COMP 530
Introduction to Operating Systems

Higher-Level Synchronization Primitives

Kevin Jeffay
Department of Computer Science
University of North Carolina at Chapel Hill
jeffay@cs.unc.edu
September 25, 2013

http://www.cs.unc.edu/~jeffay/courses/comp530

Lecture 7: Higher-Level Synch Primitives
Outline and key concepts

- The problem(s) with semaphores
- "Hoare" monitors
  - » Condition variables
- A disciplined use of synchronization primitives
- Implementing monitors
- "Mesa" monitors
  - » The priority inversion problem
- Readers/Writers synchronization
- Readings:
  - » Chapter 6 (Process Synchronization)

Higher-Level Synchronization Primitives

The *multiple-producer/multiple-consumer* problem

- Recall our *producer/consumer* solution
  - » What changes if there are multiple producers & consumers?

```
globals
  fullBuffers  : semaphore := 0     buf   : array [0..n-1] of char
  emptyBuffers : semaphore := n     nextIn,nextOut : 0..n-1 := 0
```

```
process Producer
begin
  loop
    <produce a character "c">

    emptyBuffers.down()

    buf[nextIn] := c
    nextIn := nextIn+1 mod n

    fullBuffers.up()
  end loop
end Producer
```

```
process Consumer
begin
  loop
    fullBuffers.down()

    data := buf[nextOut]
    nextOut := nextOut+1 mod n

    emptyBuffers.up()

    <consume "data">
  end loop
end Consumer
```

©2013 by Kevin Jeffay 3

Higher-Level Synchronization Primitives

The problem with semaphores

- Too general: we have one primitive for both *mutual exclusion* and *condition synchronization*
  - » The relationship between mutual exclusion and synchronization is often blurred or unclear

```
process Producer
begin
  loop
    <produce a character "c">

    emptyBuffers.down()
    mutex.down_b()
    buf[nextIn] := c
    nextIn := nextIn+1 mod n
    mutex.up_b()
    fullBuffers.up()
  end loop
end Producer
```

```
process Consumer
begin
  loop
    fullBuffers.down()
    mutex.down_b()
    data := buf[nextOut]
    nextOut := nextOut+1 mod n
    mutex.up_b()
    emptyBuffers.up()

    <consume "data">
  end loop
end Consumer
```

©2013 by Kevin Jeffay 4

Higher-Level Synchronization Primitives

Hoare Monitors

- Collect related shared objects together into a module
- Define data operations
  - » Calls to monitor entries guaranteed to be mutually exclusive

```
monitor : BoundedBuffer
  var buffer          : …
      nextIn,nextOut  : …

  entry deposit(c : char)
  entry remove(var c : char)
end BoundedBuffer
```

- Condition synchronization is via *condition variables*
  - » **wait(*cv*)** — Blocks the caller on a condition-specific queue
  - » **signal(*cv*)** — Wakes up a waiter if one exists
  - » **empty(*cv*)** — Indicates if any process is currently waiting

Monitor Example

Producer/Consumer synchronization

```
process Producer
begin
  loop
    <produce a character "c">
    BoundedBuffer.deposit(c)
  end loop
end Producer
```

WAIT!

```
process Consumer
begin
  loop
    BoundedBuffer.remove(data)
    <consume "data">
  end loop
end Consumer
```

```
monitor : BoundedBuffer
var
    nextIn,nextOut : …

    entry deposit(c : char)
    begin
      :
      :
    end deposit

    entry remove(var c : char)
    begin
      :
      :
    end remove

end BoundedBuffer
```

Monitor Example
Bounded buffer implementation

```
monitor : BoundedBuffer
var buffer           : array [0..n-1] of char
    nextIn,nextOut   : 0..n-1 := 0

    entry deposit(c : char)
    begin

      buffer[nextIn] := c
      nextIn    := nextIn+1 mod n

    end deposit

    entry remove(var c : char)
    begin

      c := buffer[nextOut]
      nextOut   := nextOut+1 mod n

    end remove

end BoundedBuffer
```


Lecture 7: Higher-Level Synch Primitives
Outline and key concepts

- The problem(s) with semaphores
- "Hoare" monitors
  - » Condition variables
- A disciplined use of synchronization primitives
- Implementing monitors
- "Mesa" monitors
  - » The priority inversion problem
- Readers/Writers synchronization
- Readings:
  - » Chapter 6 (Process Synchronization)

