CS 4410
Operating Systems

Synchronization
Monitors

Summer 2011

Cornell University

Today

- What are the monitors and how can we use them?
- Problems with semaphores
- Monitor Semantics/Structure
- Solving synchronization problems with monitors
- Mapping to Real Languages

Semaphors: Common programming errors

Process i	Process j	Process k	Process m
P(S)	V(S)	P(S)	P(S)
CS	CS	CS	if(something or other)
P(S)	V(S)		return;
			CS
			V(S)

Revisiting semaphores!

- **Semaphores** are very “low-level” primitives
  - Users could easily make small **errors**
  - Similar to programming in assembly language
  - Small error brings system to grinding halt
  - Very **difficult to debug**

- Also, we seem to be using them in two ways
  - For **mutual exclusion**, the “real” abstraction is a critical section
  - But the bounded buffer example illustrates something different, where threads “**communicate**” using semaphores

- Simplification: Provide concurrency support in compiler
  - **Monitors**

Monitors

- Hoare 1974
- Abstract Data Type for handling/defining shared resources
- Comprises:
  - Shared Private Data
    - The resource
    - Cannot be accessed from outside
  - Procedures that operate on the data
    - Gateway to the resource
    - Can only act on data local to the monitor
  - Synchronization primitives
    - Among threads that access the procedures

Monitor Semantics

- Monitors guarantee mutual exclusion
  - Only one thread can execute monitor procedure at any time
    - “in the monitor”
  - If second thread invokes monitor procedure at that time
    - It will block and wait for entry to the monitor
    - Need for a wait queue
- If thread within a monitor blocks, another can enter

Structure of a Monitor

```
Monitor monitor_name
{
    // shared variable declarations

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

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

    initialization_code(. . . .) {
        . . . .
    }
}
```

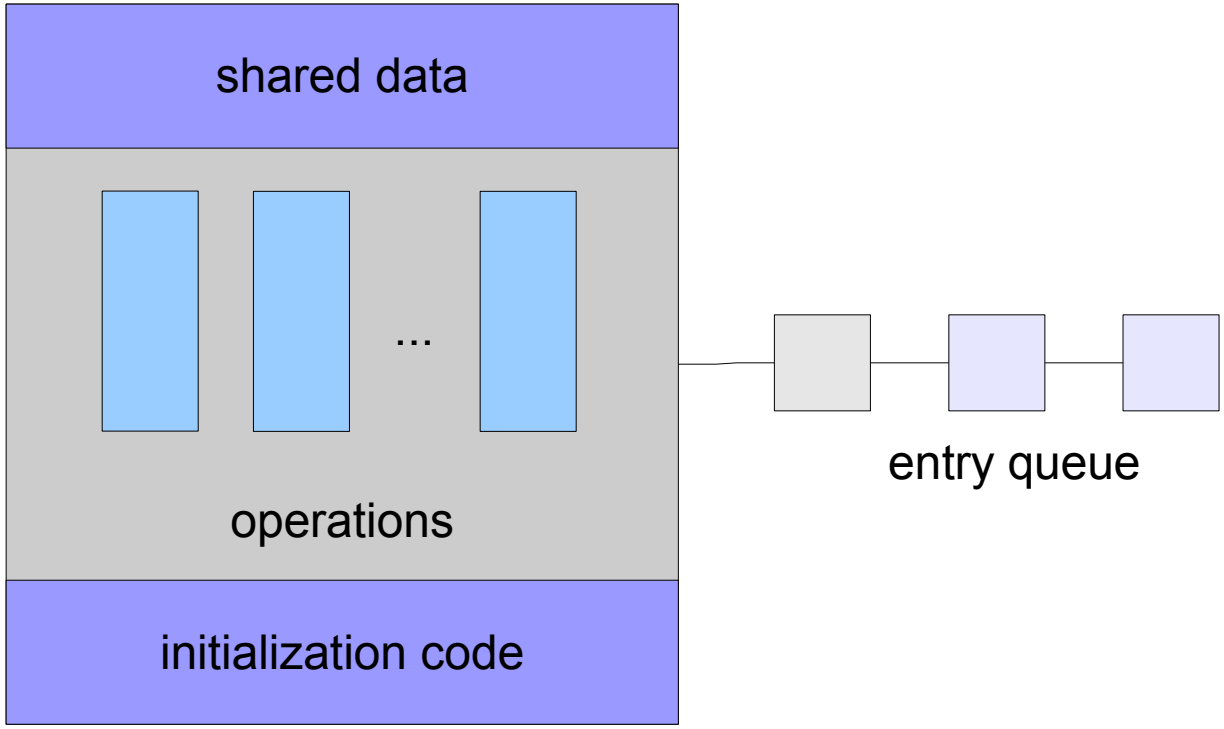
For example:

```
Monitor stack
{
        int top;
    void push(any_t *) {
        . . . .
    }

    any_t * pop() {
        . . . .
    }

    initialization_code() {
        . . . .
    }
}
```

