Topic 12: Monitors

- Semaphores are powerful, but difficult to use correctly.
  - Single solution for mutual exclusion and scheduling.
  - Programmer must mind P's and V's.
- Mailboxes are (also) powerful.
  - Add the ability to send data.
  - Can be more difficult to implement.
- *Monitors* are a high-level language construct combining three features:
  - Shared data.
  - Mutually exclusive operations on the data.
    - Implicitly (automatically) provided by the monitor.
  - Scheduling.
- Monitors are especially convenient for synchronization involving lots of state (many variables).
- All procedures in a monitor are protected by a *monitor lock*, which automatically provides mutual exclusion.

- Few programming languages actually support monitors. Mesa from Xerox was used to build a real operating system (Pilots).
  - When using threads with C: pthreads library provides monitor locks.
  - And then, Java appeared…
- Using C-like pseudo-code, a queue manipulation monitor might look like:

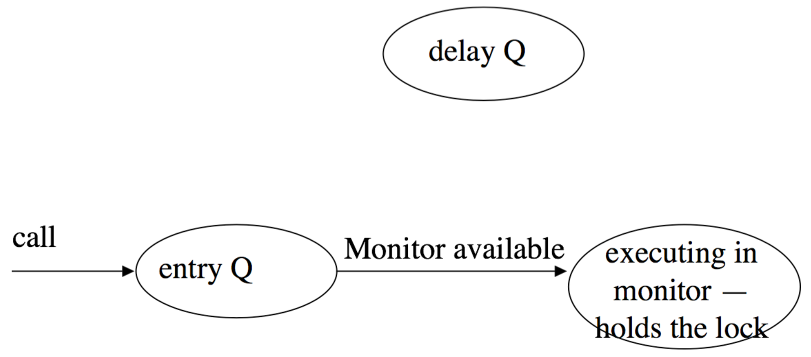
```
monitor QueueHandler;
   struct Queue queue;

   void AddToQueue( int val )
   {
      … add val to end of queue …
   } /* AddToQueue */

   int Remove From Queue()
   {
      … remove value from queue, return it …
   } /* RemoveFromQueue */

endmonitor;
```

- Monitors are a higher-level concept than P and V, or send and receive.
  - Monitors are both easier and safer to use.
- Mutual exclusion is implicit. In the above, the process can add **`val`** to the end of the queue without worrying if another process is also adding to or removing from the queue.

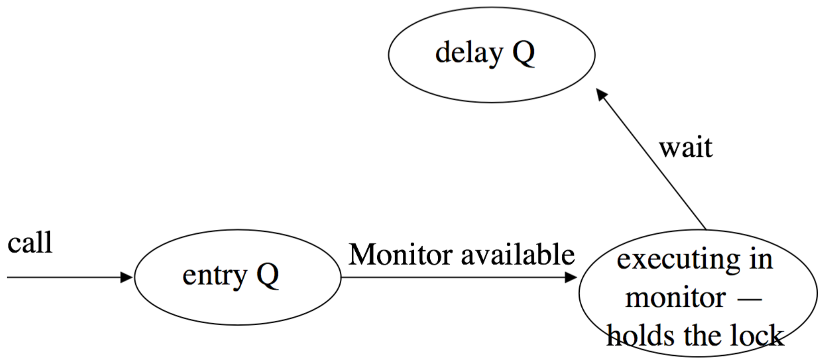
- Monitors need more facilities than just mutual exclusion. Need some way to wait.
  - Busy-wait inside monitor?
  - Put process to sleep inside monitor?

- Solution:
  - *Condition variable*: something to wait on.
    - All operations on a condition variable must occur <u>inside</u> its monitor.
    - **`wait( condition )`** : release monitor lock, put process to sleep. When process wakes up again, re-acquire monitor lock before proceeding
    - **`signal( condition )`** : wake up <u>one</u> process waiting on the condition variable (FIFO).
      - If nobody is waiting, do nothing (*no history*).
      - Java calls this **`notify`**
    - **`broadcast( condition )`** : wake up all processes waiting on the condition variable.
      - If nobody is waiting, do nothing.
      - Java calls this **`notifyAll`**

