CS 333
Introduction to Operating Systems

Class 6 – Monitors and Message Passing

Jonathan Walpole
Computer Science
Portland State University
Monitors

- It is difficult to produce correct programs using semaphores
  - correct ordering of up and down is tricky!
  - avoiding race conditions and deadlock is tricky!
  - boundary conditions are tricky!

- Can we get the compiler to generate the correct semaphore code for us?
  - what are suitable higher level abstractions for synchronization?
Monitors

- Related shared objects are collected together
- **Compiler enforces encapsulation/mutual exclusion**
  - Encapsulation:
    - Local data variables are accessible only via the monitor’s entry procedures (like methods)
  - Mutual exclusion
    - A monitor has an associated mutex lock
    - Threads must acquire the monitor’s mutex lock before invoking one of its procedures
Monitors and condition variables

- But we need two flavors of synchronization
  - Mutual exclusion
    - Only one at a time in the critical section
    - Handled by the monitor’s mutex
  - Condition synchronization
    - Wait until a certain condition holds
    - Signal waiting threads when the condition holds
Monitors and condition variables

- Condition variables (cv) for use within monitors
  - `wait(cv)`
    - thread blocked (queued) until condition holds
    - monitor mutex released!!
  - `signal(cv)`
    - signals the condition and unblocks (dequeues) a thread
Monitor structures

Shared data

Condition variables

Monitor entry queue

"Entry" methods

Local methods

Initialization code

List of threads waiting to enter the monitor

Can be called from outside the monitor. Only one active at any moment.

Local to monitor (Each has an associated list of waiting threads)

Can be called from outside the monitor. Only one active at any moment.
Monitor example for mutual exclusion

process Producer
begin
  loop
    <produce char “c”>
      BoundedBuffer.deposit(c)
    end loop
  end Producer

process Consumer
begin
  loop
    BoundedBuffer.remove(c)
    <consume char “c”>
  end loop
  end Consumer
Observations

- That’s much simpler than the semaphore-based solution to producer/consumer (bounded buffer)!
- ... but where is the mutex?
- ... and what do the bodies of the monitor procedures look like?
Monitor example with condition variables

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 = n) then
    wait(notEmpty)
  end if
  c := buffer[nextOut]
  nextOut := nextOut+1 mod n
  fullCount := fullCount-1
  signal(notFull)
end remove

end BoundedBuffer
**Condition variables**

“Condition variables allow processes to synchronize based on some state of the monitor variables.”
Condition variables in producer/consumer

“NotFull” condition
“NotEmpty” condition

- Operations `Wait()` and `Signal()` allow synchronization within the monitor

- When a producer thread adds an element...
  - A consumer may be sleeping
  - Need to wake the consumer... `Signal`
Condition synchronization semantics

- "Only one thread can be executing in the monitor at any one time."

- **Scenario:**
  - Thread A is executing in the monitor
  - Thread A does a *signal* waking up thread B
  - What happens now?
  - Signaling and signaled threads can not both run!
  - ... so which one runs, which one blocks, and on what queue?
Monitor design choices

- **Condition variables introduce a problem for mutual exclusion**
  - only one process active in the monitor at a time, so what to do when a process is unblocked on **signal**?
  - must not block holding the mutex, so what to do when a process blocks on **wait**?

- **Should signals be stored/remembered?**
  - signals are not stored
  - if signal occurs before wait, signal is lost!

- **Should condition variables count?**
Monitor design choices

- **Choices when A signals a condition that unblocks B**
  - A waits for B to exit the monitor or blocks again
  - B waits for A to exit the monitor or block
  - Signal causes A to immediately exit the monitor or block (... but awaiting what condition?)

- **Choices when A signals a condition that unblocks B & C**
  - B is unblocked, but C remains blocked
  - C is unblocked, but B remains blocked
  - Both B & C are unblocked ... and compete for the mutex?

