Semaphores

• Semaphore = a synchronization primitive
  – higher level of abstraction than locks
  – invented by Dijkstra in 1968, as part of the THE operating system

• A semaphore is:
  – a variable that is manipulated through two operations, P and V (Dutch for “test” and “increment”)
    • P(sem) (wait/down)
      – block until sem > 0, then subtract 1 from sem and proceed
    • V(sem) (signal/up)
      – add 1 to sem

• Do these operations atomically
Blocking in semaphores

• Each semaphore has an associated queue of threads
  – when P(sem) is called by a thread,
    • if sem was “available” (>0), decrement sem and let thread continue
    • if sem was “unavailable” (<=0), place thread on associated queue; dispatch some other runnable thread
  – when V(sem) is called by a thread
    • if thread(s) are waiting on the associated queue, unblock one
      – place it on the ready queue
      – might as well let the “V-ing” thread continue execution
      – or not, depending on priority
    • otherwise (when no threads are waiting on the sem), increment sem
      – the signal is “remembered” for next time P(sem) is called

• Semaphores thus have history
Abstract implementation

- **P/wait/down(sem)**
  - acquire “real” mutual exclusion
    - if sem is “available” (>0), decrement sem; *release “real” mutual exclusion*; let thread continue
    - otherwise, place thread on associated queue; *release “real” mutual exclusion*; run some other thread

- **V/signal/up(sem)**
  - acquire “real” mutual exclusion
    - if thread(s) are waiting on the associated queue, unblock one (place it on the ready queue)
    - if no threads are on the queue, sem is incremented
      - the signal is “remembered” for next time P(sem) is called
  - *release “real” mutual exclusion*
  - [the “V-ing” thread continues execution or is preempted]
Two types of semaphores

• **Binary semaphore** (aka mutex semaphore)
  – sem is initialized to 1
  – guarantees mutually exclusive access to resource (e.g., a critical section of code)
  – only one thread/process allowed entry at a time

• **Counting semaphore**
  – sem is initialized to N
    • N = number of units available
  – represents resources with many (identical) units available
  – allows threads to enter as long as more units are available
Usage

• From the programmer’s perspective, P and V on a binary semaphore are just like Acquire and Release on a lock
  
  P(sem)  
  ::  
  ::  
  ::  
  do whatever stuff requires mutual exclusion; could conceivably be a lot of code  
  ::  
  ::  
  V(sem)  
  – same lack of programming language support for correct usage

• Important differences in the underlying implementation, however
Pressing questions

• How do you acquire “real” mutual exclusion?

• Why is this any better than using a spinlock (test-and-set) or disabling interrupts (assuming you’re in the kernel) in lieu of a semaphore?

• What if some bozo issues an extra V?

• What if some bozo forgets to P?
**Example: Bounded buffer problem**

- **AKA “producer/consumer” problem**
  - there is a buffer in memory with N entries
  - producer threads insert entries into it (one at a time)
  - consumer threads remove entries from it (one at a time)

- **Threads are concurrent**
  - so, we must use synchronization constructs to control access to shared variables describing buffer state
Bounded buffer using semaphores
(both binary and counting)

```plaintext
var mutex: semaphore = 1 ; mutual exclusion to shared data
empty: semaphore = n ; count of empty buffers (all empty to start)
full: semaphore = 0 ; count of full buffers (none full to start)

producer:
P(empty) ; one fewer buffer, block if none available
P(mutex) ; get access to pointers
  <add item to buffer>
V(mutex) ; done with pointers
V(full) ; note one more full buffer

consumer:
P(full) ; wait until there’s a full buffer
P(mutex) ; get access to pointers
  <remove item from buffer>
V(mutex) ; done with pointers
V(empty) ; note there’s an empty buffer
  <use the item>
```

Note 1:
I have elided all the code concerning which is the first full buffer, which is the last full buffer, etc.

Note 2:
Try to figure out how to do this without using counting semaphores!
Example: Readers/Writers

• Description:
  – A single object is shared among several threads/processes
  – Sometimes a thread just reads the object
  – Sometimes a thread updates (writes) the object

  – We can allow multiple readers at a time
    • why?