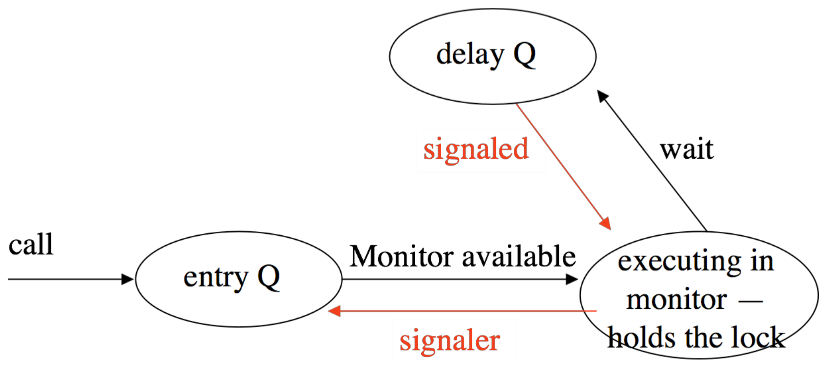
- Wait/Signal mechanism:
  - Basic idea:
    - There is an entry Queue — need to gain access to the monitor routines.
    - Once inside, there is only one process executing in the monitor.
      - Can call any of the monitor routines.
      - Can access/change any of the monitor variables.



- Wait/Signal mechanism:
  - When a process must wait on a condition variable:
    - Where does the process wait?
      - Can not wait inside the monitor.
        - No other process can enter.
      - Wait's on a delay queue:

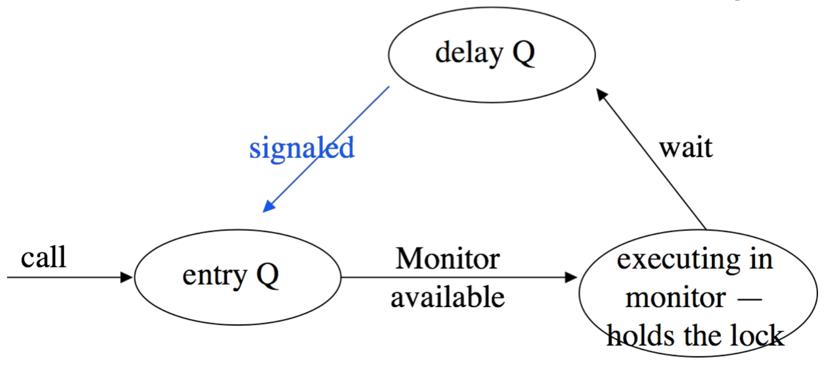


- Wait/Signal mechanism:
  - What happens when the condition variable is signaled?
  - Hoare semantics: awakened (signaled) process gets monitor lock immediately.
    - Process doing the signal gets “thrown out” temporarily.
      - Goes back on the Entry Queue
    - Probably need to signal as the last thing in the monitor (Hansen).



Also known as "Signal-and-Wait" because the signaler has to "wait" before continuing.

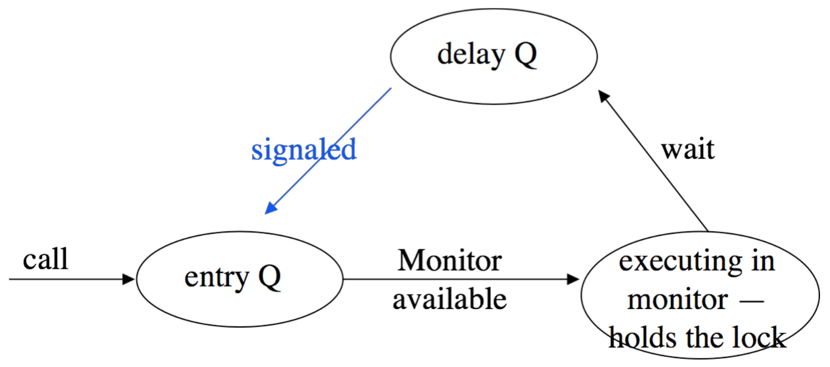
- Wait/Signal mechanism:
  - What happens when the condition variable is signaled?
  - Mesa semantics: signaler keeps monitor lock.
    - Awakened process waits for monitor lock with no special priority (a new process could get in before it).
    - This means that the event it was waiting for could have come and gone: must check again (use a loop) and be prepared to wait again if someone else took it.
    - Signal and broadcast are therefore hints rather than guarantees.



Also known as "Signal-and-Continue" because the signaler continues executing.

Mesa: Programming language developed at Xerox's Palo Alto Research Center (PARC).