- **Choices when A calls wait and blocks**
  - a new external process is allowed to enter
  - but which one?
Option 1: Hoare semantics

- What happens when a Signal is performed?
  - signaling thread (A) is suspended
  - signaled thread (B) wakes up and runs immediately

- Result:
  - B can assume the condition is now true/satisfied
  - Hoare semantics give strong guarantees
  - Easier to prove correctness

- When B leaves monitor, A can run.
  - A might resume execution immediately
  - ... or maybe another thread (C) will slip in!
Option 2: MESA Semantics (Xerox PARC)

- **What happens when a Signal is performed?**
  - the signaling thread (A) continues.
  - the signaled thread (B) waits.
  - when A leaves monitor, then B runs.

- **Issue: What happens while B is waiting?**
  - can another thread (C) run after A signals, but before B runs?

- **In MESA semantics a signal is more like a hint**
  - Requires B to recheck the state of the monitor variables (the invariant) to see if it can proceed or must wait some more
Code for the “deposit” entry routine

```
monitor BoundedBuffer
  var buffer: array[n] of char
  nextIn, nextOut: int = 0
  cntFull: int = 0
  notEmpty: Condition
  notFull: Condition

  entry deposit(c: char)
    if cntFull == N
      notFull.Wait()
    endIf
    buffer[nextIn] = c
    nextIn = (nextIn + 1) mod N
    cntFull = cntFull + 1
    notEmpty.Signal()
  endEntry

  entry remove()
    ...

endMonitor
```
Code for the “deposit” entry routine

```plaintext
monitor BoundedBuffer
  var buffer: array[n] of char
  nextIn, nextOut: int = 0
  cntFull: int = 0
  notEmpty: Condition
  notFull: Condition

  entry deposit(c: char)
    while cntFull == N
      notFull.Wait()
    endWhile
    buffer[nextIn] = c
    nextIn = (nextIn + 1) mod N
    cntFull = cntFull + 1
    notEmpty.Signal()
  endEntry

  entry remove()
    ...
  endEntry
endMonitor
```
Code for the "remove" entry routine

```plaintext
monitor BoundedBuffer
  var buffer: array[n] of char
  nextIn, nextOut: int = 0
  cntFull: int = 0
  notEmpty: Condition
  notFull: Condition

  entry deposit(c: char)
    ...

  entry remove()
    if cntFull == 0
      notEmpty.Wait()
    endIf
    c = buffer[nextOut]
    nextOut = (nextOut+1) mod N
    cntFull = cntFull - 1
    notFull.Signal()
  endEntry

endMonitor
```
Code for the “remove” entry routine

```
monitor BoundedBuffer
    var buffer: array[n] of char
    nextIn, nextOut: int = 0
    cntFull: int = 0
    notEmpty: Condition
    notFull: Condition
    
    entry deposit(c: char)
    ...

    entry remove()
        while cntFull == 0
            notEmpty.Wait()
        endwhile
        c = buffer[nextOut]
        nextOut = (nextOut+1) mod N
        cntFull = cntFull - 1
        notFull.Signal()
    endEntry

endMonitor
```

{ MESA Semantics }
“Hoare Semantics”

What happens when a Signal is performed?

The signaling thread (A) is suspended. The signaled thread (B) wakes up and runs immediately.

B can assume the condition is now true/satisfied

From the original Hoare Paper:

“No other thread can intervene [and enter the monitor] between the signal and the continuation of exactly one waiting thread."

“If more than one thread is waiting on a condition, we postulate that the signal operation will reactivate the longest waiting thread. This gives a simple neutral queuing discipline which ensures that every waiting thread will eventually get its turn.”
Implementing Hoare Semantics

- Thread A holds the monitor lock
- Thread A *signals* a condition that thread B was waiting on
- Thread B is moved back to the ready queue
- Thread A must be suspended...
  - the monitor lock must be passed from A to B
- When B finishes it releases the monitor lock
- Thread A must re-quire the lock
  - Perhaps A is blocked, waiting to re-aquire the lock
Implementing Hoare Semantics

- **Problem:**
  - Possession of the monitor lock must be passed **directly** from A to B
  - Simply ending monitor entry methods with `monLock.Unlock()` will not work
  - A’s request for the monitor lock must be expedited
Implementing Hoare Semantics

- Implementation Ideas:
  - Consider a thread like A to be “urgent”.
    - Thread C is not “urgent”.
  - Consider two wait lists associated with each MonitorLock
    - UrgentlyWaitingThreads
    - NonurgentlyWaitingThreads
  - Want to wake up urgent threads first, if any
Recommendation for SPANK implementation:

- Do not modify the methods that I supplied, because future code will use them
- Create new classes:
  - MonitorLock -- similar to Mutex
  - HoareCondition -- similar to Condition
Brinch-Hansen Semantics

- **Hoare Semantics**
  - On signal, allow signaled process to run
  - Upon its exit from the monitor, signaler process continues.

