COS 318: Operating Systems

Semaphores, Monitors and Condition Variables

Prof. Margaret Martonosi
Computer Science Department
Princeton University

http://www.cs.princeton.edu/courses/archive/fall11/cos318/

Today's Topics

- Semaphores
- Monitors
- Mesa-style monitors
- Programming idiom

Mutual Exclusion and Critical Sections

- A critical section is a piece of code in which a process or thread accesses a common (shared or global) resource.
- Mutual Exclusion algorithms are used to avoid the simultaneous use of a common resource, such as a global variable.
- In the buying milk example, what is the portion that requires mutual exclusion?

Conditions for a good Mutex solution:

- No two processes may be simultaneously inside their critical regions.
- No assumptions may be made about speeds or the number of CPUs.
- No process running outside its critical region may block other processes.
- No process should have to wait forever to enter its critical region.

The Big Picture

	OS codes and concurrent applications			
High-Level Atomic API	Mutex	Semaphores	Monitors	Send/Recv
Low-Level Atomic Ops	Load/store	Interrupt disable/enable	Test&Set	Other atomic instructions
	Interrupts (I/O, timer)	Multiprocessors		CPU scheduling

Semaphores (Dijkstra, 1965)

- Initialization
  - Initialize a value atomically
- P (or Down or Wait) definition
  - Atomic operation
  - Wait for semaphore to become positive and then decrement

```
P(s){
  while (s <= 0)
    ;
  s--;
}
```

Analogy: Think about semaphore value as the number of empty chairs at a table…

- V (or Up or Signal) definition
  - Atomic operation
  - Increment semaphore by 1

```
V(s){
  s++;
}
```

The atomicity and the waiting can be implemented by either busywaiting or blocking solutions.

An aside on Dijkstra…

- Quite a personality…Avoided owning a computer for several decades into his career…Won the 1972 Turing Award…
- Created a series of numbered memos with his thoughts on computing topics
  - Now Archived at U. Texas:
  - http://www.cs.utexas.edu/~EWD/
  - Example: “A Tutorial on the Split Binary Semaphore”
    - http://www.cs.utexas.edu/~EWD/ewd07xx/EWD703.PDF
  - Some are short proofs or papers, others are jokes or rants.
  - Go-to statement considered harmful: Published in CACM 1968, also as EWD215…

Semaphores can be used for…

- Binary semaphores can provide mutual exclusion (solution of critical section problem)
- Counting semaphores can represent a resource with multiple instances (e.g. solving producer/consumer problem)
- Signaling events (persistent events that stay relevant even if nobody listening right now)

Classic Synchronization Problems

- There are a number of "classic" problems that represent a class of synchronization situations
- Critical Section problem
- Producer/Consumer problem
- Reader/Writer problem
- 5 Dining Philosophers
- Why? Once you know the "generic" solutions, you can recognize other special cases in which to apply them (e.g., this is just a version of the reader/writer problem)

Producer / Consumer

Producer:

while(whatever)

{

locally generate item

fill empty buffer with item

}

Consumer:

while(whatever)

{

get item from full buffer

use item

}

Producer / Consumer (With Counting Semaphores)

Producer:

```
while(whatever)
{
locally generate item

    P(emptybuf);

fill empty buffer with item

    V(fullbuf);

}
```

Consumer:

```
while(whatever)
{
    P(fullbuf);

get item from full buffer

    V(emptybuf);

use item
}
```

Semaphores: emptybuf initially N; fullbuf initially 0;

Producer Consumer (Bounded Buffer) with Semaphores: More detail…

```
producer() {
  while (1) {
    produce an item
    P(emptyBuf);

    P(mutex);
    put the item in buffer
    V(mutex);

    V(fullBuf);
  }
}
```

```
consumer() {
  while (1) {
    P(fullBuf);

    P(mutex);
    take an item from buffer
    V(mutex);

    V(emptyBuf);
    consume the item
  }
}
```

- Init: emptyCount = N; fullCount = 0; mutex = 1
- Are **`P(mutex)`** and **`V(mutex)`** necessary?

Example: Interrupt Handler

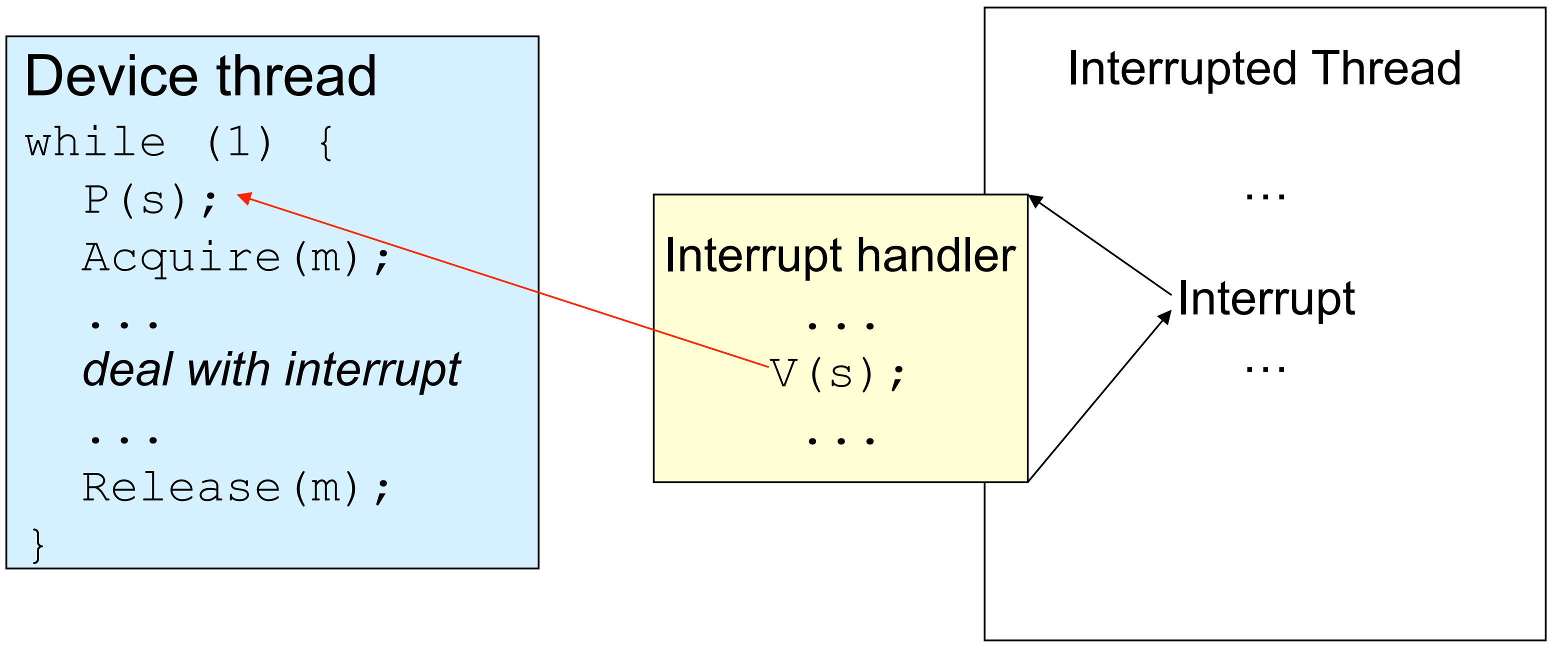
- A device thread works with an interrupt handler
- What to do with shared data?
- What if "m" is held by another thread or by itself?

