Computer Science 322
Operating Systems
Mount Holyoke College
Spring 2008

Topic Notes: Process Synchronization

Cooperating Processes

An *Independent* process is not affected by other running processes.

Cooperating processes may affect each other, hopefully in some controlled way.

Why cooperating processes?

- information sharing
- computational speedup
- modularity or convenience

It's hard to find a computer system where processes do not cooperate. Consider the commands you type at the Unix command line. Your shell process and the process that executes your command must cooperate. If you use a pipe to hook up two commands, you have even more process cooperation (See the shell lab later this semester).

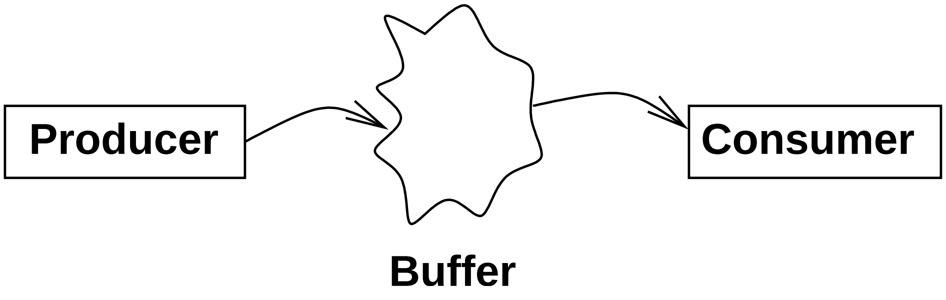
For the processes to cooperate, they must have a way to communicate with each other. Two common methods:

- shared variables – some segment of memory accessible to both processes
- message passing – a process sends an explicit message that is received by another

For now, we will consider shared-memory communication. We saw that threads, for example, share their global context, so that is one way to get two processes (threads) to share a variable.

Producer-Consumer Problem

The classic example for studying cooperating processes is the Producer-Consumer problem.



One or more produces processes is "producing" data. This data is stored in a buffer to be "consumed" by one or more consumer processes.

The buffer may be:

- *unbounded* – We assume that the producer can continue producing items and storing them in the buffer at all times. However, the consumer must wait for an item to be inserted into the buffer before it can take one out for consumption.
- *bounded* – The producer must also check to make sure there is space available in the buffer.

Bounded Buffer, buffer size n

For simplicity, we will assume the objects being produced and consumed are `int` values.

This solution leaves one buffer entry empty at all times:

- Shared data

```
int buffer[n];
int in=0;
int out=0;
```

- Producer process

```
while (1) {
  ...
  produce item;
  ...
  while (((in+1)%n) == out); /* busy wait */
  buffer[in]=item;
  in=(in+1)%n;
}
```

- Consumer process

```
while (1) {
  while (in==out); /* busy wait */
  item=buffer[out];
  out=(out+1)%n;
  ...
  consume item;
  ...
}
```

See Example:
`/cluster/examples/prodcons-shmem`

See Example:
`/cluster/examples/prodcons-pthreads`

Is there any danger with this solution in terms of concurrency? Remember that these processes can be interleaved in any order – the system could preempt the producer at any time and run the consumer.. Things to be careful about are shared references to variables.

Note that only one of the processes can *modify* the variables `in` and `out`. Both use the values, but only the producer modifies `in` and only the consumer modifies `out`. Try to come up with a situation that causes incorrect behavior – hopefully you cannot.

Perhaps we want to use the entire buffer...let's add a variable to keep track of how many items are in the buffer, so we can tell the difference between an empty and a full buffer:

- Shared data

```
int buffer[n];
int in=0;
int out=0;
int counter=0;
```

- Producer process

```
while (1) {
  ...
  produce item;
  ...
  while (counter==n); /* busy wait */
  buffer[in]=item;
  in=(in+1)%n;
  counter=counter+1;
}
```

- Consumer process

```
while (1) {
  while (counter==0); /* busy wait */
  item=buffer[out];
  out=(out+1)%n;
  counter=counter-1;
  ...
  consume item;
  ...
}
```

We can now use the entire buffer. However, there is a potential danger here. We modify `counter` in both the producer and the consumer.

See Example:
`/cluster/examples/prodcons-shmem`

See Example:
`/cluster/examples/prodcons-pthreads`

Everything looks fine, but let's think about how a computer actually executes those statements to increment or decrement `counter`.

