running M threads.

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

# Cost of Threads

```
int main(int argc, char *argv[]) {
   ...
   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 (void *)x;
}
```

**Not a Quiz**

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

## Problem

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

  …


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

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

## Thread Attributes

```
pthread_t thread;
pthread_attr_t thr_attr;

pthread_attr_init(&thr_attr);

...

/* establish some attributes */

...

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

...
```

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

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

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

## Stack Size

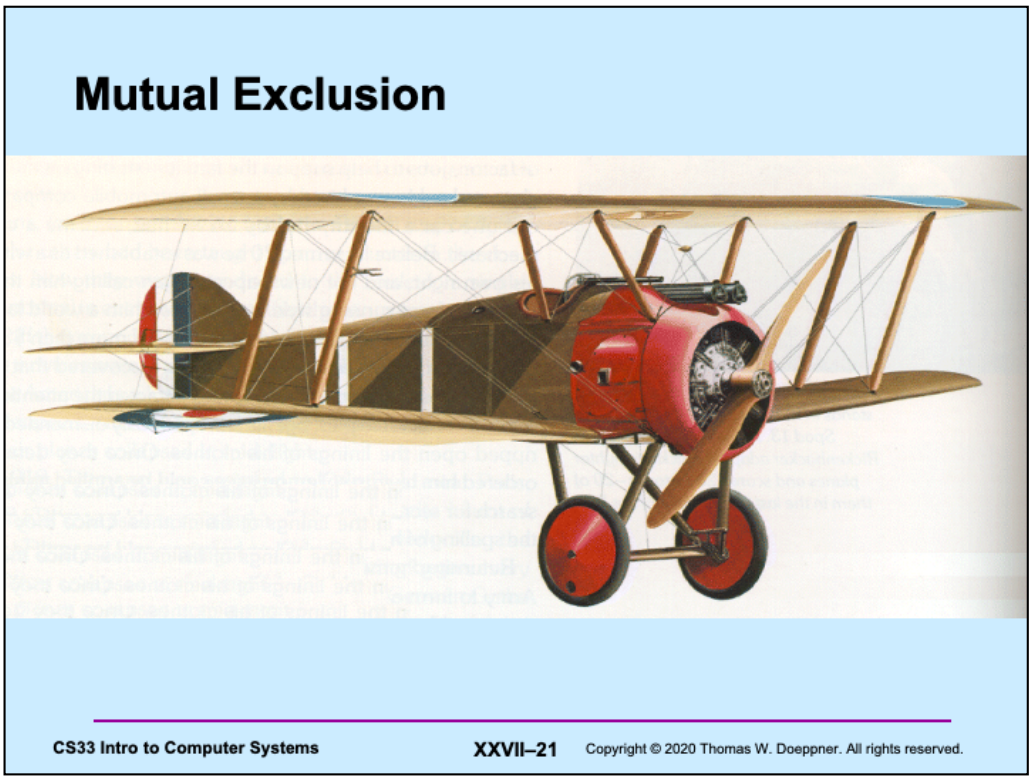
```
pthread_t thread;
pthread_attr_t thr_attr;

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

...

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

...
```

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

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.