Device thread	Interrupt handler

`...`
`Acquire(m);`
`...`
 	
`Release(m);`
`...`
	`...`
	`Acquire(m);`
	`...`
	
	`Release(m);`
	`...`
	?

Use Semaphore to Signal

```
Init(s,0);
```



Semaphores Are Not Always Convenient

- A shared queue has Enqueue and Dequeue:

```
Enqueue(q, item)
{
  Acquire(mutex);
  put item into q;
  Release(mutex);
}
```

```
Dequeue(q)
{
  Acquire(mutex);
  take an item from q;
  Release(mutex);
  return item;
}
```

- It is a consumer and producer problem
  - `Dequeue(q)` should block until `q` is not empty
- Semaphores are difficult to use: orders are important

Today's Topics

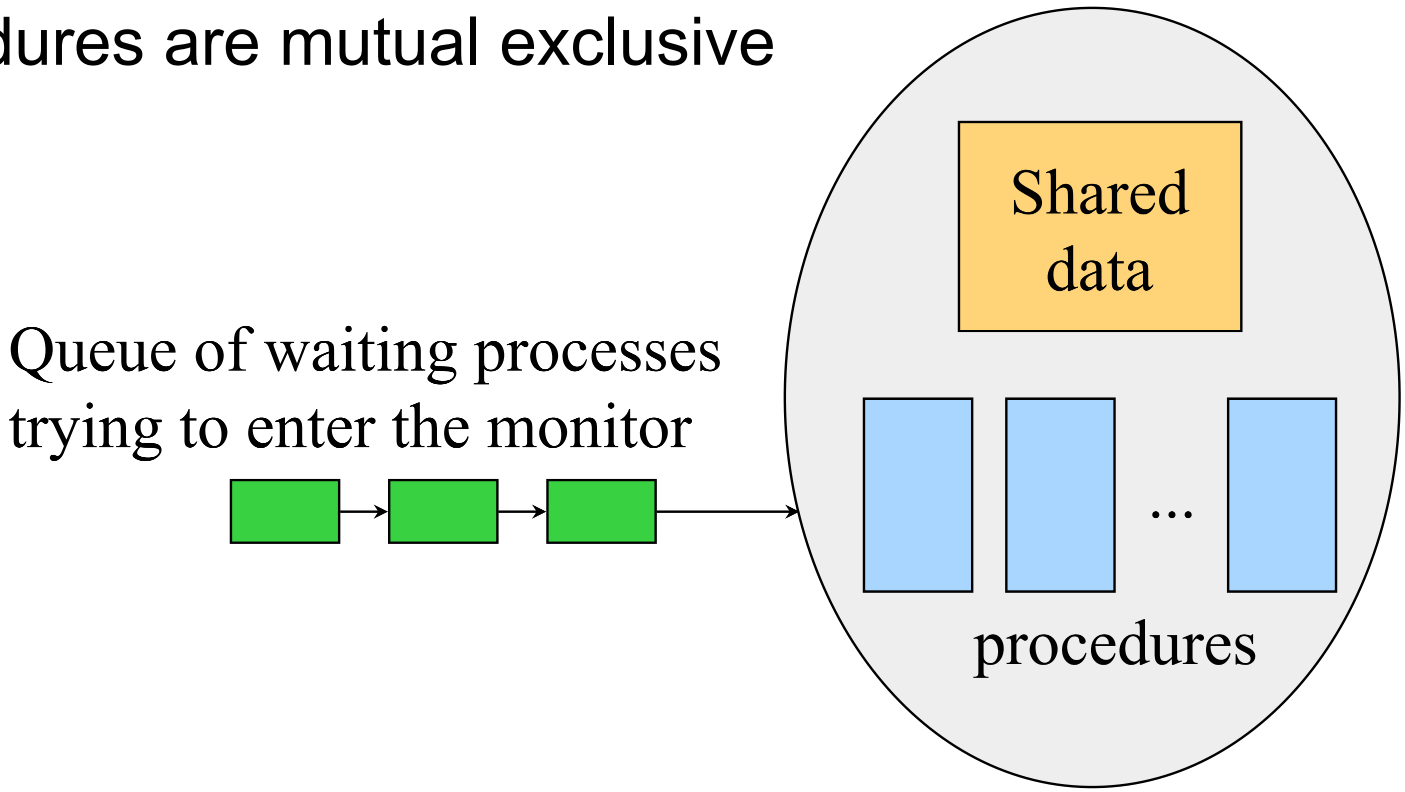
- Semaphores
- Monitors
- Mesa-style monitors
- Programming idiom
- Barriers

The Big Picture

	OS codes and concurrent applications			
High-Level Atomic API	Mutex	Semaphores	Monitors	Send/Recv
Low-Level Atomic Ops	Load/store	Interrupt disable/enable	Test&Set	Other atomic instructions
	Interrupts (I/O, timer)	Multiprocessors		CPU scheduling