Monitor Example
Bounded buffer implementation

```
monitor : BoundedBuffer
var buffer            : array [0..n-1] of char
    nextIn,nextOut    : 0..n-1 := 0
    fullCount         : 0..n   := 0
    notEmpty, notFull : condition

    entry deposit(c : char)
    begin
      if (fullCount = n) then
        wait(notFull)
      end if

      buffer[nextIn] := c
      nextIn     := nextIn+1 mod n
      fullCount := fullCount + 1

      signal(notEmpty)
    end deposit

    entry remove(var c : char)
    begin
      if (fullCount = 0) then
        wait(notEmpty)
      end if

      c := buffer[nextOut]
      nextOut    := nextOut+1 mod n
      fullCount := fullCount - 1

      signal(notFull)
    end remove

end BoundedBuffer
```

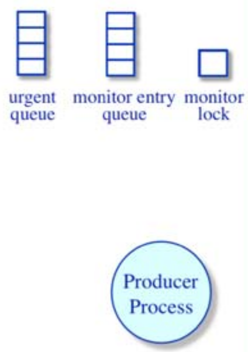

Semantics of synchronization
A discipline of concurrent programming

- What is the strongest statement we can make about the state of a monitor after a *waiter* wakes up?

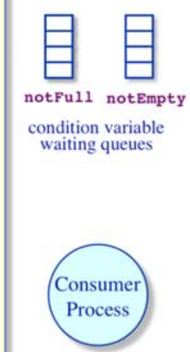
```
entry deposit(c : char)
begin
  if (fullCount = n) then

    {I}
    wait(notFull)
    {I ∧ B_cv}

  end if

   :
   :
end deposit
```

```
entry remove(var c : char)
begin
   :
   :
  c := buffer[nextOut]
  fullCount := fullCount - 1

  {I ∧ B_cv}
  signal(notFull)
  {I}

end remove
```


Realizing the Semantics
Implementing Hoare Monitors

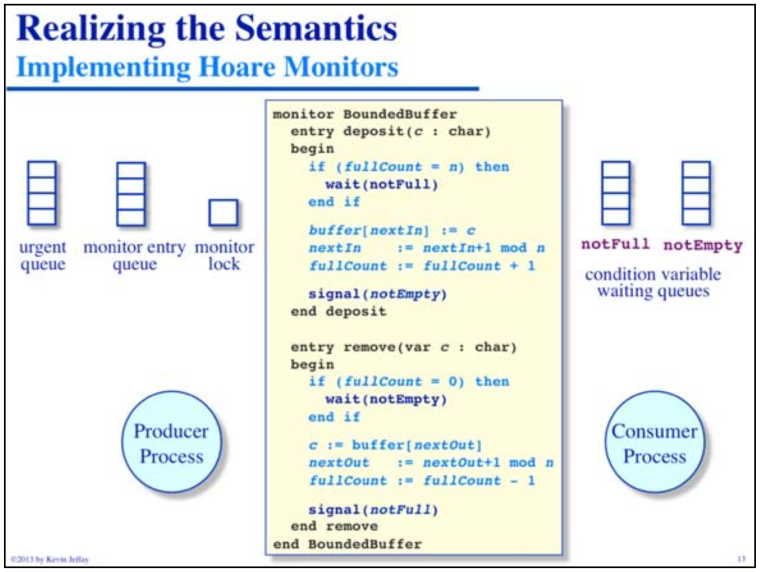


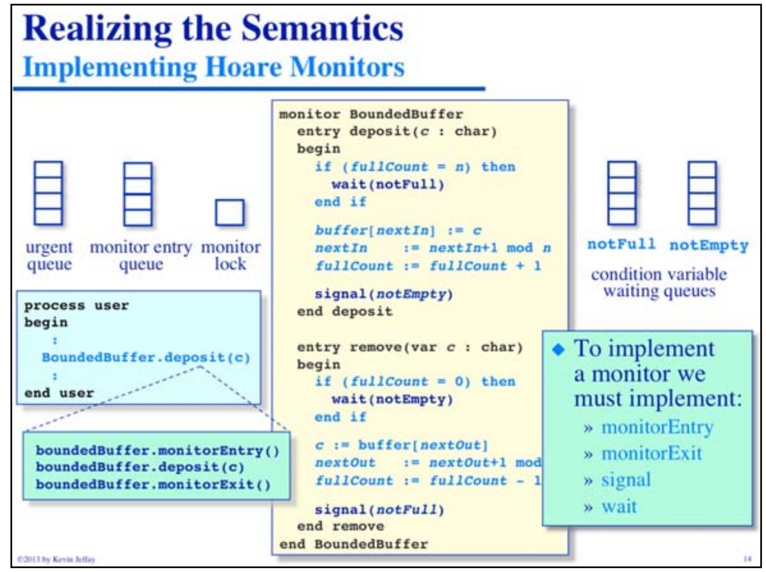

```
monitor BoundedBuffer
  entry deposit(c : char)
  begin
    if (fullCount = n) then
      wait(notFull)
    end if

    buffer[nextIn] := c
    nextIn    := nextIn+1 mod n
    fullCount := fullCount + 1

    signal(notEmpty)
  end deposit

  entry remove(var c : char)
  begin
    if (fullCount = 0) then
      wait(notEmpty)
    end if

    c := buffer[nextOut]
    nextOut   := nextOut+1 mod n
    fullCount := fullCount - 1

    signal(notFull)
  end remove
end BoundedBuffer
```