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

## Threads and Mutual Exclusion

**Thread 1:**

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

**Thread 2:**

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

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

## Quiz 3

**Suppose 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);
```

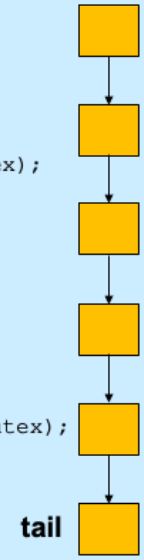
To solve our synchronization problem, we introduce mutexes — a synchronization construct providing mutual exclusion. A mutex is used to insure either that only one thread is executing a particular piece of code at once (code locking) or that only one thread is accessing a particular data structure at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

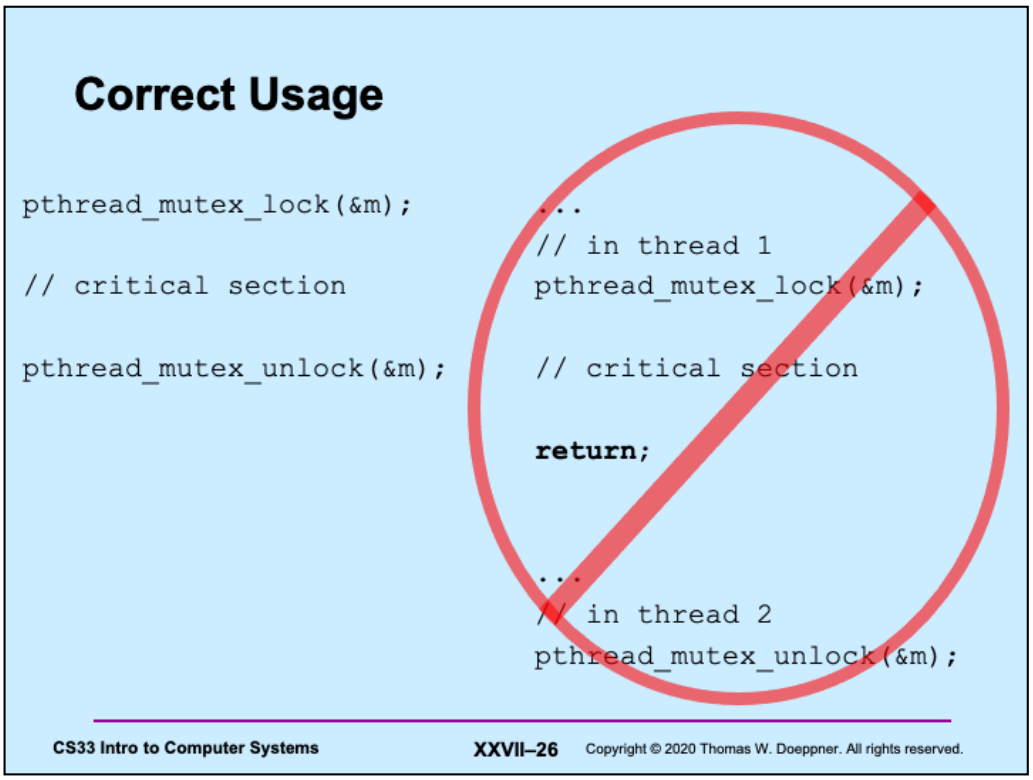
Like any other data structure, mutexes must be initialized. This can be done via a call to *pthread_mutex_init* or can be done statically by assigning PTHREAD_MUTEX_INITIALIZER to a mutex. The initial state of such initialized mutexes is unlocked. Of course, a mutex should be initialized only once! (I.e.,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

# A Queue

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




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




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.

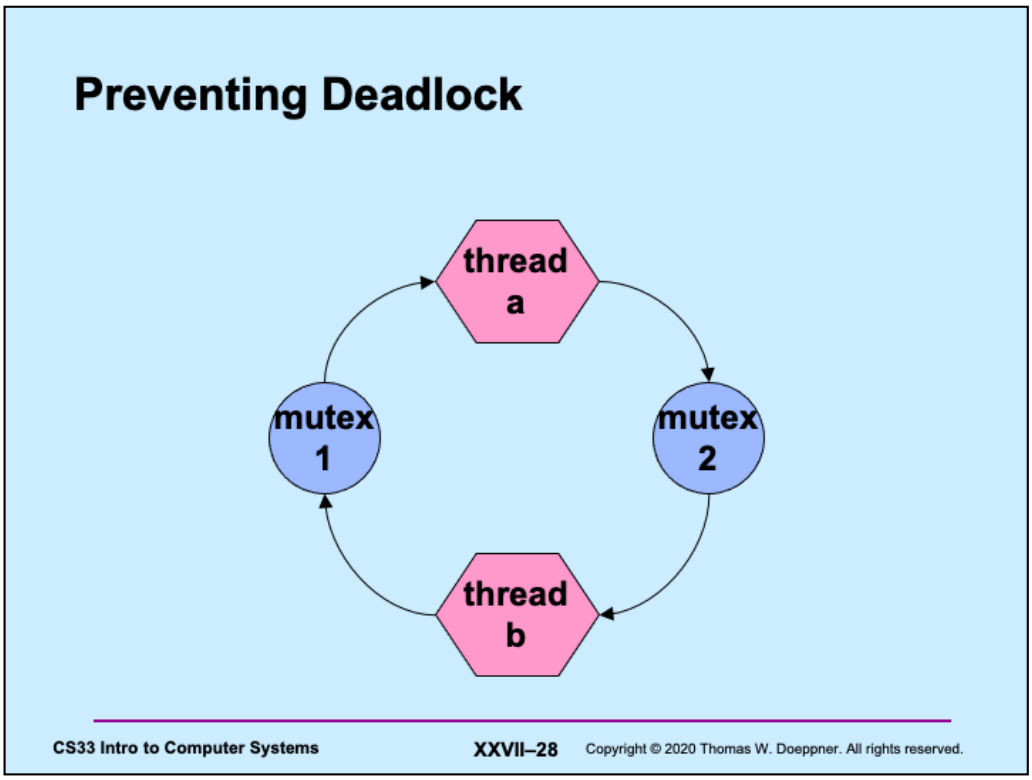
## Taking Multiple Locks

