CSCE 451/851
Operating Systems Principles

Higher-Level Synchronization Primitives

Steve Goddard
goddard@cse.unl.edu

http://www.cse.unl.edu/~goddard/Courses/CSCE451

The problem with semaphores

- Too general — one primitive for both *mutual exclusion* and *condition synchronization*
- Example: A P/C system with multiple producers & consumers

```
globals fullBuffers  : semaphore := 0
        emptyBuffers : semaphore := n
        mutex : binary_semaphore :=

process Producer                    process Consumer
begin                               begin
  loop                                loop
    <produce a character "c">           down(fullBuffers)
      down(emptyBuffers)                  down(mutex)
      down(mutex)                         c := buf[nextOut]
      buf[nextIn] := c                    nextOut := nextOut+1 mod n
      nextIn := nextIn+1 mod n            up(mutex)
      up(mutex)                           up(emptyBuffers)
      up(fullBuffers)                     <consume a character "c">
    end loop                            end loop
end Producer                        end Consumer
```

Higher-Level Synchronization

Hoare Monitors

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

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

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

- Condition synchronization 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

3

Monitor Example

Producer/Consumer synchronization

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

```
process Consumer
begin
  loop
    BoundedBuffer.remove(c)
    < consume a character "c" >
  end loop
end Consumer
```

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

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

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

end BoundedBuffer
```

4

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

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

      signal(notEmpty)                 signal(notFull)
    end deposit                      end remove

end BoundedBuffer
```

5

A Different Synchronization Example
Readers/Writers synchronization

- A generalization of producer consumer systems
  - » Typically involves a *set* of processes (> 2)
  - » Reading is non-destructive
  - » Writing updates data (rather than creating it)
- 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)

6

Readers/Writers Synchronization

A monitor-based solution — *structure*

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

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

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

    entry StartRead()
    entry EndRead()

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

7

Readers/Writers Synchronization

A monitor-based solution — *details*

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

  entry StartRead()
  begin
    if (writerBusy OR
        NOT empty(OKtoWrite)) then
      wait(OKtoRead)
    end if
    numReaders += 1
    signal(OKtoRead)
  end StartRead

  entry StartWrite()
  begin
    if (writerBusy OR
        numReaders ° 0) then
      wait(OKtoWrite)
    end if
    writerBusy := TRUE
  end StartWrite

  entry EndRead()
  begin
    numReaders := numReaders - 1
    if (numReaders = 0) then
      signal(OKtoWrite)
    end if
  end EndRead

  entry EndWrite()
  begin
    writerBusy := FALSE
    if (NOT empty(OKtoRead)) then
      signal(OKtoRead)
    else
      signal(OKtoWrite)
    end if
  end EndWrite

end Sync_RW
```

8

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?

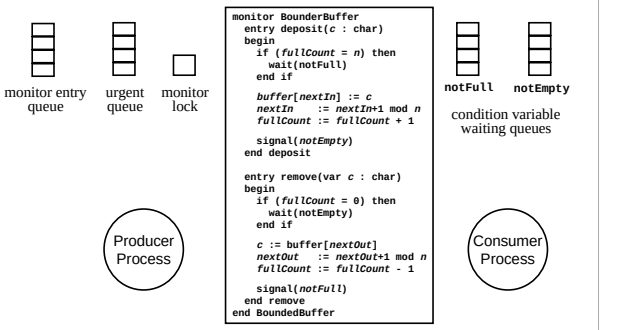
```
entry deposit(c : char)
begin
  if (fullCount = n) then

     wait(notFull)


  end if

   :
   :
end deposit
```

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


  signal(notFull)


end remove
```

9

Realizing the Semantics

Implementing Monitors




```
monitor BounderBuffer
  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
```

10

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

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

    notify(notEmpty)                 notify(notFull)
  end deposit                      end remove

end BoundedBuffer
```

11