`counter++` really requires three machine instructions: (i) load a register with the value of `counter`'s memory location, (ii) increment the register, and (iii) store the register value back in `counter`'s memory location. There's no reason that the operating system can't switch the process out in the middle of this.

Consider the two statements that modify `counter`:

Producer		Consumer	
P_1	`R0 = counter;`	C_1	`R1 = counter;`
P_2	`R0 = R0 + 1;`	C_2	`R1 = R1 - 1;`
P_3	`counter = R0;`	C_3	`counter = R1;`

Consider one possible ordering: P_1 P_2 C_1 P_3 C_2 C_3 , where `counter=17` before starting. Uh oh.

What we have here is a *race condition*.

You may be thinking, "well, what are the chances, one in a million that the scheduler will choose to preempt the process at exactly the wrong time?"

Doing something millions or billions of times isn't really that unusual for a computer, so it would come up..

Some of the most difficult bugs to find in software (often in operating systems) arise from race conditions.

This sort of interference comes up in painful ways when "real" processes are interacting.

Consider two processes modifying a linked list, one inserting and one removing. A context switch at the wrong time can lead to a corrupted structure:

```
struct node {
  ...
  struct node *next;
}

struct node *head, *tail;

void insert(val) {
  struct node *newnode;
  newnode = getnode();
  newnode->next = NULL;
  if (head == NULL){
     head = tail = newnode;
   } else {     //  <==== THE WRONG PLACE
     tail->next = newnode;
     tail = newnode;
   }
}

void remove() {
  // ... code to remove value ...
  head = head->next;
  if (head == NULL) tail = NULL;
  return (value);
}
```

If the process executing insert is interrupted at “the wrong place” and then another process calls remove until the list is empty, when the insert process resumes, it will be operating on invalid assumptions and the list will be corrupted.

In the bounded buffer, we need to make sure that when one process starts modifying `counter`, that it finishes before the other can try to modify it. This requires *synchronization* of the processes.

Process synchronization is one of the major topics of this course, and one of the biggest reasons I think every undergraduate CS major should take an OS course.

If there were mutliple producers or consumers, we would have the same issue with the modification of `in` and `out`, so we can’t rely on the “empty slot” approach in the more general case.

We need to make those statements that increment and decrement `counter` *atomic*.

We say that the modification of `counter` is a *critical section*.

Critical Sections

The Critical-Section problem:

- n processes, all competing to use some shared data
- each process has a code segment (the critical section) in which shared data is accessed

```
      while (1) {
         <CS Entry>
         critical section
```

```
        <CS Exit>
        non-critical section
    }
```

- Need to ensure that when one process is executing in its critical section, no other process is allowed to do so

Example: Intersection/traffic light analogy

Example: one-lane bridges during construction

Any solution to the critical section problem must satisfy three conditions:

1. *Mutual exclusion*: If process P_i is executing in its critical section, then no other processes can be executing in their critical sections. “One at a time.”
2. *Progress*: If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely. “no unnecessary waiting.”
3. *Bounded waiting*: A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted. “no starvation.” (We must assume that each process executes at non-zero speed, but make no assumptions about relative speeds of processes)

One possible way to deal with this is to make sure the problematic context switch doesn’t happen.

If we disable interrupts so a context switch isn’t possible while we’re executing the critical section, we will prevent the interference.

However, this is a bit extreme, since it doesn’t just restrict another process that will be modifying the same shared variable from being switched in, it prevents ANY other process from being switched in.

This approach would also not work in a multiprocessor environment when the interference could be from two processes running truly concurrently.

Algorithmic Approaches for 2 Processes

We first attempt to solve this for two processes, P_0 and P_1. They share some variables to synchronize. We fill in `<CS Entry>` and `<CS Exit>` from above with code that should satisfy the three conditions.

Critical Section Algorithm 1

- Shared data

```
int turn=0;
```

- Process P_i (define $j = 1 - i$, the other process)

```
while (1) {
  while (turn!=i); /* busy wait */

    /* critical section */

  turn=j;

    /* non-critical section */

}
```

Note the semicolon at the end of the while statement's condition at the line labeled "busy wait" above. This means that P_i just keeps comparing `turn` to `i` over and over until it succeeds. This is sometimes called a *spin lock*. For now, this is our only method of making one process wait for something to happen. More on this later.