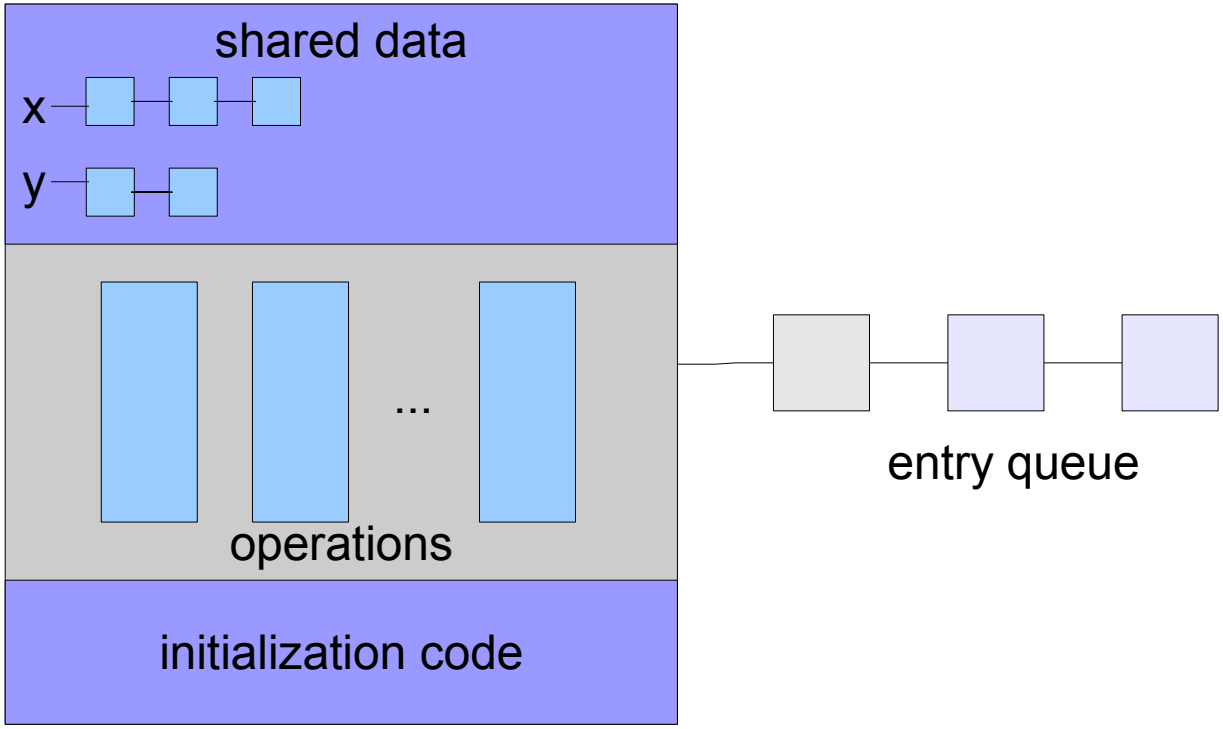
Structure of a Monitor



Synchronization Using Monitors

- Defines Condition Variables:
  - condition x;
  - Provides a mechanism to wait for events
    - Resources available, any writers
- 3 atomic operations on *Condition Variables*
  - x.wait(): release monitor lock, sleep until woken up
    - Condition variables have a waiting queue
  - x.notify(): wake one process waiting on condition (if there is one)
    - No history associated with signal
    - Notify and wait
    - Notify and continue
  - x.notifyAll(): wake all processes waiting on condition
    - Useful for resource manager