Lecture 7: Higher-Level Synch Primitives
Outline and key concepts

- The problem(s) with semaphores
- "Hoare" monitors
  - » Condition variables
- A disciplined use of synchronization primitives
- Implementing monitors
- "Mesa" monitors
  - » The priority inversion problem
- Readers/Writers synchronization
- Readings:
  - » Chapter 6 (Process Synchronization)


Realizing the Semantics
Implementing Hoare Monitors
urgent queue
monitor entry queue
monitor lock
monitor BoundedBuffer
entry deposit(c : char)
begin
if (fullCount = n) then
wait(notFull)
end if
buffer[nextIn] := c
nextIn := nextIn+1 mod n
fullCount := fullCount + 1
signal(notEmpty)
end deposit
entry remove(var c : char)
begin
if (fullCount = 0) then
wait(notEmpty)
end if
c := buffer[nextOut]
nextOut := nextOut+1 mod n
fullCount := fullCount - 1
signal(notFull)
end remove
end BoundedBuffer
notFull notEmpty
condition variable waiting queues
Producer Process
Consumer Process

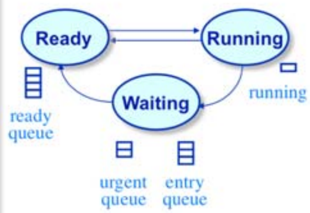

Realizing the Semantics
Implementing Hoare Monitors
urgent queue
monitor entry queue
monitor lock
process user
begin
:
BoundedBuffer.deposit(c)
:
end user
boundedBuffer.monitorEntry()
boundedBuffer.deposit(c)
boundedBuffer.monitorExit()
monitor BoundedBuffer
entry deposit(c : char)
begin
if (fullCount = n) then
wait(notFull)
end if
buffer[nextIn] := c
nextIn := nextIn+1 mod n
fullCount := fullCount + 1
signal(notEmpty)
end deposit
entry remove(var c : char)
begin
if (fullCount = 0) then
wait(notEmpty)
end if
c := buffer[nextOut]
nextOut := nextOut+1 mod
fullCount := fullCount - 1
signal(notFull)
end remove
end BoundedBuffer
notFull notEmpty
condition variable waiting queues
To implement a monitor we must implement:
» monitorEntry
» monitorExit
» signal
» wait

Realizing the Semantics

Implementing Hoare Monitors

```
var
  monitorCodeMutex : binarySem := 1       urgentQueue  : systemQueue
  monitorBusy      : boolean := FALSE     numWaiting   : integer    := 0
  entryQueue       : systemQueue          numSignalers : integer    := 0
```

```
procedure EnterMonitor()
begin




end EnterMonitor
```



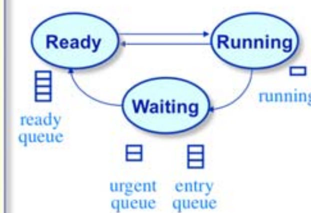
 15

Realizing the Semantics

Implementing Hoare Monitors

```
var
  monitorCodeMutex : binarySem := 1       urgentQueue  : systemQueue
  monitorBusy      : boolean := FALSE     numWaiting   : integer    := 0
  entryQueue       : systemQueue          numSignalers : integer    := 0
```

```
procedure ExitMonitor()
begin




end ExitMonitor
```