This does satisfy mutual exclusion, but not progress (alternation is forced).

Critical Section Algorithm 2

We'll avoid this alternation problem by having a process wait only when the other has "indicated interest" in the critical section.

- Shared data

```
boolean flag[2];
flag[0]=false;
flag[1]=false;
```

- Process P_i

```
while (1) {
  flag[i]=true;
  while (flag[j]);

    /* critical section */

  flag[i]=false;

    /* non-critical section */

}
```

`flag[i]` set to `true` means that P_i is requesting access to the critical section.

This one also satisties mutual exclusion, but not progress.

Both can set their flags, then both start waiting for the other to set `flag[j]` back to false. Not going to happen...

If we swap the order of the `flag[i]=true;` and `while (flag[j]);` statements, we no longer satisfy mutual exclusion.

Critical Section Algorithm 3

We combine the two previous approaches:

- Shared data

```
int turn=0;
boolean flag[2];
flag[0]=false;
flag[1]=false;
```

- Process P_i

```
while (1) {
  flag[i]=true;
  turn=j;
  while (flag[j] && turn==j);

    /* critical section */

  flag[i]=false;

    /* non-critical section */

}
```

So, we first indicate interest. Then we set `turn=j;`, effectively saying “no, you first” to the other process. Even if both processes are interested and both get to the while loop at the “same” time, only one can proceed. Whoever set `turn` first gets to go first.

This one satisfies all three of our conditions. This is known as *Peterson’s Algorithm*.

Peterson’s Algorithm in action:

See Example:
`/cluster/examples/prodcons-pthreads`

Algorithmic Approach for n Processes: Bakery algorithm

Can we generalize this for n processes? The Bakery Algorithm (think deli/bakery "now serving customer X" systems) does this.

The idea is that each process, when it wants to enter the critical section, takes a number. Whoever has the smallest number gets to go in. This is more complex than the bakery ticket-spitters because two processes may grab the same number (to guarantee that they wouldn't would require mutual exclusion – exactly the thing we're trying to implement), and because there is no attendant to call out the next number – the processes all must come to agreement on who should proceed next into the critical section.

We break ties by allowing the process with the lower process identifier (PID) to proceed. For P_i, we call it i. This assumes PIDs from 0 to $n-1$ for n processes, but this can be generalized.

Although two processes that try to pick a number at about the same time may get the same number, we do guarantee that once a process with number k is in, all processes choosing numbers will get a number $> k$.

Notation used below: an ordered pair `(number, pid)` fully identifies a process' number. We define a *lexicographic order* of these:

- $(a, b) < (c, d)$ is $a < c$ or if $a = c$ and $b < d$

The algorithm:

- Shared data, initialized to 0's and false

```
boolean choosing[n];
int number[n];
```

- Process P_i

```
while (1) {
  choosing[i]=true;
  number[i]=max(number[0],number[i],...,number[n-1])+1;
  choosing[i]=false;
  for (j=0; j<n; j++) {
    while (choosing[j]);
    while ((number[j]!=0) &&
           ((number[j],j) < (number[i],i)));
  }

    /* critical section */

  number[i]=0;

    /* non-critical section */

}
```

Before choosing a number, a process indicates that it is doing so. Then it looks at everyone else's number and picks a number one larger. Then it says it's done choosing.

Then look at every other process. First, wait for that process not to be choosing. Then make sure we are allowed to go before that process. Once we have successfully decided that it's safe to go before every other process, then go!

To leave the CS, just reset the number back to 0.

So great, we have a solution. But...problems:

1. That's a lot of code. Lots of while loops and for loops. Could be expensive if we're going to do this a lot.
2. If this is a highly popular critical section, the numbers might never reset, and we could overflow our integers. Unlikely, but think what could happen if we did.
3. It's kind of inconvenient and in some circumstances, unreasonable, to have these arrays of n values. There may not always be n processes, as some may come and go.

Synchronization hardware

Hardware support can make some of this a little easier. Problems can arise when a process is preempted within a single high-level language line. But we can't preempt in the middle of a machine instruction.

If we have a single machine instruction that checks the value of a variable and sets it **atomically**, we can use that to our advantage.