Synchronization Using Monitors



Types of wait queues

- Monitors have two kinds of “wait” queues
  - Entry to the monitor: has a queue of threads waiting to obtain mutual exclusion so they can enter
  - Condition variables: each condition variable has a queue of threads waiting on the associated condition

A Simple Monitor

```
Monitor EventTracker {
   int numburgers = 0;
   condition hungrycustomer;

   void customerenter() {
        if (numburgers == 0)
            hungrycustomer.wait()
        numburgers -= 1
   }
   void produceburger() {
       ++numburger;
       hungrycustomer.signal();
   }
}
```

- Because condition variables lack state, all state must be kept in the monitor
- The condition for which the threads are waiting is necessarily made explicit in the code
  - Numburgers > 0
- **Signal and wait** vs. **Signal and continue** semantics
- What happens if there are lots of customers?

A Simple Monitor

```
Monitor EventTracker {
   int numburgers = 0;
   condition hungrycustomer;

   void customerenter() {
       while (numburgers == 0)
          hungrycustomer.wait()
       numburgers -= 1
   }
   void produceburger() {
         ++numburger;
        hungrycustomer.signal();
   }
}
```

- Because condition variables lack state, all state must be kept in the monitor
- The condition for which the threads are waiting is necessarily made explicit in the code
  - Numburgers > 0
- **Signal and wait** vs. **Signal and continue** semantics
- What happens if there are lots of customers?

Producer Consumer using Monitors

```
Monitor Producer_Consumer {
   any_t buf[N];
   int n = 0, tail = 0, head = 0;
   condition not_empty, not_full;

   void put(char ch) {
     if(n == N)
          wait(not_full);
     buf[head%N] = ch;
     head++;
     n++;
     signal(not_empty);
    }

   char get()  {
        if(n == 0)
             wait(not_empty);
        ch = buf[tail%N];
        tail++;
        n--;
        signal(not_full);
        return ch;
      }

}
```

Readers and Writers

```
Monitor ReadersNWriters {

 int WaitingWriters,
WaitingReaders,NReaders, Nwriters;
  Condition CanRead, CanWrite;

 Void BeginWrite()  {





 }

 Void EndWrite()
 {



 }
```

```
Void BeginRead(){

}

Void EndRead(){

}

}
```

Readers and Writers

```
Monitor ReadersNWriters {

  int WaitingWriters,
WaitingReaders,NReaders, NWriters;
  Condition CanRead, CanWrite;

 Void BeginWrite()
  {




        NWriters = 1;
  }

  Void EndWrite()
  {
        NWriters = 0;



  }
```

```
Void BeginRead(){



      ++NReaders;

                          }


Void EndRead() {

      --NReaders

                          }

}
```

Readers and Writers

```
Monitor ReadersNWriters {

  int WaitingWriters,
WaitingReaders,NReaders, NWriters;
  Condition CanRead, CanWrite;

 Void BeginWrite(){

      NWriters = 1;}

  Void EndWrite() {
       NWriters = 0;
       if(WaitingReaders)
           Signal(CanRead);
       else
           Signal(CanWrite); }
```

```
Void BeginRead() {

      ++NReaders;

                    }

 Void EndRead(){

      if(--NReaders == 0)

            Signal(CanWrite); }

}
```

Readers and Writers

```
Monitor ReadersNWriters {

  int WaitingWriters,
WaitingReaders,NReaders, NWriters;
  Condition CanRead, CanWrite;

 Void BeginWrite(){
       if(NWriters == 1 || NReaders > 0)
       {
            ++WaitingWriters;
           wait(CanWrite);
           --WaitingWriters;
       }
       NWriters = 1; }

  Void EndWrite(){
        NWriters = 0;
        if(WaitingReaders)
             Signal(CanRead);
        else
             Signal(CanWrite);}

 Void BeginRead(){

       if(NWriters == 1 || WaitingWriters > 0)

       {

            ++WaitingReaders;

            Wait(CanRead);

            --WaitingReaders;

       }

       ++NReaders;

       Signal(CanRead);}


  Void EndRead(){

        if(--NReaders == 0)

                Signal(CanWrite); }

}
```

Understanding the Solution

- A writer can enter if there are no other active writers and no readers are waiting