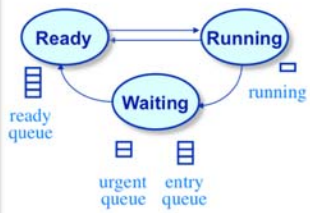
 16

Realizing the Semantics

Implementing Hoare Monitors

```
var
  monitorCodeMutex : binarySem := 1      urgentQueue  : systemQueue
  monitorBusy      : boolean := FALSE    numWaiting   : integer   := 0
  entryQueue       : systemQueue         numSignalers : integer   := 0
```

```
function wakeAWaiter() : boolean
begin

end wakeAWaiter
```



 17

Realizing the Semantics

Implementing Hoare Monitors

```
var
  monitorCodeMutex : binarySem := 1      urgentQueue  : systemQueue
  monitorBusy      : boolean := FALSE    numWaiting   : integer   := 0
  entryQueue       : systemQueue         numSignalers : integer   := 0
```

```
procedure Wait(cv : conditionVar)
begin
  var next : processID

end Wait
```

```
struct conditionVar
  queue      : systemQueue
  numWaiting : integer := 0
end struct
```

 18

Realizing the Semantics
Implementing Hoare Monitors

```
var
  monitorCodeMutex : binarySem := 1     urgentQueue   : systemQueue
  monitorBusy      : boolean := FALSE   numWaiting    : integer    := 0
  entryQueue       : systemQueue        numSignalers  : integer    := 0
```

```
procedure Signal(cv : conditionVar)
begin
  var waiter := processID









end Signal
```

```
struct conditionVar
  queue      : systemQueue
  numWaiting : integer := 0
end struct
```

Lecture 7: Higher-Level Synch Primitives
Outline and key concepts

- The problem(s) with semaphores
- "Hoare" monitors
  - » Condition variables
- A disciplined use of synchronization primitives
- Implementing monitors
- "Mesa" monitors
  - » The priority inversion problem
- Readers/Writers synchronization
- Readings:
  - » Chapter 6 (Process Synchronization)

Semantics of synchronization II

"Mesa" semantics

- Synchronization in the Mesa language from Xerox PARC:
  - » a *signal* (called **`notify()`**) is a "hint"

```
monitor BoundedBuffer
  var ...

  entry deposit(c : char)
  begin
    if (fullCount = n) then
      wait(notFull)
    end if

    buffer[nextIn] := c
    nextIn    := nextIn+1 mod n
    fullCount := fullCount + 1

    notify(notEmpty)
  end deposit

  entry remove(var c : char)
  begin
    if (fullCount = 0) then
      wait(notEmpty)
    end if

    c := buffer[nextOut]
    nextOut   := nextOut+1 mod n
    fullCount := fullCount - 1

    notify(notFull)
  end remove

end BoundedBuffer
```


Mesa Synchronization Semantics

The signal operation as a "hint"

- If the signal operation is a "hint" then the synchronization condition must be re-tested upon awakening

```
monitor BoundedBuffer
  var ...

  entry deposit(c : char)
  begin
    while (fullCount = n) do
      wait(notFull)
    end while

    buffer[nextIn] := c
    nextIn    := nextIn+1 mod n
    fullCount := fullCount + 1

    notify(notEmpty)
  end deposit

  entry remove(var c : char)
  begin
    while (fullCount = 0) do
      wait(notEmpty)
    end while

    c := buffer[nextOut]
    nextOut   := nextOut+1 mod n
    fullCount := fullCount - 1

    notify(notFull)
  end remove

end BoundedBuffer
```

Mesa Synchronization Semantics

Concurrent programming in *Java*

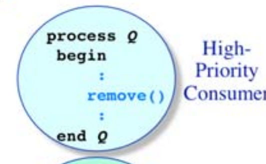
- Synchronization achieved via *synchronized classes*
  - » Provides mutual exclusion
- `wait` and `notify` synchronization
  - » *with* Mesa semantics
  - » *without* condition variables
- Other goodies:
  - » Any object can be synchronized
  - » `notifyAll` wakes up *all* waiting threads
  - » `wait` can take a *timeout* parameter