This is often called a **Test-and-Set** or `Test and Set Lock` instruction, and does this, atomically:

```
boolean TestAndSet(boolean *target) {
   boolean orig_val = *target;
   *target = TRUE;
   return orig_val;
}
```

So it sets the variable passed in to true, and tells us if it was true or false *before* we called it. So if two processes do this operation, both will set the value of `target` to true, but only one will get a return value of false.

This is the functionality we want, but how does the instruction actually work?

Really, this would be an instruction that takes two operands:

```
TAS R, X
```

Where R is a CPU register and X is a memory location. After the instruction completes (atomically), the value that was in memory at X is copied into R, and the value of X is set to 1. R contains a 0 if X previously contained a 0. If two processes do this operation concurrently, only one will actually get the 0.

The Intel x86 BTS instruction sets a single bit in memory and sets the carry bit of the status word to the previous value. This is equivalent.

Think about how you might implement an atomic test and set on, for example, a microprogrammed architecture.

Any of these can be used to implement a "test and set" function as described above.

Armed with this atomic test-and-set, we can make a simple mutual exclution solution for any number of processes, with just a single shared variable:

- Shared data

```
boolean lock = false;
```

- Process P_i

```
while (1) {
  while (TestAndSet(&lock));  /* busy wait */

    /* critical section */

  lock = false;

    /* non-critical section */
}
```

This satisfies mutual exclusion and progress, but not bounded waiting (a process can leave the CS and come back around and grab the lock again before others who may be waiting ever get a chance to look).

A solution that does satisfy bounded waiting is still fairly complicated:

- Shared data

```
boolean lock=false;
boolean waiting[n];  /* all initialized to false */
```

- Process P_i and its local data

```
int j;
boolean key;

while (1) {
  waiting[i]=true;
  key=true;
  while (waiting[i] && key)
    key = TestAndSet(&lock);
  waiting[i]=false;

    /* critical section */

  j=(i+1)%n;
  while ((j!=i) && !waiting[j])
    j=(j+1)%n;
  if (j==i) lock=false;
  else waiting[j]=false;

    /* non-critical section */

}
```

Another hardware instruction that might be available is the atomic *swap* operation:

```
void swap(boolean *a, boolean *b) {
  boolean temp = *a;
  *a = *b;
  *b = temp;
}
```

Different architectures have variations on this one, including the x86 `XCHG` instruction.

An algorithm to use this, minus the bounded wait again, is straightforward:

- Shared data

  ```
  boolean lock = false;
  ```

- Process P_i

  ```
  boolean key = true;
  while (1) {
    while (key == true) swap(&key,&lock);  /* busy wait */
  ```

```
        /* critical section */

      lock = false;

        /* non-critical section */
    }
```

It's pretty similar to what we saw before with `TestAndSet()`.

Semaphores

All that busy waiting in all of our algorithms for mutual exclusion is pretty annoying. It's just wasted time on the CPU. If we have just one CPU, it doesn't make sense for that process to take up its whole quantum spinning away waiting for a shared variable to change that can't change until the current process relinquishes the CPU!

This inspired the development of the *semaphore*. The name comes from old-style railroad traffic control signals (see `http://www.semaphores.com`), where mechanical arms swing down to block a train from a section of track that another train is currently using. When the track was free, the arm would swing up, and the waiting train could now proceed.

A semaphore `S` is basically an integer variable, with two atomic operations:

```
wait(S):
  while (S <= 0);  /* wait */
  S--;

signal(S):
  S++;
```

`wait` and `signal` are also often called `down` and `up` (from the railroad semaphore analogy) and occasionally are called `P` and `V` (because Dijkstra, who invented them, was Dutch, and these are the first letters of the Dutch words *proberen* (to test) and *verhogen* (to increment)).

Important!!! Processes using a semaphore in its most pure form are not allowed to set or examine its value. They can use the semaphore *only* through the `wait` and `signal` operations.

Note, however, that we don't want to do a busy-wait. A process that has to wait should be put to sleep, and should wake up only when a corresponding `signal` occurs, as that is the only time the process has any chance to proceed.

Semaphores are built using hardware support, or using software techniques such as the ones we discussed for critical section management.

Since the best approach is just to take the process out of the ready queue, some operating systems provide semaphores through system calls. We will examine their implementation in this context soon.