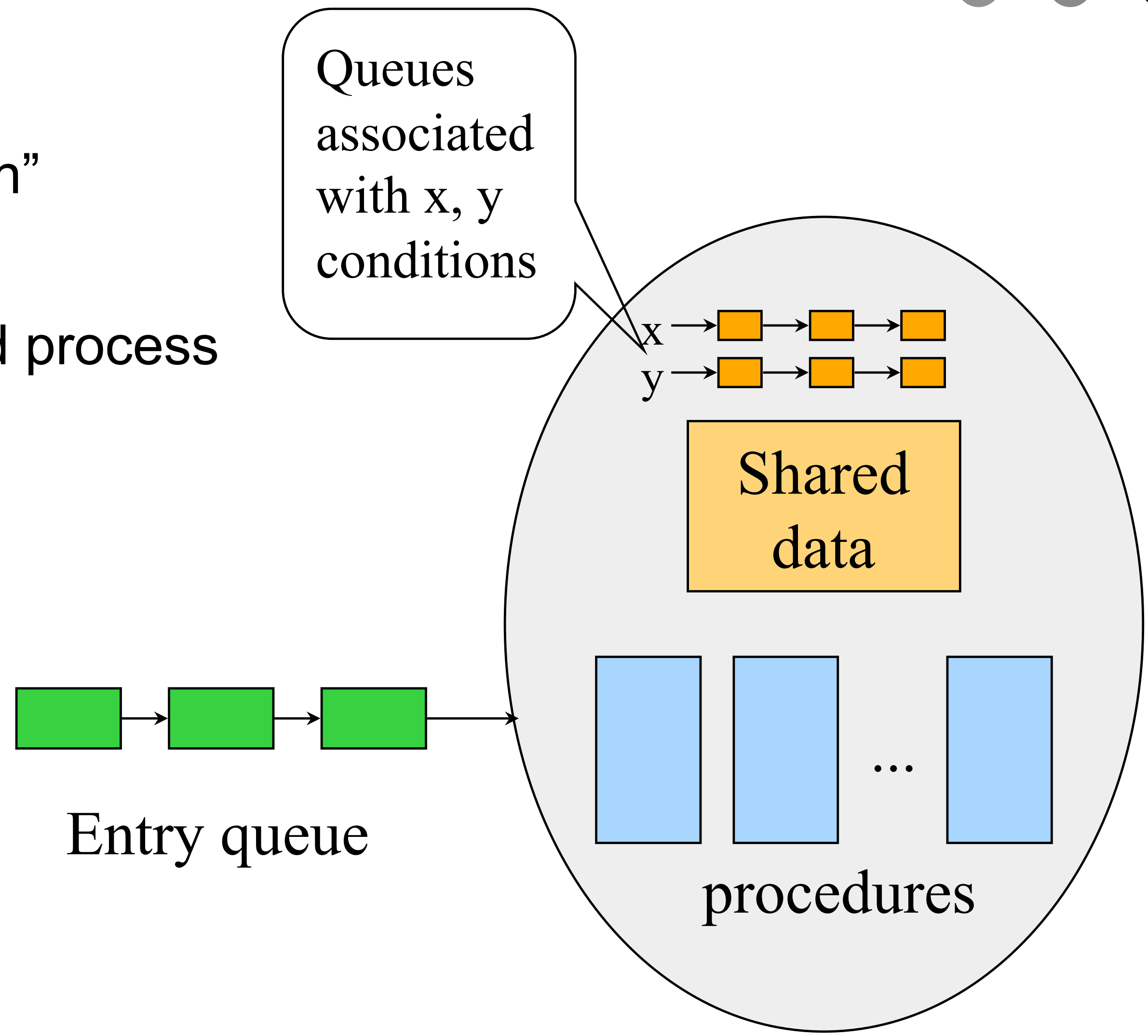
Monitor: Hide Mutual Exclusion

- Brinch-Hansen (73), Hoare (74)
- Procedures are mutual exclusive



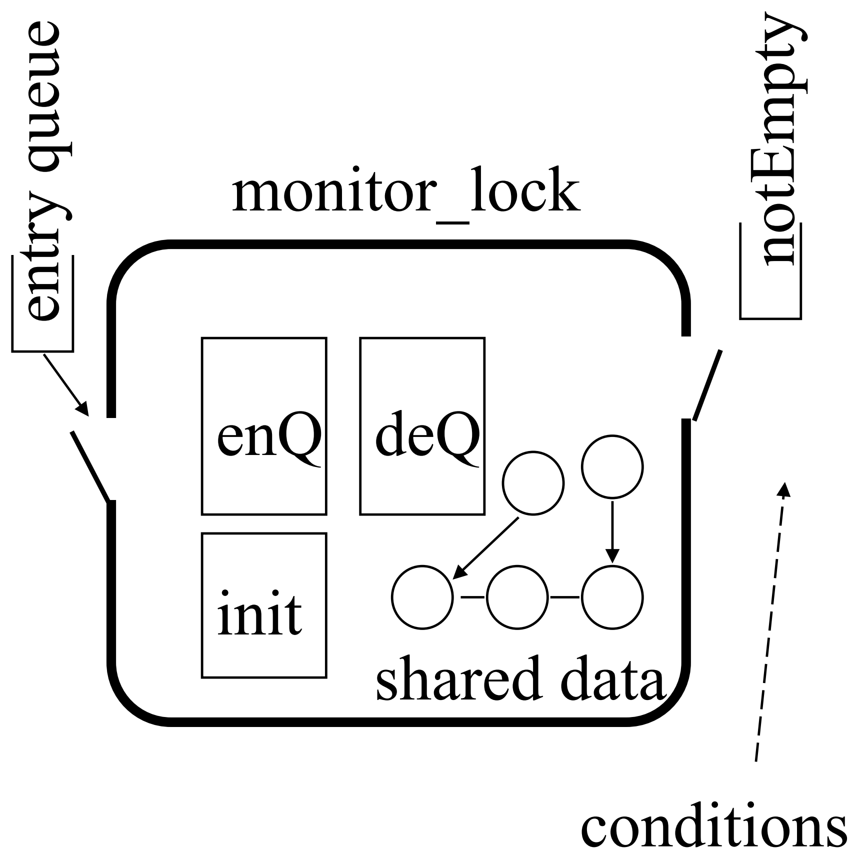
Condition Variables in A Monitor

- Wait(condition)
  - Block on "condition"
- Signal(condition)
  - Wakeup a blocked process on "condition"



Monitor Abstraction

- Encapsulates shared data and operations with mutual exclusive use of the object (an associated *lock*).
- Associated *Condition Variables* with operations of *Wait* and *Signal*.

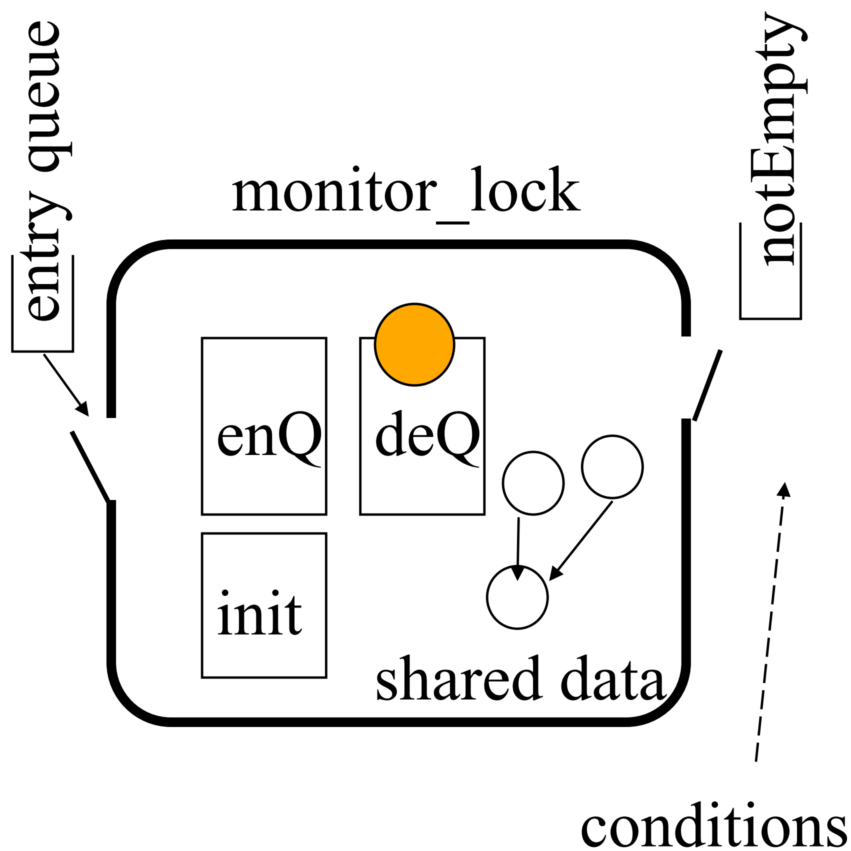


Condition Variables

- We build the monitor abstraction out of a lock (for the mutual exclusion) and a set of associated condition variables.
- *Wait on condition*: releases lock held by caller, caller goes to sleep on condition's queue.
  When awakened, it must reacquire lock.
- *Signal condition*: wakes up one waiting thread.
- *Broadcast*: wakes up all threads waiting on this condition.