- Wait/Signal mechanism:
  - We will use Mesa semantics in this course
    - It does make a difference, so be careful!
    - Mesa semantics are easier to understand (I claim!).
    - Java uses Mesa semantics.



- Queue manipulation: What happens when the queue has a finite size?

```
monitor QueueHandler;
struct Queue queue;
condition fullAvail, emptyAvail;

void AddToQueue( int val ) {
   while ( queue is full ) {
      wait( emptyAvail );
   }
   . . . add val to the end of the queue . . .

   signal( fullAvail );

} /* AddToQueue */

int RemoveFromQueue() {
   while( queue is empty ) {
      wait( fullAvail );
   }
   . . . remove value from queue . . .
   signal( emptyAvail );
   return value;
} /* RemoveFromQueue */
endmonitor;
```

- How do wait and signal compare to P and V?

Example: Bounded Buffer Producer-Consumer with Monitors:

```
monitor Bounded Buffer;
     buffer buffer[N];
     int front = 1, rear = 1, count = 0;
     condition empty, full;
```

```
deposit( item ) {
   while ( count == N )
      wait( full );
   // Guaranteed empty slot is now available
   buffer[rear] = item;
   rear = (rear + 1) % N;
   count++;
   signal( empty );
} /* deposit */
```

```
fetch( item ) {
   while ( count == 0 )
      wait( empty );
   // Guaranteed full slot is now available
   item = buffer[front];
   front = (front + 1) % N;
   count--;
   signal( full );
} /* fetch */
```

Example: Readers/Writers Problem with Monitors:

- The synchronization operations get encapsulated in monitored procedures: **`beginRead`**, **`beginWrite`**, **`doneRead`**, and **`doneWrite`**.

```
Reader process:
beginRead();
... read value(s) from the data base ...
doneRead();
```

```
Writer process:
    beginWrite():
    ... write value(s) to the data base ...
    doneWrite();
```

```
int ActiveWriters = 0, WaitingWriters = 0, ActiveReaders = 0, WaitingReaders = 0;
```

Example: Readers/Writers Problem with Monitors:

- The synchronization operations get encapsulated in monitored procedures: **beginRead**, **beginWrite**, **doneRead**, and **doneWrite**.

```
condition OKToRead, OKToWrite;
int ActiveWriters = 0, WaitingWriters = 0, ActiveReaders = 0, WaitingReaders = 0;
```

```
beginRead() {
   while ( (ActiveWriters > 0) ||
           (WaitingWriters > 0) ) {
      WaitingReaders++;
      wait( OKToRead );
      WaitingReaders--;
   }
   ActiveReaders++;
} /* beginRead */

doneRead() {
   ActiveReaders--;
   if ( ( ActiveReaders == 0 ) &&
        ( WaitingWriters > 0 ) {
      signal( OKToWrite );
   }
} /* doneRead */
```

```
beginWrite() {
   while ( ( ActiveWriters > 0 ) ||
           ( ActiveReaders > 0) ) {
      WaitingWriters++;
      wait( OKToWrite );
      WaitingWriters--;
   }
   ActiveWriters++;
} /* beginWrite */

doneWrite() {
   ActiveWriters--;
   if ( WaitingWriters > 0 ) {
      signal( OKToWrite );
   else if ( WaitingReaders > 0 ) {
      broadcast( OKToRead );
   }
} /* doneWrite */
```

a.) Reader 2 arrives
b.) Writer 1 arrives OKToWrite
c.) Writer 2 arrives OKToWrite
d.) Reader 3 arrives OKToRead
e.) Reader 4 arrives OKToRead
f.) Reader 1 leaves the database
g.) Writer 3 arrives OkToWrite

- Why are **`while`** loops needed?

- Could all of the **`signal's`** be **`broadcast's`**?

- Is **`WaitingReaders`** needed? Is **`WaitingWriters`** needed?

- Result of this monitor example: have used one synchronization primitive (monitors) to build a more powerful, higher-level synchronization primitive.

Implementing Monitors with Semaphores (Mesa semantics):

- Procedure entry: `P( mutex );`
- Procedure exit: `V( mutex );`

```
typedef struct {
   Semaphore waiting;
   int count;
} Condition;

wait( Condition *cond ) {
   cond->count++;
   V( mutex );
   P( cond->waiting );
   P( mutex );
} /* wait */

signal( Condition *cond ) {
   if ( cond->count > 0 ) {
      V( cond->waiting );
      cond->count--;
   }
} /* signal */
```

- Why is `count` needed?