Given semaphores, we can create a much simpler solution to the critical section problem for n processes:

- Shared data

```
semaphore mutex=1;
```

- Process P_i

```
while (1) {
  wait(mutex);

    /* critical section */

  signal(mutex);

   /* non-critical section */
}
```

The semaphore provides the mutual exclusion for sure, and should satify progress, but depending on the implementation of semaphores, may or may not provide bounded waiting.

A semaphore implementation might look like this:

```
struct semaphore {
  int value;
  proclist L;
};
```

- *block* operation suspends the calling process, removes it from consideration by the scheduler
- *wakeup(P)* resumes execution of suspended process P, puts it back into consideration
- wait(S):

```
S.value--;
if (S.value < 0) {
  add this process to S.L;
  block;
}
```

- signal(S):

```
S.value++;
if (S.value <= 0) {
  remove a process P from S.L;
  wakeup(P);
}
```

There is a fairly standard implementation of semaphores on many Unix systems: POSIX semaphores

- create a shared variable of type `sem_t`
- initialize it with `sem_init(3)`
- wait operation is `sem_wait(3)`
- signal operation is `sem_post(3)`
- deallocate with `sem_destroy(3)`

Examples using POSIX semaphores and a semaphore-like construct called a *mutex* provided by pthreads to provide mutual exclusion in the bounded buffer problem:

See Example:
`/cluster/examples/prodcons-pthreads-counter-sem`

The pthreads library *mutex* is essentially a binary semaphore (only 0 and 1 are allowed). See `pthread_mutex_init(3)`.

Note that what we have been looking at are *counting semaphores*. This means that is the semaphore's value is 0 and there are two `signal` operations, its value will be 2. This means that the next two `wait` operations will not block.

This means that semaphores can be used in more general situations than simple mutual exclusion. Perhaps we have a section that we want at most 3 processes in at the same time. We can start with a semaphore initialized to 3.

Semaphores can also be used as a more general-purpose synchronization tool. Suppose statement B in process P_j can be executed only after statement A in P_i. We use a semaphore called `flag`, initialized to 0:

P_i	P_j
...	...
A;	`wait(flag);`
`signal(flag);`	B;
...	...

Here, P_j will be forced to wait only if it arrives at the `wait` call before P_i has executed the `signal`.

Of course, we can introduce *deadlocks* (two or more processes waiting indefinitely for an event that can only be caused by one of the waiting processes).

Consider semaphores Q and R, initialized to 1, and two processes that need to wait on both. A careless programmer could write:

P_0	P_1
`wait(Q);`	`wait(R);`
`wait(R);`	`wait(Q);`
...	...
`signal(R);`	`signal(Q);`
`signal(Q);`	`signal(R);`
...	...

Things might be fine, but they might not be.

There's also the possibility that a process might just forget a `signal` and leave one or more other processes (maybe even itself) waiting indefinitely.

We will look into the implementation of semaphores and mutexes later, but for now we will look more at how to use them as a synchronization tool.

Classical Problems of Synchronization

We will use semaphores to consider some synchronization problems. While some actual implementations provide the ability to "try to wait", or to examine the value of a semaphore's counter, we restict ourselves to initialization, `wait`, and `signal`.

Bounded buffer using semaphores

First, we revisit our friend the bounded buffer.

- Shared data:

```
semaphore fullslots, emptyslots, mutex;
full=0; empty=n; mutex=1;
```

- Producer process:

```
while (1) {
  produce item;
  wait(emptyslots);
```

```
    wait(mutex);
    add item to buffer;
    signal(mutex);
    signal(fullslots);
  }
```

- Consumer process:

```
  while (1) {
    wait(fullslots);
    wait(mutex);
    remove item from buffer;
    signal(mutex);
    signal(emptyslots);
    consume item;
  }
```

`mutex` provides mutual exclusion for the modification of the buffer (not shown in detail). The others make sure that the consumer doesn't try to remove from an empty buffer (`fullslots` is > 0) or that the producer doesn't try to add to a full buffer (`emptyslots` is > 0).

Dining Philsophers

- One way to tell a computer scientist from other people is to ask about the dining philosophers.
- Five philosophers alternate thinking and eating
- Need 2 forks to do so (eating spaghetti), one from each side
- keep both forks until done eating, then replace both

Since `fork` is the name of a C function, we'll use a different (and possibly more appropriate) analogy of chopsticks. The philosophers needs two chopsticks to eat rice.