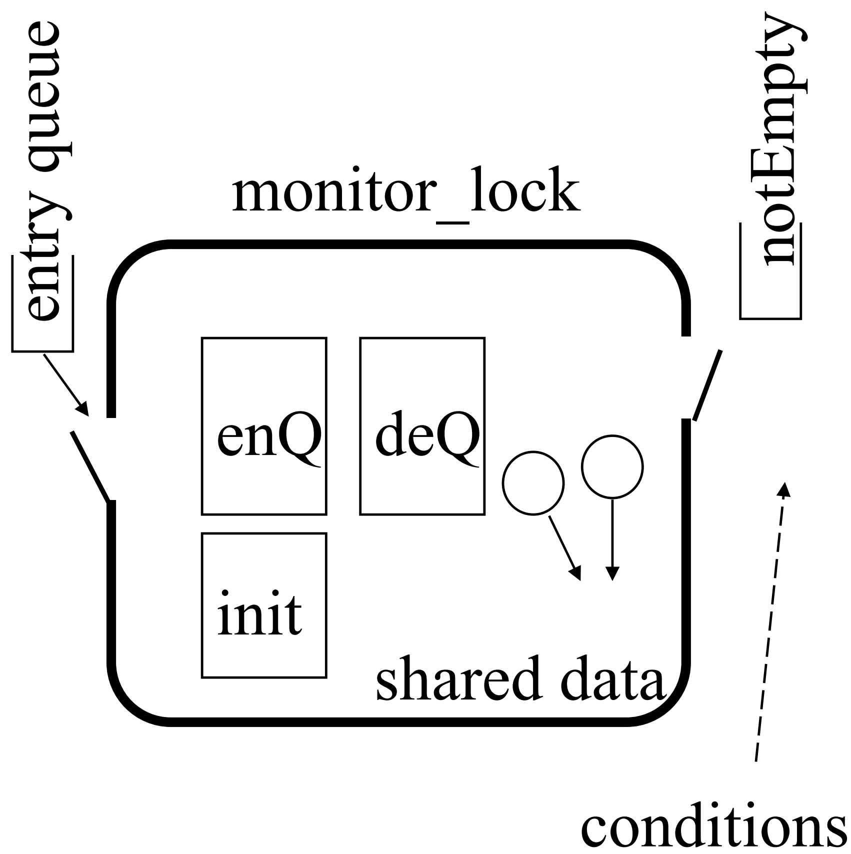
Monitor Abstraction

```
EnQ:{acquire (lock);
    if (head == null)
        {head = item;
        signal (lock, notEmpty);}
    else tail->next = item;
    tail = item;
    release(lock);}
deQ:{acquire (lock);
    if (head == null)
        wait (lock, notEmpty);
    item = head;
    if (tail == head) tail = null;
    head=item->next;
    release(lock);}
```



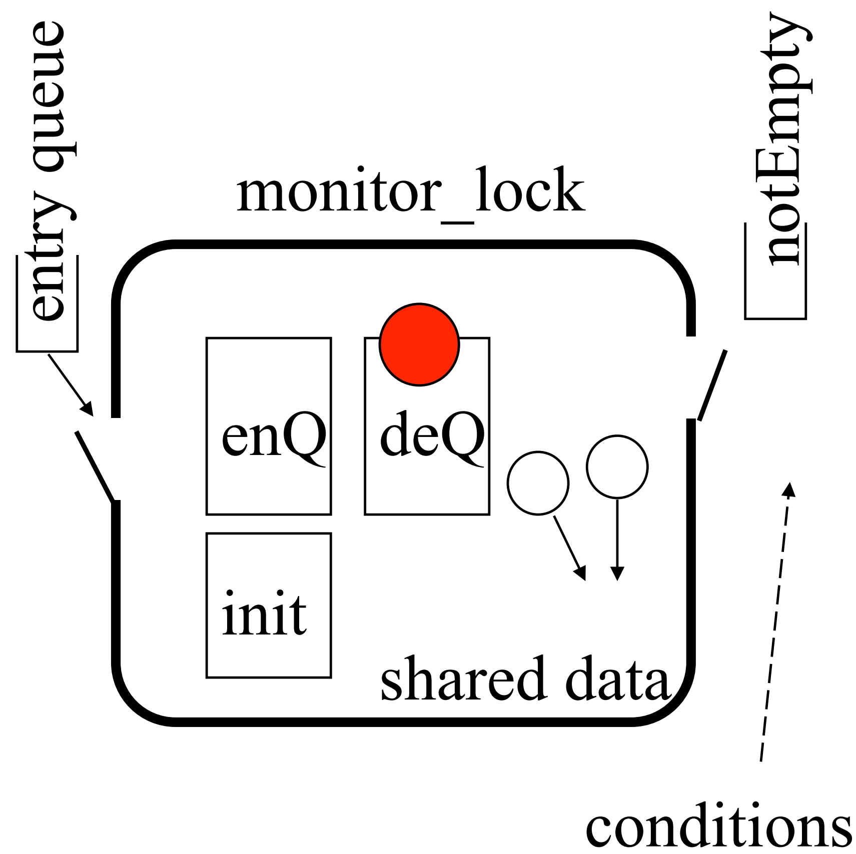
Monitor Abstraction

```
EnQ:{acquire (lock);
    if (head == null)
        {head = item;
        signal (lock, notEmpty);}
    else tail->next = item;
    tail = item;
    release(lock);}
deQ:{acquire (lock);
    if (head == null)
        wait (lock, notEmpty);
     item = head;
     if (tail == head) tail = null;
     head=item->next;
    release(lock);}
```



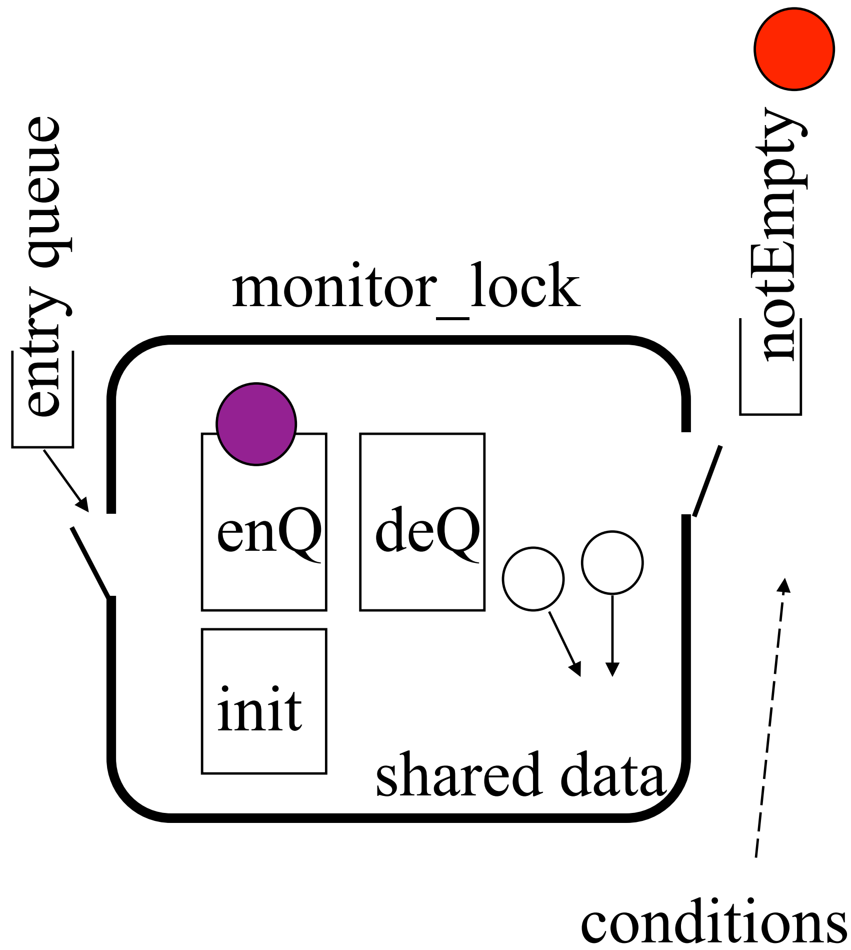
Monitor Abstraction

```
EnQ:{acquire (lock);
     if (head == null)
          {head = item;
          signal (lock, notEmpty);}
     else tail->next = item;
     tail = item;
     release(lock);}
deQ:{acquire (lock);
     if (head == null)
          wait (lock, notEmpty);
      item = head;
      if (tail == head) tail = null;
      head=item->next;
     release(lock);}
```