- **Brinch-Hansen Semantics**
  - Signaler must immediately exit following any invocation of signal
  - Restricts the kind of solutions that can be written
  - ... but monitor implementation is easier
Reentrant code

- A function/method is said to be reentrant if...

  A function that has been invoked may be invoked again before the first invocation has returned, and will still work correctly

- Recursive routines are reentrant

- In the context of concurrent programming...

  A reentrant function can be executed simultaneously by more than one thread, with no ill effects
Reentrant Code

- Consider this function...

```plaintext
var count: int = 0
function GetUnique () returns int
    count = count + 1
return count
endFunction
```

- What if it is executed by different threads concurrently?
Reentrant Code

- Consider this function...

```plaintext
var count: int = 0
function GetUnique () returns int
    count = count + 1
    return count
endFunction
```

- What if it is executed by different threads concurrently?
  - The results may be incorrect!
  - This routine is not reentrant!
When is code reentrant?

- **Some variables are**
  - "local" -- to the function/method/routine
  - "global" -- sometimes called "static"

- **Access to local variables?**
  - A new stack frame is created for each invocation
  - Each thread has its own stack

- **What about access to global variables?**
  - *Must use synchronization!*
Making this function reentrant

```plaintext
var count: int = 0
    myLock: Mutex

function GetUnique () returns int
    var i: int
    myLock.Lock()
    count = count + 1
    i = count
    myLock.Unlock()
    return i
endFunction
```
Message Passing

- Interprocess Communication
  - via shared memory
  - across machine boundaries

- Message passing can be used for synchronization or general communication

- Processes use send and receive primitives
  - receive can block (like waiting on a Semaphore)
  - send unblocks a process blocked on receive (just as a signal unblocks a waiting process)
Producer-consumer with message passing

- The basic idea:
  - After producing, the producer sends the data to consumer in a message
  - The system buffers messages
    - The producer can out-run the consumer
    - The messages will be kept in order
  - But how does the producer avoid overflowing the buffer?
    - After consuming the data, the consumer sends back an “empty” message
  - A fixed number of messages (N=100)
  - The messages circulate back and forth.
Producer-consumer with message passing

const N = 100 -- Size of message buffer
var em: char
for i = 1 to N -- Get things started by
  Send (producer, &em) -- sending N empty messages
endFor

thread consumer
  var c, em: char
  while true
    Receive (producer, &c) -- Wait for a char
    Send (producer, &em) -- Send empty message back
    // Consume char...
  endwhile
end
Producer-consumer with message passing

thread producer
  var c, em: char
  while true
    // Produce char c...
    Receive(consumer, &em)  -- Wait for an empty msg
    Send(consumer, &c)   -- Send c to consumer
  endwhile
end
Design choices for message passing

- **Option 1: Mailboxes**
  - System maintains a buffer of sent, but not yet received, messages
  - Must specify the size of the mailbox ahead of time
  - Sender will be blocked if the buffer is full
  - Receiver will be blocked if the buffer is empty
Design choices for message passing

- **Option 2: No buffering**
  - If Send happens first, the sending thread blocks
  - If Receiver happens first, the receiving thread blocks
  - Sender and receiver must **Rendezvous** (ie. meet)
  - Both threads are ready for the transfer
  - The data is copied / transmitted
  - Both threads are then allowed to proceed
Barriers

- Processes approaching a barrier
- All processes but one blocked at barrier
- Last process arrives; all are let through
Quiz

- What is the difference between a monitor and a semaphore?
  - Why might you prefer one over the other?
- How do the wait/signal methods of a condition variable differ from the up/down methods of a semaphore?
- What is the difference between Hoare and Mesa semantics for condition variables?
  - What implications does this difference have for code surrounding a wait() call?