CS510 Operating System Foundations

Jonathan Walpole
Monitor Semantics, Message Passing, & Concurrency Review
Overview of Monitor Structure

Monitor’s shared data:
- Each condition variable 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.

Condition variables (these are also shared monitor variables!)

Monitor entry queue:

Local methods:

Initialization code:
Compiler Support for Monitors

If monitors are part of the programming language, the compiler can generate the monitor lock and unlock code automatically

- programmer may simply define the type of a data item to be *monitor* (or *synchronized*)
- *the monitor mutex to be released in wait is identified implicitly by the condition variable’s monitor*

If monitors are unknown to the compiler (as in Blitz), the programmer writes the monitor mutex code

- wait takes the monitor mutex as a parameter
Producer-Consumer with Monitors

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
Use of 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) 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 vs Semaphores

OK to wait on a condition variable while holding the monitor’s mutex
- because the wait call atomically releases the mutex before sleeping
- and acquires it again after waking, and before returning

Not OK to down on a semaphore while holding the mutex needed by the thread that will issue the corresponding up!
Other Observations

In this example, monitor types are part of the programming language.

The compiler is enforcing mutual exclusion among accesses to a monitor type:
- the monitor mutex lock and unlock operations are not visible in the source code
- the monitor lock is not passed as a parameter to wait
Condition Variables in Blitz

Monitors are not known to the KPL compiler in Blitz!

Condition class is used to implement monitors
- `Condition.wait (mutex)`
- `Condition.signal (mutex)`
- `Condition.broadcast(mutex)`

Mutex must be passed to condition methods to indicate which monitor the condition is in

Mutex lock/unlock code must be written by the programmer on entry/exit to/from monitor methods
Condition Variable Semantics

Mutual exclusion requirement:
- *Only one thread at a time can execute in the monitor*

Scenario:
- Thread A is executing in the monitor
- Thread A does a *signal* waking up thread B
- Which thread runs in the monitor next?
- A and B must run in some order
- Which one runs, which one blocks, and how (on what queue)?
Option 1: Hoare Semantics

What happens when A signals B?
- A is suspended (added to waiting thread list for monitor mutex)
- B wakes up and runs immediately (i.e., nothing else runs in the monitor between the signal and the wakeup of B)
- A hands B the monitor mutex directly

A can only run when B leaves the monitor on exit or in another wait
- A only resumes immediately if it is at the head of the mutex list
- If not, other threads run before it
Memory Invariance under Hoare Semantics

When B wakes up, the state of the monitor’s variables are the same as when A issued the signal.

When A resumes after signal, the state of the monitor’s variables may have changed!

Implications:
- Memory invariance is lost across wait
- Memory invariance is lost across signal
- Memory invariance is preserved across signal to wakeup from wait
Option 2: MESA Semantics

What happens when A signals B?
- A continues executing in the monitor
- B tries to lock the monitor mutex, and takes its place at the back of the list
- B resumes when its turn comes around
Memory Invariance under MESA Semantics

When B wakes up, the state of the monitor’s variables may not be the same as when A issued the signal!
- the signal is more like a hint
- the waking thread must check to see if it should continue or wait again

When A continues after signal, the state of the monitor’s variables have not changed, because A retained the mutex
Memory Invariance under MESA Semantics

Implications:
- Memory invariance is lost across wait
- Memory invariance is not lost across signal
- Memory invariance is not preserved across signal to wakeup from wait

The different memory invariance effect determine how you write code that uses condition variables!
- You really need to know the semantics of your condition variables!!!
Example Use of Hoare Semantics

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

    entry remove()
        ...

endMonitor
```
Example Use of Hoare 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()
        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 have MESA semantics
- When a waiting thread resumes, you can’t assume that the state of the monitor’s variables is the same as when signal was called!

There is a broadcast call in addition to signal
- broadcast wakes up all threads waiting on the condition variable, not just the first one
- broadcast makes no sense for condition variables with Hoare semantics!
Implementing Hoare Semantics

In Assignment 4 you must modify Blitz condition variables to have Hoare semantics:
- Do not modify the mutex methods provided, because future code will use them
- Create new classes:
  MonitorLock  -- similar to Mutex
  HoareCondition  -- similar to Condition

You must write your own test code to determine if they actually have Hoare semantics!
Message Passing
Message Passing

Inter-Process Communication (IPC) system calls to send and receive messages to/from ports
- for shared memory architectures and distributed systems
- receive can block a thread and add it to a waiting thread’s list for the port (like wait and down)
- send unblocks a thread from the list and makes it ready to run (like signal and up)

Message passing can be used for synchronization or general communication
Message Passing Example

Producer-consumer example:
- The producer **sends** the data to consumer in a message
- The consumer **receives** the data in the message
- The system buffers outstanding messages (kept in order)
- We need flow control to prevent the producer out-running the consumer!

Flow control idea:
- 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 consumer’s buffer size
Message Passing Consumer

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 Producer

```plaintext
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: System Buffering

- System stores sent, but not yet received, messages
- Buffering capacity is finite
- 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 sender blocks
- If Receive happens first, the receiver blocks
- Sender and receiver must *Rendezvous* (ie. meet) to exchange the message
- Both threads are ready for the transfer
- The data is copied directly from one to the other
- Both threads are then allowed to proceed
Thread-Safe Routines
Thread-Safe Functions

Consider this library function...

```c
var count: int = 0

function GetUnique () returns int
    count = count + 1
    return count
endFunction
```

Is it safe to execute concurrently in multiple threads?
Thread-Safe 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

Variables that are local to the function invocation are allocated (dynamically at function-call time) on the stack of the calling thread.

- these are only accessible to this one thread!
- they are not directly involved in race conditions

Static variables, allocated at compile time, and dynamically allocated variables (malloc) on the heap, are global variables

- they are accessible to all threads
- they can be involved in race conditions
- they require 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
```
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
```
Concurrency/Synchronization Review

Race conditions
Mutual exclusion concept
  - Interrupt disabling
  - Atomic instructions
  - Spin locks
  - Blocking locks
Semaphores
  - Binary semaphores
  - Counting semaphores
Monitors and condition variables