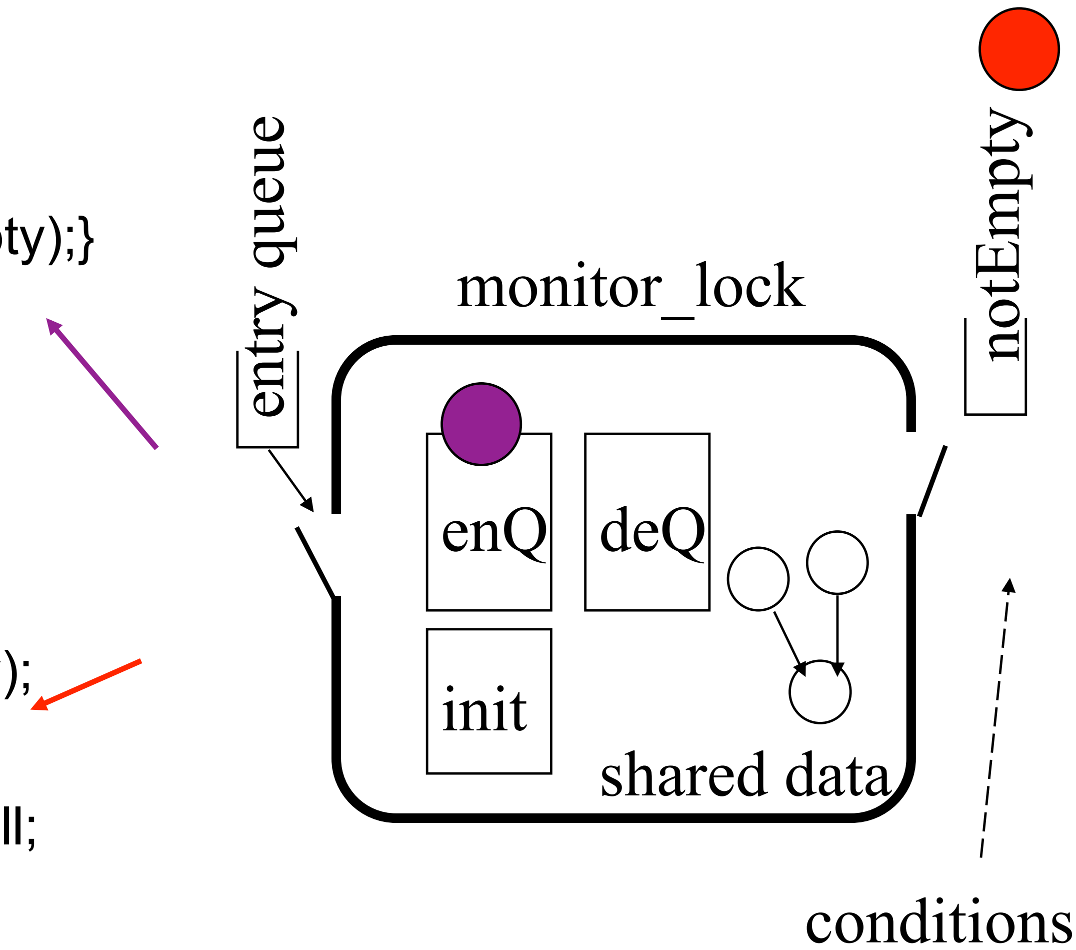
Monitor Abstraction

```
EnQ:{acquire (lock);
     if (head == null)
          {head = item;
          signal (lock, notEmpty);}
     else tail->next = item;
     tail = item;
     release(lock);}
deQ:{acquire (lock);
     if (head == null)
          wait (lock, notEmpty);
      item = head;
      if (tail == head) tail = null;
      head=item->next;
     release(lock);}
```



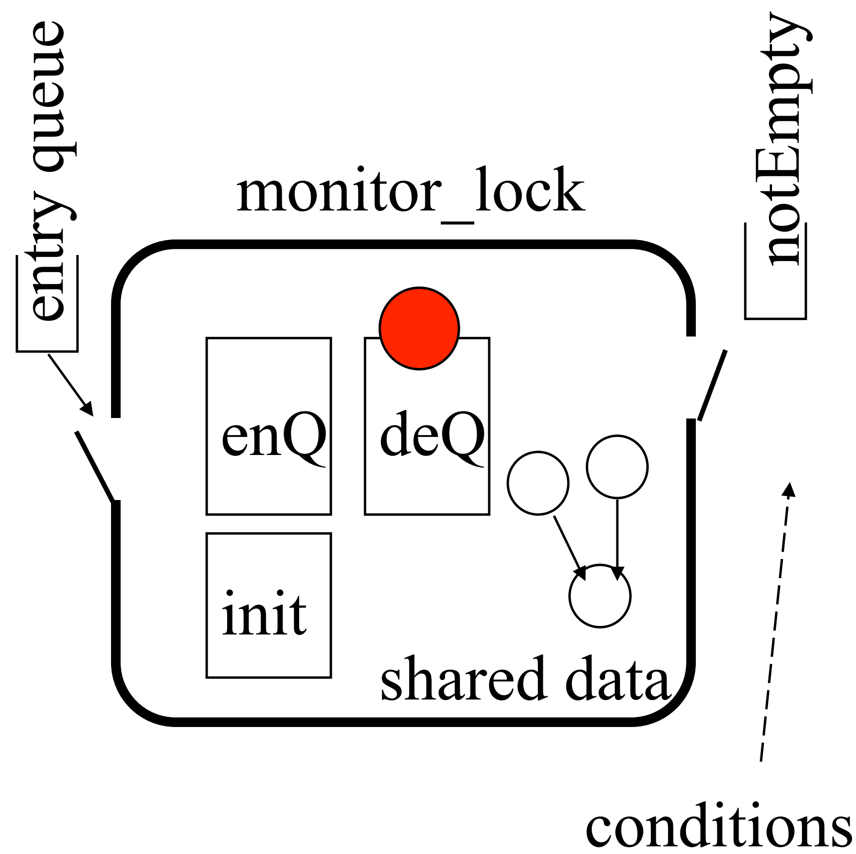
Monitor Abstraction

```
EnQ:{acquire (lock);
    if (head == null)
        {head = item;
        signal (lock, notEmpty);}
    else tail->next = item;
    tail = item;
    release(lock);}
deQ:{acquire (lock);
    if (head == null)
        wait (lock, notEmpty);
    item = head;
    if (tail == head) tail = null;
    head=item->next;
    release(lock);}
```



