Monitors
Programming Complexity

There are many ways to introduce bugs using locks and semaphores
- forget to lock *somewhere* in the program
- forget to unlock
  - maybe in an error path or conditional
- non-unique association between data and locks
- reuse of global value copied to a local
- initialize lock or semaphore to wrong value
- incorrect ordering of down and up
Can The Compiler Help?

If the set of locking rules is well-defined, why can’t the compiler generate the correct synchronization code for us?

We need high level language abstractions for concurrency
Monitors

*Related* shared/global variables are grouped together and protected by the same lock.

Encapsulation and mutual exclusion can be enforced by the compiler or by programming convention:

- **Encapsulation**
  
  *Local data variables are accessible only via the monitor’s “entry procedures” (i.e., external methods)*

- **Mutual exclusion**
  
  *Threads must acquire the monitor’s mutex lock before invoking one of its procedures*
Mutual Exclusion in Monitors

A monitor consists of its data and the methods that access it.
Each monitor has a mutex lock associated with it, called the monitor’s mutex.
The monitor’s mutex is acquired at the start of every monitor method and released at the end.
Only one thread is active in the monitor at any time!
Condition Variables

What if a thread needs to wait during its computation?

- maybe it can’t proceed until one of the monitor’s variables reaches a certain value
- i.e., producer can’t proceed if buffer is full
- another thread must run in the monitor while this one waits, so it must give up the mutex!

How will this thread wake up?
Condition Variables

Condition variables (cv) for use within monitors

**cv.wait()**
- Caller thread is blocked
- It is enqueued on this variable’s waiting threads list
- It must not block while holding the monitor’s mutex!
- The monitor’s mutex must be released in wait, so another thread can access the monitor’s data

**cv.signal()**
- Dequeues the thread at the head of the waiting list
- Unblocks it and places it on the ready list
Condition Variables vs Semaphores

Condition variables look like semaphores

The key difference:
- condition variables do not have an integer value
- a wait always causes the thread to block
- a signal only wakes up a thread if one is waiting
- a signal that occurs when no threads are waiting is lost!

Consider the implications of this for the Dining Philosopher’s solution in assignment 2!
Overview of Monitor Structure

- **Monitor’s shared data**
- **Condition variables** (these are also shared monitor variables!)
- **Initialization code**
- **Local methods**
- **Entry methods**
- **Monitor entry queue**

- **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.**
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
Use of 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

end BoundedBuffer

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
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
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

Implications:
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 implications for memory invariance affect how you write code that uses condition variables.

You really need to know the semantics of your condition variables!!!
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()
        ...
    endEntry
endMonitor
```
Example Use of Mesa Semantics

```c
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) % N
    cntFull = cntFull + 1
    notEmpty.Signal()
endEntry

entry remove()
    ...
endMonitor
```
Example Use of Hoare Semantics

```markdown
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
```

Hoare Semantics
Example Use of Mesa 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()
    while cntFull == 0
      notEmpty.Wait()
      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
- 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!