- Shared data:

```
  semaphore chopstick[5];
  chopstick[0..4]=1;
```

- Philosopher i:

```
    while (1) {
      wait(chopstick[i]);
      wait(chopstick[(i+1)%5]);

       /* eat */

      signal(chopstick[i]);
      signal(chopstick[(i+1)%5]);

       /* think */

    }
```

This solution may deadlock.

One way to reduce the chances of deadlock might be to think first, since each might think for a different amount of time.

Another possibility:

Each philosopher

1. Picks up their left chopstick
2. Checks to see if the right chopstick is in use
3. If so, the philosopher puts down their left chopstick, and starts over at 1.
4. Otherwise, the philosopher eats.

Does this work?

No! It livelocks. Consider this: all could pick up their left chopstick, look right, put down the left, and repeat indefinitely.

How to solve this? Must either

1. introduce an asymmetry, or
2. limit the number of concurrently hungry philosophers to $n - 1$.

Here's one that includes an asymmetry, by having odd numbered philosophers pick up to the right first. The code for philosopher i and problem size n.

```
void philosopher() {
  think;
  if (odd(i)) {
    wait(chopstick[(i+1) % n]);
    wait(chopstick[i]);
  }
  else {
    wait(chopstick[i]);
    wait(chopstick[(i+1) % n]);
  }
  eat;
  if (odd(i)) {
    signal(chopstick[(i+1) % n]);
    signal(chopstick[i]);
  }
  else {
    signal(chopstick[i]);
    signal(chopstick[(i+1) % n]);
  }
}
```

Readers-Writers

We have a database and a number of processes that need access to it. We would like to maximize concurrency.

- There are multiple "reader processes" and multiple "writer processes".
- Readers see what's there, but don't change anything. Like a person on a travel web site seeing what flights have seats available.
- Writers change the database. The act of making the actual reservation.
- It's bad to have a writer in with any other writers or readers – may sell the same seat to a number of people (airline, sporting event, etc). Remember `counter++` and `counter--`!
- Multiple readers are safe, and in fact we want to allow as much concurrent access to readers as we can. Don't want to keep potential customers waiting.

A possible solution:

- Shared data:

  ```
  semaphore wrt, mutex;
  int readcount;
  wrt=1; mutex=1; readcount=0;
  ```

- Writer process:

```
while (1) {
  wait(wrt);

    /* perform writing */

  signal(wrt);
}
```

- Reader process:

```
while (1) {
  wait(mutex);
  readcount++;
  if (readcount == 1) wait(wrt);
  signal(mutex);

    /* perform reading */

  wait(mutex);
  readcount--;
  if (readcount == 0) signal(wrt);
  signal(mutex);
}
```

Note that the semaphore `mutex` protects `readcount` and is shared among readers only.

Semaphore `wrt` is indicates whether it is safe for a writer, or the first reader, to enter.

Danger: a reader may `wait(wrt)` while inside mutual exclusion of `mutex`. Is this OK?

This is a reader-preference solution. Writers can starve! This might not be good if the readers are "browsing customers" but the writers are "paying customers!"

Sleeping Barber

Problem: A certain barber shop has a single barber, a barber's chair, and n chairs in a waiting area. The barber spends his life in the shop. If there are no customers in the shop, the barber sleeps in the chair until a customer enters and wakes him. When customers are in the shop, the barber is giving one a haircut, and will call for the next customer when he finishes with each. Customers arrive periodically. If the shop is empty and the barber is asleep in the barber's chair, he wakes the barber and gets a haircut. If the barber is busy, but there are chairs available in the waiting room, he sleeps in one of those chairs until called. Finally, if there are no available chairs in the waiting room, the customer leaves and comes back another time.

A possible solution:

- Shared Data

```
constant CHAIRS = maximum number of chairs (including barber chair)
semaphore mutex=1,next_cust=0,barber_ready=0;
int cust_count=0;
```

- Customer process

```
while (1) {
  /* live your non barber-shop life until you decide you need
     a haircut */
  wait(mutex);
   if (cust_count>=CHAIRS) {
     signal(mutex);
     exit; /* leave the shop if full, try tomorrow */
   }
   cust_count++;  /* increment customer count */
   signal(mutex);
   signal(next_cust);  /* wake the barber if he’s sleeping */
   wait(barber_ready); /* wait in the waiting room */

     /* get haircut here */

   wait(mutex);
   cust_count--;  /* leave the shop, freeing a chair */
   signal(mutex);
}
```

- Barber process

```
while (1) {
  wait(next_cust); /* sleep until a customer shows up */
  signal(barber_ready);  /* tell the next customer you are ready */

  /* give the haircut here */

}
```

Semaphore Implementations

POSIX semaphores

We have seen examples using POSIX semaphores and pthread mutexes.