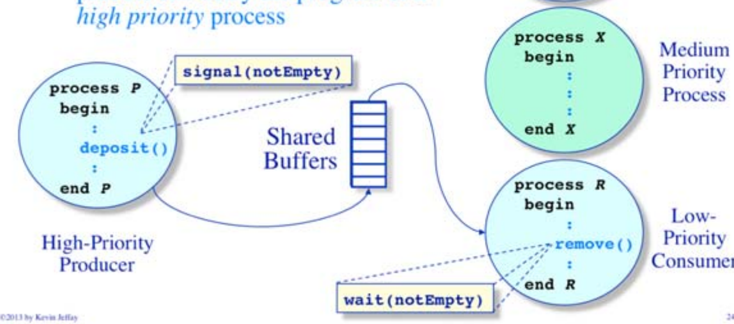
```
class BoundedBuffer {
  private char buffer[MAX_CHARS];
  private int nextIn,nextOut,fullCount;

  public bundedBuffer {
    nextIn    = 0; nextOut  = 0;
    fullCount = 0;
  }

  synchronized public deposit(char c) {
    while(fullCount == MAX_CHARS) {
      wait(); }
      :
    notify();
  }

  synchronized public char remove() {
    while(fullCount == 0) {
      wait(); }
      :
    notify();
  }
}
```


Mesa v. Hoare semantics; why the difference?

The *priority inversion* problem

- Consider a set of communicating processes with varying priority
  - » With Hoare semantics a *low priority* process can delay the progress of a *high priority* process





Mesa v. Hoare semantics; why the difference?
The *priority inversion* problem

- Can priority inversion really happen?!
- Consider the (ill-fated!) 1997 Mars rover…

http://research.microsoft.com/en-us/um/people/mbj/mars_pathfinder/

```
process Q
 begin
   :
   remove()
   :
 end Q
```

High-Priority Consumer

```
process X
 begin
   :
   :
   :
 end X
```

Medium Priority Process

```
process R
 begin
   :
   remove()
   :
 end R
```

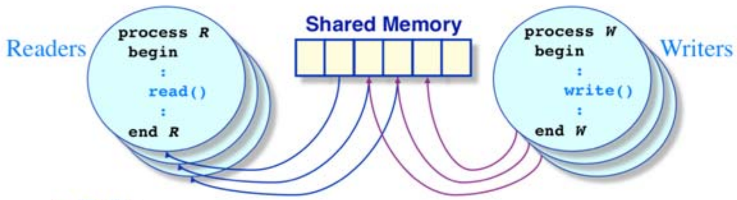
Low-Priority Consumer

Lecture 7: Higher-Level Synch Primitives
Outline and key concepts

- The problem(s) with semaphores
- "Hoare" monitors
  - » Condition variables
- A disciplined use of synchronization primitives
- Implementing monitors
- "Mesa" monitors
  - » The priority inversion problem
- Readers/Writers synchronization
- Readings:
  - » Chapter 6 (Process Synchronization)

Readers/Writers Synchronization

A generalization of producer/consumer systems



- Rules
  - » Multiple readers may be reading simultaneously
  - » Only one writer may be active at a time
  - » Reading and writing cannot proceed simultaneously
- Issues
  - » Makes sure readers don't starve writers (& vice versa)

Readers/Writers Synchronization

A monitor-based solution — *structure*

```
process Readeri
begin
  loop
    Sync_RW.StartRead
    <read the desired data>
    Sync_RW.EndRead
  end loop
end Readeri
```

WAIT!

```
globals
  buffer : array [0..n-1] of char
  readerPrt,writerPtr : 0..n-1
```

```
monitor : Sync_RW
var numReaders  : int      := 0
    writerBusy  : boolean  := FALSE
    OKtoRead, OKtoWrite : condition

    entry StartRead()
    entry EndRead()

    entry StartWrite()
    entry EndWrite()
end Sync_RW
```

```
process Writeri
begin
  loop
    Sync_RW.StartWrite
    <write the desired data>
    Sync_RW.EndWrite
  end loop
end Writeri
```


Readers/Writers Synchronization

A monitor-based solution — *details*

```
monitor : Sync_RW
  var numReaders : int := 0,   writerBusy : boolean := FALSE
      OKtoRead, OKtoWrite : condition

  entry StartRead()                    entry StartWrite()
  begin                                begin

                                       end StartWrite
  end StartRead

  entryEndRead()                       entry EndWrite()
  begin                                begin

  end EndRead
                                       end EndWrite
end Sync_RW
```


We're Done With Shared Memory!

So how's your memory holding out?!


- If you haven't shared too much of your memory then you should be able to remember the difference between…
  - » A condition variable and a semaphore
  - » Producer/consumer synchronization and readers/writers synchronization
  - » A "Hoare" monitor and a Mesa monitor/Java synchronized class
  - » …