Monitor Abstraction

```
EnQ:{acquire (lock);
      if (head == null)
            {head = item;
            signal (lock, notEmpty);}
      else tail->next = item;
      tail = item;
      release(lock);}
deQ:{acquire (lock);
      while (head == null)
            wait (lock, notEmpty);
       item = head;
       if (tail == head) tail = null;
       head=item->next;
      release(lock);}
```



Producer-Consumer with Monitors

```
procedure Producer
begin
  while true do
  begin
    produce an item
    ProdCons.Enter();
  end;
end;

procedure Consumer
begin
  while true do
  begin
    ProdCons.Remove();
    consume an item;
  end;
end;
```

```
monitor ProdCons
  condition full, empty;

  procedure Enter;
  begin
    if (buffer is full)
      wait(full);
    put item into buffer;
    if (only one item)
      signal(empty);
  end;

  procedure Remove;
  begin
    if (buffer is empty)
      wait(empty);
    remove an item;
    if (buffer was full)
      signal(full);
  end;
```

Options of the Signaler

- Run the signaled thread immediately and suspend the current one (Hoare)
  - If the signaler has other work to do, life is complex
  - It is difficult to make sure there is nothing to do, because the signal implementation is not aware of how it is used
  - It is easy to prove things
- Exit the monitor (Hansen)
  - Signal must be the last statement of a monitor procedure
- Continues its execution (Mesa)
  - Easy to implement
  - But, the condition may not be true when the awaken process actually gets a chance to run

Today's Topics

- Semaphores
- Monitors
- Mesa-style monitors
- Programming idiom
- Barriers

Mesa Style "Monitor" (Birrell's Paper)

- Associate a condition variable with a mutex
- Wait(mutex, condition)
  - Atomically unlock the mutex and enqueued on the condition variable (block the thread)
  - Re-lock the lock when it is awakened
- Signal(condition)
  - No-op if there is no thread blocked on the condition variable
  - Wake up at least one if there are threads blocked
- Broadcast(condition)
  - Wake up all waiting threads
- Original Mesa paper
  - B. Lampson and D. Redell. Experience with processes and monitors in Mesa. *Comm. ACM* 23, 2 (feb 1980), pp 106-117.

Consumer-Producer with Mesa-Style Monitor

```
static count = 0;
static Cond full, empty;
static Mutex lock;

Enter(Item item) {
  Acquire(lock);
  if (count==N)
    Wait(lock, full);
  insert item into buffer
  count++;
  if (count==1)
    Signal(empty);
  Release(lock);
}
```

```
Remove(Item item) {
  Acquire(lock);
  if (!count)
    Wait(lock, empty);
  remove item from buffer
  count--;
  if (count==N-1)
    Signal(full);
  Release(lock);
}
```

Any issues with this?

Consumer-Producer with Mesa-Style Monitor

```
static count = 0;
static Cond full, empty;
static Mutex lock;

Enter(Item item) {
  Acquire(lock);
  while (count==N)
    Wait(lock, full);
  insert item into buffer
  count++;
  if (count==1)
    Signal(empty);
  Release(lock);
}

Remove(Item item) {
  Acquire(lock);
  while (!count)
    Wait(lock, empty);
  remove item from buffer
  count--;
  if (count==N-1)
    Signal(full);
  Release(lock);
}
```

Today’s Topics

- Semaphores
- Monitors
- Mesa-style monitors
- Programming idiom
- Barriers

The Programming Idiom

- Waiting for a resource

```
Acquire( mutex );
while ( no resource )
  wait( mutex, cond );
...
(use the resource)
...
Release( mutex);
```

- Make a resource available

```
Acquire( mutex );
...
(make resource available)
...
Signal( cond );
/* or Broadcast( cond );
Release( mutex);
```

Revisit the Motivation Example

```
Enqueue(Queue q,
        Item item) {

  Acquire(lock);

  insert an item to q;

  Signal(Empty);
  Release(lock);
}
```

```
Item GetItem(Queue q) {
  Item item;

  Acquire( lock );
  while ( q is empty )
    Wait( lock, Empty);

    remove an item;

  Release( lock );
  return item;
}
```

- Does this work?

Condition Variables Primitives

- Wait(mutex, cond)
  - Enter the critical section (min busy wait)
  - Release mutex
  - Save state to TCB, mark as blocked
  - Put my TCB on cond's queue
  - Exit the critical section
  - Call the scheduler
  - Waking up:
    - Acquire mutex
    - Resume

- Signal(cond)
  - Enter the critical section (min busy wait)
  - Wake up a TCB in cond's queue
  - Exit the critical section

More on Mesa-Style Monitor

- Signaler continues execution
- Waiters simply put on ready queue, with no special priority
  - Must reevaluate the condition
- No constraints on when the waiting thread/process must run after a "signal"
- Simple to introduce a broadcast: wake up all
- No constrains on signaler
  - Can execute after signal call (Hansen's cannot)
  - Do not need to relinquish control to awaken thread/process

Evolution of Monitors

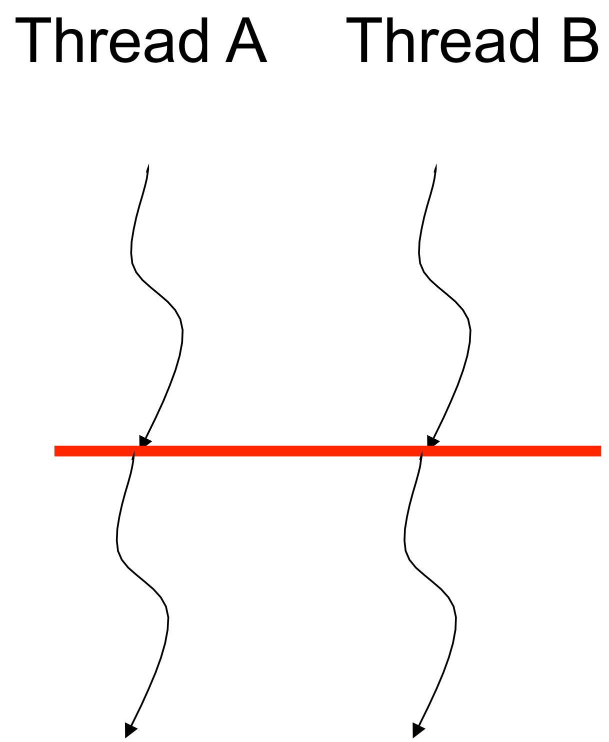
- Brinch-Hansen (73) and Hoare Monitor (74)
  - Concept, but no implementation
  - Requires Signal to be the last statement (Hansen)
  - Requires relinquishing CPU to signaler (Hoare)
- Mesa Language (77)
  - Monitor in language, but signaler keeps mutex and CPU
  - Waiter simply put on ready queue, with no special priority
- Modula-2+ (84) and Modula-3 (88)
  - Explicit LOCK primitive
  - Mesa-style monitor
- Pthreads (95)
  - Started standard effort around 1989
  - Defined by ANSI/IEEE POSIX 1003.1 Runtime library
- Java threads
  - Use 'synchronized' primitive for mutual exclusion
  - Wait() and notify() use implicit per-class condition variable

