CS 333
Introduction to Operating Systems

Class 6 - Monitors and Message Passing

Jonathan Walpole
Computer Science
Portland State University
Monitors
Monitors

- It is difficult to produce correct programs using semaphores
  - correct ordering of down and up 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 or programming convention 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
  - `cv.wait(mon-mutex)`
    - thread blocked (queued) until condition holds
    - *Must not block while holding mutex!*
    - monitor mutex must be released!
    - Monitor mutex need not be specified by programmer if compiler is enforcing mutual exclusion
  - `cv.signal()`
    - signals the condition and unblocks (dequeues) a thread
Monitor structures

- Shared data
- Condition variables
- Monitor entry queue
- "Entry" methods
- Local methods
- Initialization code

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

List of threads waiting to enter the monitor

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

monitor: BoundedBuffer
var buffer : ...;
nextIn, nextOut :... ;

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

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

end BoundedBuffer
Observations

- That’s much simpler than the semaphore-based solution to producer/consumer (bounded buffer)!
- ... but where is the mutex?
  - Here we assume the compiler is enforcing mutual exclusion among accesses to a monitor type (like synchronized types in Java)
- ... and what do the bodies of the monitor procedures look like?
Monitor example with condition variables

```plaintext
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 = n) then
    wait(notFull)                wait(notEmpty)
  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
```
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 how (on what queue)?
Monitor design choices

- **Condition variables introduce two problems for mutual exclusion**
  - What to do in signal: only one process can be active in the monitor at a time
    - the signaling one is already in
    - the signaled one was in when it waited and will be in again on return from wait
  - What to do on wait
    - must not block holding the mutex!
    - How do we know which mutex to release?
    - What if monitor calls are nested?
Monitor design choices

- **A signals a condition that unblocks B**
  - A blocks until B exits the monitor?
  - B blocks until A exits the monitor?
  - Does the condition that B was waiting for hold when B runs?

- **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, i.e. broadcast signal
    - ... they must compete for the mutex
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 it was waiting for now holds
  - 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 the condition that caused A to generate the signal be changed before B runs?

- In MESA semantics a signal is more like a hint
  - Requires B to recheck the condition on which it waited to see if it can proceed or must wait some more
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)
    if cntFull == N
      notFull.Wait()
    endIf
    buffer[nextIn] = c
    nextIn = (nextIn+1) mod N
    cntFull = cntFull + 1
    notEmpty.Signal()
  endEntry

  entry remove()
    ...
  endEntry

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

```pascal
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()
...
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()
  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

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

endMonitor
```
Monitors in Blitz

- They are implemented as a programming convention
  - The monitor lock is managed explicitly in the program
  - The wait call on condition variables takes the monitor lock as a parameter

- They have MESA semantics
  - When a waiting thread is awoken, you can’t assume that the condition it was waiting for still holds, even if it held when signal was called!
“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?
  - B should run immediately
  - 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-acquire the lock
  - A is blocked, waiting to re-acquire the lock
Implementing Hoare Semantics

- Problem:
  - Possession of the monitor lock must be passed directly from A to B and then eventually back to A
Implementing Hoare Semantics

- Recommendation for Project 4 implementation:
  - Do not modify the mutex methods provided, because future code will use them.
  - Create new classes:
    - `MonitorLock` -- similar to `Mutex`
    - `HoareCondition` -- similar to `Condition`
Message Passing
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
    - Consumer starts by sending N empty messages
  - A fixed number of messages (N=100)
  - The messages circulate back and forth.
Producer-consumer with message passing

\[
\begin{align*}
\text{const } N &= 100 \quad \text{-- Size of message buffer} \\
\text{var } em &: \text{ char} \\
\text{for } i &= 1 \text{ to } N \quad \text{-- Get things started by} \\
\quad \text{Send (producer, } \&em) \quad \text{-- sending } N \text{ empty messages} \\
\end{align*}
\]

\[
\begin{align*}
\text{thread consumer} \\
\quad \text{var } c, em &: \text{ char} \\
\quad \text{while true} \\
\quad \quad \text{Receive (producer, } \&c) \quad \text{-- Wait for a char} \\
\quad \quad \text{Send (producer, } \&em) \quad \text{-- Send empty message back} \\
\quad \quad // \text{ Consume char...} \\
\quad \quad \text{endWhile} \\
\end{align*}
\]
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 Receive 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
Review of a Practical Concurrent Programming Issue - Reentrant Functions
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

- 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!
Does this work?

```plaintext
var count: int = 0

myLock: Mutex

function GetUnique () returns int

myLock.Lock()

count = count + 1

myLock.Unlock()

return count

endFunction
```
What about this?

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

function GetUnique () returns int
    myLock.Lock()
    count = count + 1
    return count
    myLock.Unlock()
endFunction
```
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
```
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 wait/signal 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?