```
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.

## 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.

# 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.

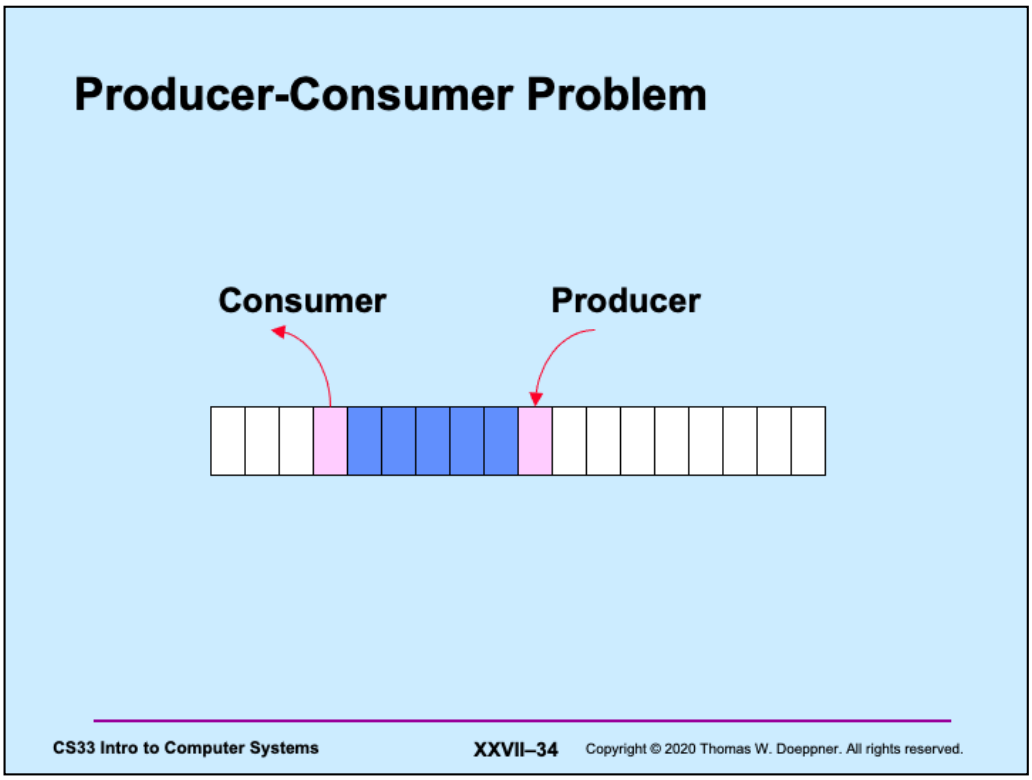
## Runtime Error Checking

```
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.)




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

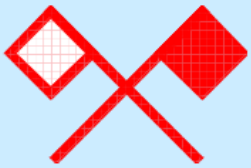
```
when (guard) [
  /*
     once the guard is true, execute this
     code atomically
   */

  ...

]
```

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 4

```
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;]
```

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

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