Today's Topics

- Semaphores
- Monitors
- Mesa-style monitors
- Programming idiom
- Barriers

Example: A Simple Barrier

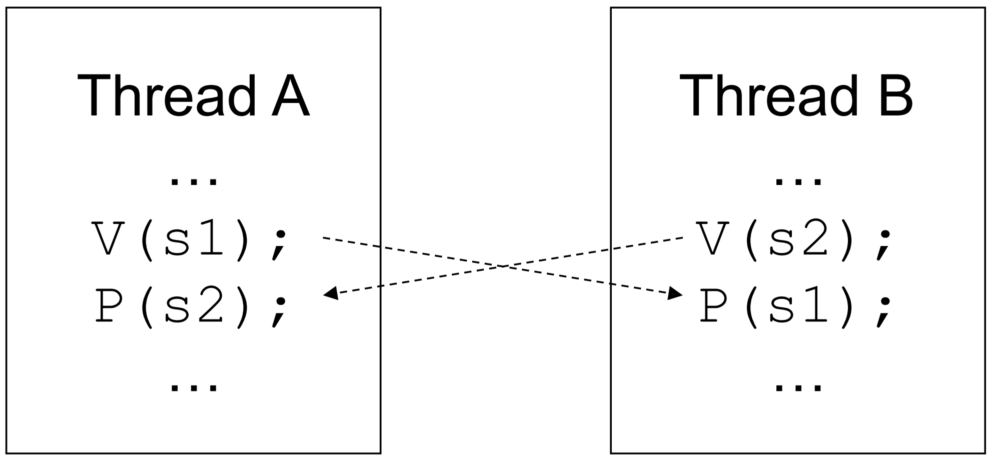
- Thread A and Thread B want to meet at a particular point and then go on
- How would you program this with a monitor?



Using Semaphores as A Barrier

- Use two semaphore?

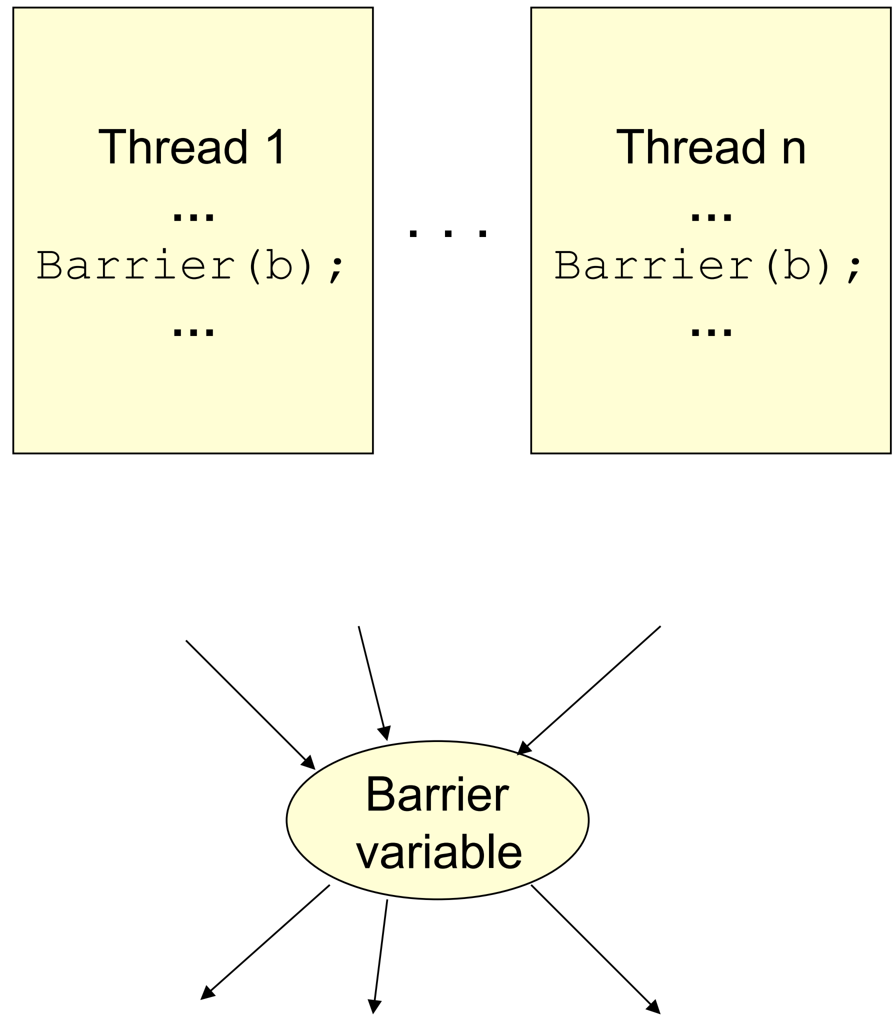
```
init(s1, 0);
init(s2, 0);
```



- What about more than two threads?

Barrier Primitive

- Functions
  - Take a barrier variable
  - Broadcast to n-1 threads
  - When barrier variable has reached n, go forward
- Hardware support on some parallel machines



