CS399 New Beginnings

Jonathan Walpole
Monitors and Message Passing
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?
- High 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

  *Threads must acquire the monitor’s mutex lock before invoking one of its procedures*
Monitors & Condition Variables

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 & Condition Variables

Condition variables (cv) for use within monitors

```cpp
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

```cpp
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

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? … and what do the bodies of the procedures look like?

Here we assume the compiler is enforcing mutual exclusion among accesses to a monitor type - like synchronized types in Java
Monitor Example

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.
Producer-Consumer Conditions

\textit{NotFull} condition
\textit{NotEmpty} condition

Operations \texttt{Wait()} and \texttt{Signal()} allow synchronization within the monitor

When a producer thread adds an element…
- A consumer may be sleeping
- Need to wake the consumer… \texttt{Signal}
Condition Variable Semantics

Only one thread at a time can execute in the monitor

Scenario:
- Thread A is executing in the monitor
- Thread A does a \textit{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

1. 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

2. 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
- Does A block until B exits the monitor?
- Does B block until A exits the monitor?
- Does the condition that B was waiting for still hold when B runs?

A signals a condition that unblocks B & C
- Is B unblocked, but C remains blocked?
- Is C unblocked, but B remains blocked?
- Are both B & C unblocked, i.e. broadcast signal
  ... if so, 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 certain strong guarantees

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

What happens when a Signal is performed?
- The signaling thread (A) continues
- The signaled thread (B) waits
- When A leaves the monitor, then B resumes

**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
Example Use of Hoare Semantics

```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()
    ...
endMonitor
```
Example Use of Mesa Semantics

\begin{verbatim}
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
\end{verbatim}
Example Use of Hoare Semantics

```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
```
Example Use of Mesa Semantics

```
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 not implemented by the compiler
- 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!
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!
  - The monitor lock must be passed from A to B immediately
  - Thread A must be suspended
When B finishes it releases the monitor lock
  - A is blocked, waiting to re-acquire the lock
  - A must re-acquire the lock eventually, but perhaps not immediately
Implementing Hoare Semantics

The challenge:
- 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)
Message Passing Example

Producer-consumer example:
- After producing, the producer sends the data to consumer in a message
- The system buffers messages (kept in order)
- The producer can out-run the consumer

How does the producer avoid overflowing the buffer?
- The consumer sends empty messages to the producer
- The producer blocks waiting for empty messages
- The consumer starts by sending N empty messages
  - N is based on the buffer size
Message Passing Example

```plaintext
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
```
Message Passing Example

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
Buffering Design Choices

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
Buffering Design Choices

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
Review of a Practical Concurrent Programming Issue – Reentrant Functions
Reentrant Functions

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 Functions

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 Functions

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!
Local vs Global Variables

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?

```go
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
```
Solution

```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?