POSIX semaphores are implemented using pthreads mutexes:

`/usr/src/lib/libc_r/uthread/uthread_sem.c`

The `struct sem` definition is in `/usr/src/lib/libc_r/uthread/pthread_private.h`

This is an implementation of full counting semaphores using a construct that is much like a binary semaphore.

Note that if we determine that a `wait()` call must result in an actual wait (rather than just a decrement of the semaphore value) we need to put the thread to sleep and unlock the mutex.

We need something else to "sleep" on. All we have to work with are pthread functions here, not the actual kernel data structures that a lower-level implementation of semaphores might use.

pthreads provides a construct called a *condition variable*, with which a thread can be put to sleep with a call to `pthread_cond_wait()`.

This essentially provides a place for pthreads to sleep awaiting someone to "signal" them by calling `pthread_cond_signal()`.

Calling `pthread_cond_wait()` also releases the mutex so some other thread can have the chance to come in and do a `sem_post()` to let us continue at some point in the future.

The thread automatically reacquires the mutex lock when it is awakened.

We can see the calls to `pthread_cond_wait()` in `sem_wait()` and `pthread_cond_signal()` in `sem_post()`.

This implementation also provides a `sem_trywait()` and a `sem_getvalue()` which make these even more flexible than the semaphores we have been assuming for our synchronization problems.

Note: in FreeBSD, these are implemented as an extention to the pthreads library, and cannot be used for synchronization of independent processes. (see the comment in `sem_init()`)

Pthreads mutex/condition variables

But what about these pthread mutexes and condition variables? How are these achieved?

See `/usr/src/lib/libc_r/uthread/pthread_private.h`, `/usr/src/lib/libc_r/uthread/uthread_mutex.c` and `/usr/src/lib/libc_r/uthread/uthread_cond.c`.

First, note the structure definitions for `struct pthread_mutex` and `struct pthread_cond`.

The `pthread_mutex_init()` function initializes the structure. Not much of interest here.

The `pthread_mutex_lock()` just calls another internal routine, `mutex_lock_common()`.

Once we check errors and defer signals, we call

```
_SPINLOCK(&(*mutex)->lock);
```

As the name suggests, this is a spin lock. But aren't we supposed to be avoiding these? Let's see what this is all about. It is a macro. It actually ends up calling a function `_spinlock` defined in `/usr/src/lib/libc_r/uthread/uthread_spinlock.c`

While we cannot obtain the lock with an atomic test and set operation, we yield the CPU and try again. Note that the atomic test operation, defined in `/usr/src/lib/libc_r/arch/i386/_atomic_lock.S`, uses the `xchg` instruction on the x86. This instruction is an atomic swap of the value in a register with the values in a memory location. Note that each architecture has something implemented in assembly that will provide equivalent functionality.

OK, so back in the `mutex_lock_common()` function, we have exclusive access to the mutex data structure. We can now see if it's locked (if it has a non-NULL owner). If not, we take it. If so, we add ourself to the list of waiters. The `_thread_kern_sched_state_unlock()` function then unlocks the mutex, and we change our state to `PS_MUTEX_WAIT`, so we will not be scheduled until the mutex is unlocked.

When it comes time to unlock, we end up in the `mutex_unlock_common()` function. We basically get the lock on the mutex (spinlock). If there are threads waiting on the lock, we pick the next one, set its state to `PS_RUNNING` and do some other bookkeeping so it can run. If no thread was waiting, the assignment inside the if statement sets the owner to `NULL`, unlocking the mutex for the next lock attempt.

So what about the condition variable?

This is implemented in `/usr/src/lib/libc_r/uthread/uthread_cond.c`.

Initialization is straightforward - not much of interest here.

`pthread_cond_wait()` is where the interesting functionality lives. Once we see that the call is valid, the current thread is added to the condition variable's waiting queue, and given a "never wake up" status, set its state to `PS_COND_WAIT`.

`pthread_cond_signal()` selects a thread that is waiting on the condition variable and wakes it up by setting its state to `PS_RUNNING`.