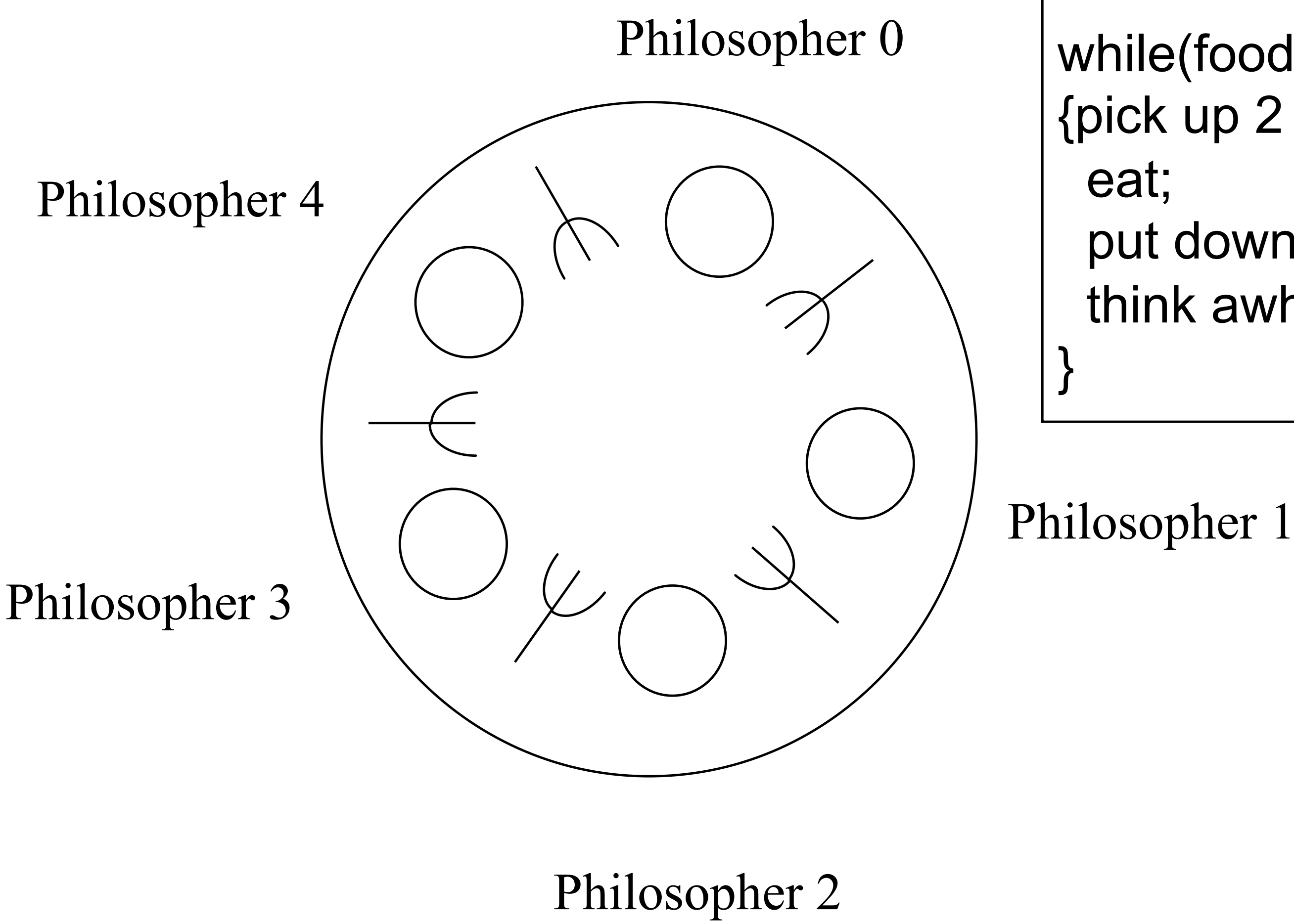
Equivalence

- Semaphores
  - Good for signaling
  - Not good for mutex because it is easy to introduce a bug
- Monitors
  - Good for scheduling and mutex
  - Maybe costly for a simple signaling

Summary

- Semaphores
- Monitors
- Mesa-style monitor and its idiom
- Barriers

5 Dining Philosophers




```
while(food available)
{pick up 2 adj. forks;
  eat;
  put down forks;
  think awhile;
}
```

Template for Philosopher

```
while (food available)
{                                   /*pick up forks*/

    eat;
                                    /*put down forks*/

    think awhile;
}
```

Naive Solution

```
while (food available)
{       P(fork[left(me)]);               /*pick up forks*/
        P(fork[right(me)]);
    eat;
        V(fork[left(me)]);               /*put down forks*/
        V(fork[right(me)]);
    think awhile;
}
```

Does this work?

Simplest Example of Deadlock

Thread 0	Interleaving	Thread 1
P(R1)		P(R2)
P(R2)	P(R1)	P(R1)
V(R1)	P(R2)	V(R2)
V(R2)	P(R1) waits	V(R1)
	P(R2) waits	

R1 and R2 initially 1 (binary semaphore)

Conditions for Deadlock

- Mutually exclusive use of resources
  - Binary semaphores R1 and R2
- Circular waiting
  - Thread 0 waits for Thread 1 to V(R2) and
    Thread 1 waits for Thread 0 to V(R1)
- Hold and wait
  - Holding either R1 or R2 while waiting on other
- No pre-emption
  - Neither R1 nor R2 are removed from their respective holding Threads.

Philosophy 101 (or why 5DP is interesting)

- How to eat with your Fellows without causing ***Deadlock.***
  - Circular arguments (the circular wait condition)
  - Not giving up on firmly held things (no preemption)
  - Infinite patience with Half-baked schemes (hold some & wait for more)
- Why ***Starvation*** exists and what we can do about it.

Dealing with Deadlock

It can be ***prevented*** by breaking one of the prerequisite conditions:

- Mutually exclusive use of resources
  - Example: Allowing shared access to read-only files (readers/writers problem)
- circular waiting
  - Example: Define an ***ordering*** on resources and acquire them in order
- hold and wait
- no pre-emption

Circular Wait Condition

```
while (food available)
{  if (me == 0) {P(fork[left(me)]); P(fork[right(me)]);}
   else {(P(fork[right(me)]); P(fork[left(me)]); }
   eat;
   V(fork[left(me)]); V(fork[right(me)]);

   think awhile;
}
```

Hold and Wait Condition

```
while (food available)
{  P(mutex);
   while (forks [me] != 2)
         {blocking[me] = true; V(mutex); P(sleepy[me]); P(mutex);}
   forks [leftneighbor(me)] --;  forks [rightneighbor(me)]--;
   V(mutex):
   eat;
   P(mutex); forks [leftneighbor(me)] ++;  forks [rightneighbor(me)]++;
   if (blocking[leftneighbor(me)]) {blocking [leftneighbor(me)] = false; V
   (sleepy[leftneighbor(me)]); }
   if (blocking[rightneighbor(me)]) {blocking[rightneighbor(me)] = false; V
   (sleepy[rightneighbor(me)]); }     V(mutex);
   think awhile;
}
```

Starvation

The difference between deadlock and starvation is subtle:

- Once a set of processes are deadlocked, there is no future execution sequence that can get them out of it.
- In starvation, there does exist some execution sequence that is favorable to the starving process although there is no guarantee it will ever occur.
- Rollback and Retry solutions are prone to starvation.
- Continuous arrival of higher priority processes is another common starvation situation.

5DP - Monitor Style

```
Boolean eating [5];
Lock forkMutex;
Condition forksAvail;

void PickupForks (int i) {
    forkMutex.Acquire( );
    while ( eating[(i-1)%5] || eating [(i+1)%5] )
        forksAvail.Wait(&forkMutex);
    eating[i] = true;
    forkMutex.Release( );
}

void PutdownForks (int i) {
    forkMutex.Acquire( );
    eating[i] = false;
    forksAvail.Broadcast (&forkMutex);
    forkMutex.Release( );
}
```

What about this?

```
while (food available)
{  forkMutex.Acquire( );
   while (forks [me] != 2) {blocking[me]=true;
          forkMutex.Release( ); sleep( ); forkMutex.Acquire( );}
   forks [leftneighbor(me)]--;  forks [rightneighbor(me)]--;
    forkMutex.Release( ):
   eat;
    forkMutex.Acquire( );
   forks[leftneighbor(me)] ++;  forks [rightneighbor(me)]++;
   if (blocking[leftneighbor(me)] || blocking[rightneighbor(me)])
          wakeup ( );  forkMutex.Release( );
   think awhile;
}
```

Classic Synchronization Problems

- There are a number of “classic” problems that represent a class of synchronization situations
- Critical Section problem
- Producer/Consumer problem
- Reader/Writer problem
- 5 Dining Philosophers
- Why? Once you know the “generic” solutions, you can recognize other special cases in which to apply them (e.g., this is just a version of the reader/writer problem)

Readers/Writers Problem

Synchronizing access to a file or data record in a database such that any number of threads requesting read-only access are allowed but only one thread requesting write access is allowed, excluding all readers.