Readers and Writers

```
Monitor ReadersNWriters {

  int WaitingWriters,
WaitingReaders,NReaders, NWriters;
  Condition CanRead, CanWrite;

 Void BeginWrite(){
      if(NWriters == 1 || NReaders > 0)
      {
           ++WaitingWriters;
          wait(CanWrite);
          --WaitingWriters;
      }
      NWriters = 1; }

  Void EndWrite(){
       NWriters = 0;
       if(WaitingReaders)
            Signal(CanRead);
       else
            Signal(CanWrite); }

 Void BeginRead(){

      if(NWriters == 1 || WaitingWriters > 0)

      {

           ++WaitingReaders;

           Wait(CanRead);

           --WaitingReaders;

      }

      ++NReaders;

      Signal(CanRead);}


  Void EndRead(){

       if(--NReaders == 0)

                Signal(CanWrite);}

}
```

Understanding the Solution

- A reader can enter if
  - There are no writers active or waiting
- So we can have many readers active all at once
- Otherwise, a reader waits (maybe many do)

Readers and Writers

```
Monitor ReadersNWriters {

  int WaitingWriters,
WaitingReaders,NReaders, NWriters;
  Condition CanRead, CanWrite;

 Void BeginWrite(){
       if(NWriters == 1 || NReaders > 0)
       {
            ++WaitingWriters;
           wait(CanWrite);
           --WaitingWriters;
       }
       NWriters = 1;}

  Void EndWrite() {
        NWriters = 0;
        if(WaitingReaders)
             Signal(CanRead);
        else
             Signal(CanWrite);}

 Void BeginRead(){

      if(NWriters == 1 || WaitingWriters > 0)

      {

           ++WaitingReaders;

           Wait(CanRead);

           --WaitingReaders;

      }

      ++NReaders;

      Signal(CanRead); }


  Void EndRead(){

       if(--NReaders == 0)

              Signal(CanWrite); }

}
```

Understanding the Solution

- When a writer finishes, it checks to see if any readers are waiting
  - If so, it lets one of them enter
  - That one will let the next one enter, etc…
- Similarly, when a reader finishes, if it was the last reader, it lets a writer in (if any is there)
- Do not forget: For **Signal and Continue semantics** we have '**while'** instead of **'if'** at BeginWrite and BeginRead!

Readers and Writers

```
Monitor ReadersNWriters {

  int WaitingWriters,
WaitingReaders,NReaders, NWriters;
  Condition CanRead, CanWrite;

 Void BeginWrite(){
       while(NWriters == 1 || NReaders > 0)
       {
            ++WaitingWriters;
           wait(CanWrite);
           --WaitingWriters;
       }
       NWriters = 1;}

  Void EndWrite(){
        NWriters = 0;
        if(WaitingReaders)
            Signal(CanRead);
        else
            Signal(CanWrite);}

 Void BeginRead(){

       while(NWriters == 1 || WaitingWriters > 0)

       {

            ++WaitingReaders;

            Wait(CanRead);

            --WaitingReaders;

       }

       ++NReaders;

       Signal(CanRead);}


  Void EndRead() {

       if(--NReaders == 0)

            Signal(CanWrite);}

}
```

Understanding the Solution

- It wants to be fair
  - If a writer is waiting, readers queue up
  - If a reader (or another writer) is active or waiting, writers queue up
  - … this is mostly fair, although once it lets a reader in, it lets ALL waiting readers in all at once, even if some showed up “after” other waiting writers

Mapping to Real Languages

```
Monitor ReadersNWriters {
 int x;
 Condition foo

 Void func()
 {
      while(x == 0)
      {
          foo.wait()
      }
      x = 1
 }
```

```
Class ReadersNWriters:
      def __init__(self):
            self.lock = Lock()
            self.foo = Condition(self.lock)

      def  func():
            with self.lock:
                  while x == 0:
                        self.foo.wait()
                  x = 1
```

Condition Variables & Semaphores

- Condition Variables != semaphores
- Access to monitor is controlled by a lock
  - Wait: blocks thread and gives up the monitor lock
    - To call wait, thread has to be in monitor, hence the lock
    - Semaphore P() blocks thread only if value less than 0
  - Signal: causes waiting thread to wake up
    - If there is no waiting thread, the signal is lost
    - V() increments value, so future threads need not wait on P()
    - Condition variables have no history!

However they can be used to implement each other

Today

- What are the monitors and how can we use them?
- Problems with semaphores
- Monitor Semantics/Structure
- Solving synchronization problems with monitors
- Mapping to Real Languages