SysV Semaphores

FreeBSD also implements System V semaphores.

SysV (along with BSD) was one of the two major "flavors" of Unix.

Most Unix systems were either BSD-based (SunOS up to 4.x, Ultrix) or SysV-based (Irix, Solaris).

Despite being (as you may have guessed) BSD-based, FreeBSD relies on a semaphore implementation from a long, long time ago from SysV.

These are described in great detail in Bach.

They are allocated in groups, stored in arrays.

See Example:
`/cluster/examples/prodcons-sysvsemaphore`

This example demonstrates the use of these semaphores to synchronize independent processes.

The buffer process creates and initializes the semaphores (and some shared memory). Semaphores created with `semget()`, we set their initial values by calling `semctl()` with the `SETVAL` command.

Producer and consumer processes attach to the existing shared memory and semaphores, then use the semaphores as we have seen in producer/consumer examples in class. Using `semop()` call.

We won't look at the implementation of these in as much detail as we did for the POSIX semaphores, but here are some highlights.

Implementation for FreeBSD is in `/usr/src/sys/kern/sysv_sem.c`

See `semop()` calling `msleep()` (Line 1052).

`msleep()` is a kernel call that puts a process to sleep until someone calls `wakeup()`.

These are defined in `/usr/src/sys/kern/kern_synch.c` and this is where PCBs are actually put into sleep queues and removed from ready/run queues.

Windows XP Semaphores

Windows XP provides both binary semaphores (mutexes) and counting semaphores.

Monitors

Semaphores are error-prone (oops, did I say `wait`? I meant `signal`!). You might never release a mutex, might run into unexpected orderings that lead to deadlock.

Monitors are a high-level language constuct intended to avoid some of these problems. A monitor is an abstract data type with shared variables and methods, similar to a C++ or Java class.

```
monitor example_mon {
  shared variables;

  procedure P1(...) {
    ...
  }

  procedure P2(...) {
    ...
  }

  ...

  initialization/constructor;
}
```

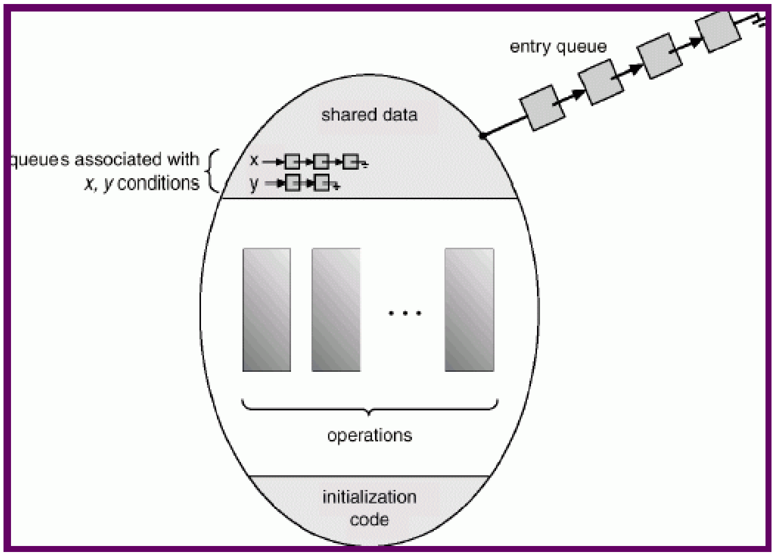
A monitor has a special property that at most one process can be actively executing in its methods at any time. Mutual exclusion everywhere inside the monitor!

But if only one process can be in, and a process needs to wait, no other processes can get in. So an additional feature of monitors is the *condition variable*. This is a shared variable that has semaphore-like operations `wait()` and `signal()`. When a process in a monitor has to wait on a condition variable, other processes are allowed in.

But when happens on a signal? If we just wake up the waiting process and let the signalling process continue, we have violated our monitor's rules and have two active processes in the monitor. Two possibilities:

- Force the signaler to leave immediately
- Force the signaler to wait until the awakened waiter leaves

There are waiting queues associated with each condition variable, and with the entire monitor for processes outside waiting for initial entry.



Note that monitors are a language construct. They can be implemented using OS-provided functionality such as semaphores.

Also note the similarity between these monitors and Java classes and methods that use the `synchronized` keyword.