- Java and Monitors:
  - Used to synchronize threads.
  - Useful methods:
    - **`wait()`**
    - **`notify()`   `notify_all()`**
  - **`synchronized`** keyword: locks the object.
  - Can be applied to a section of code (a critical section):

```
// somewhere within a method:
synchronized ( expression ) {
   . . . various Java code here . . .
}
```

- Can apply **synchronized** to methods: (code example from Concurrent Programming in Java by Doug Lea, 1997)

```
public final class CountingSemaphore {

   private int count_  = 0;

   public CountingSemaphore(int initialCount) {
      count_ = initialCount;
   } /* CountingSemaphore constructor */

   public synchronized void P() {
      while (count_ <= 0)
         try {  wait();  }  catch (InterruptedException ex) {}
      --count_;
   } /* P */

   public synchronized void V() {
      ++count_;
      notify();
   } /* V */

} /* CountingSemaphore class */
```

- Note that use of **synchronized** locks the object, not just the method. No other **synchronized** code within the object can be executed at the same time.
  - This is how Java solves the implicit exclusion that monitors require.
  - Note: a method within the object that is not **synchronized** can be executed at the same time!

NARROW BRIDGE
ALTERNATING
ONE WAY
TRAFFIC ONLY
WEIGHT LIMIT
15
TONS

Example (former exam question): Single-lane Bridge:

- You have been hired by the highway department to synchronize traffic over a narrow light-duty bridge on a public highway. Traffic may only cross the bridge in one direction at a time, and if there are ever more than 3 vehicles on the bridge at one time, it will collapse under their weight. In this system, each car is represented by one process that executes the procedure **`OneVehicle`** when it arrives at the bridge:

```
OneVehicle( int direction )
{
   ArriveBridge( direction );
   CrossBridge( direction );
   ExitBridge( direction );
} /* OneVehicle */
```

- **`direction`** is either 0 or 1; it gives the direction in which the vehicle will cross the bridge.
- a.) Write a monitor supplying the procedures **`ArriveBridge`** and **`ExitBridge`** (you do not need to write the **`CrossBridge`** procedure). **`ArriveBridge`** must not return until it is safe for the car to cross the bridge in the given direction (it must guarantee that there will be no head-on collisions or bridge collapses). **`ExitBridge`** is called to indicate that the caller has finished crossing the bridge; **`ExitBridge`** should take steps to let additional cars cross the bridge. You may use whatever local variables and conditions you need inside the monitor (but be sure to declare them and explain what they are used for). This is a lightly-travelled rural bridge, so you do not need to guarantee fairness or freedom from starvation.
- b.) In your solution, if a car arrives while traffic is currently moving in its direction of travel across the bridge, but there is another car already waiting to cross in the opposite direction, will the new arrival cross before the car waiting on the other side, after the car on the other side, or is it impossible to say? Explain briefly.

- Constraints:
  - Traffic may only flow in one direction at a time.
  - Only 3 vehicles can be on the bridge at one time.

```
Monitor Bridge;
condition waitForDirection[2];  // wait  your turn
int       numCrossing = 0;      // number on the bridge
int       currentDirection = 0; // current direction of traffic flow
```

```
ArriveBridge( int direction )
{
   if ( numCrossing == 0 ) {
      currentDirection = direction;
   }
   /* While we can not cross, keep waiting */
   while ( currentDirection != direction ||
           numCrossing >= 3 ) {
      wait( waitForDirection[direction] );
   }
   numCrossing++;
} /* ArriveBridge */
```

```
ExitBridge( int direction )
{
   numCrossing--;
   if ( numCrossing == 0 ) {
      /* if you are the last car on the
       * bridge, then change the direction so
       * that waiting cars can go. */
      currentDirection = 1 - direction;
   }
   /* Let another car go */
   signal(waitForDirection[currentDirecton]);
}  /* ExitBridge */
endmonitor;
```

- b.) It depends on which car gets the monitor lock first, as that car will cross the bridge first.

- <u>Summary</u>:
  - Semaphores use a single mechanism for both exclusion and schedulling; monitors use different mechanisms for each.
  - Monitors enforce a style of programming where complex synchronization code does not get mixed with the other code; it is separated and put in monitors.
  - Monitors are not present in very many languages, but still useful.
    - Can simulate in languages like C (by using the pthreads library).
  - In Java: **`wait`**, **`notify`**, **`notifyAll`**, **`synchronized`**.