  – We can only allow one writer at a time
    • why?
Readers/Writers using semaphores

\[
\begin{align*}
\text{var} & \quad \text{mutex: semaphore} = 1 \quad ; \text{controls access to readcount} \\
& \quad \text{wrt: semaphore} = 1 \quad ; \text{control entry for a writer or first reader} \\
& \quad \text{readcount: integer} = 0 \quad ; \text{number of active readers} \\
\text{writer:} & \\
& P(\text{wrt}) \quad ; \text{any writers or readers?} \\
& \quad \text{<perform write operation>} \\
& V(\text{wrt}) \quad ; \text{allow others} \\
\text{reader:} & \\
& P(\text{mutex}) \quad ; \text{ensure exclusion} \\
& \quad \text{readcount}++ \quad ; \text{one more reader} \\
& \quad \text{if readcount} == 1 \text{ then } P(\text{wrt}) \quad ; \text{if we’re the first, synch with writers} \\
& V(\text{mutex}) \quad \text{<perform read operation>} \\
& P(\text{mutex}) \quad ; \text{ensure exclusion} \\
& \quad \text{readcount}-- \quad ; \text{one fewer reader} \\
& \quad \text{if readcount} == 0 \text{ then } V(\text{wrt}) \quad ; \text{no more readers, allow a writer} \\
& V(\text{mutex})
\end{align*}
\]
Readers/Writers notes

- Notes:
  - the first reader blocks on P(wrt) if there is a writer
    - any other readers will then block on P(mutex)
  - if a waiting writer exists, the last reader to exit signals the waiting writer
    - can new readers get in while a writer is waiting?
  - when writer exits, if there is both a reader and writer waiting, which one goes next?
Semaphores vs. Locks

- Threads that are blocked at the level of program logic are placed on queues, rather than busy-waiting.

- Busy-waiting may be used for the “real” mutual exclusion required to implement P and V.
  - but these are very short critical sections – totally independent of program logic.

- In the not-very-interesting case of a thread package implemented in an address space “powered by” only a single kernel thread, it’s even easier that this.
Problems with semaphores (and locks)

• They can be used to solve any of the traditional synchronization problems, but:
  – semaphores are essentially shared global variables
    • can be accessed from anywhere (bad software engineering)
  – there is no connection between the semaphore and the data being controlled by it
  – used for both critical sections (mutual exclusion) and for coordination (scheduling)
  – no control over their use, no guarantee of proper usage

• Thus, they are prone to bugs
  – another (better?) approach: use programming language support
One More Approach: Monitors

• A *monitor* is a *programming language* construct that supports controlled access to shared data
  – synchronization code is added by the compiler
    • why does this help?

• A monitor encapsulates:
  – *shared data* structures
  – *procedures* that operate on the shared data
  – *synchronization* between concurrent threads that invoke those procedures

• Data can only be accessed from within the monitor, using the provided procedures
  – protects the data from unstructured access

• Addresses the key usability issues that arise with semaphores
A monitor

- Waiting queue of threads trying to enter the monitor
- At most one thread in monitor at a time
- Shared data
- Operations (methods)
Monitor facilities

• “Automatic” mutual exclusion
  – only one thread can be executing inside at any time
    • thus, synchronization is implicitly associated with the monitor – it “comes for free”
  – if a second thread tries to execute a monitor procedure, it blocks until the first has left the monitor
    • more restrictive than semaphores
    • but easier to use (most of the time)

• But, there’s a problem…
Example: Bounded Buffer Scenario

- Buffer is empty
- Now what?
Example: Bounded Buffer Scenario

- Buffer is empty
- Now what?
Condition variables

• A place to wait; sometimes called a rendezvous point
• “Required” for monitors
  – So useful they’re often provided even when monitors aren’t available
• Three operations on condition variables
  – wait(c)
    • release monitor lock, so somebody else can get in
    • wait for somebody else to signal condition
    • thus, condition variables have associated wait queues
  – signal(c)
    • wake up at most one waiting thread
    • if no waiting threads, signal is lost
      – this is different than semaphores: no history!
  – broadcast(c)
    • wake up all waiting threads
Bounded buffer using (Hoare) monitors

Monitor bounded_buffer {
    buffer resources[N];
    condition not_full, not_empty;

    produce(resource x) {
        if (array “resources” is full, determined maybe by a count)
            wait(not_full);
        insert “x” in array “resources”
        signal(not_empty);
    }

    consume(resource *x) {
        if (array “resources” is empty, determined maybe by a count)
            wait(not_empty);
        *x = get resource from array “resources”
        signal(not_full);
    }
}
Runtime system calls for (Hoare) monitors

- EnterMonitor(m) \{guarantee mutual exclusion\}
- ExitMonitor(m) \{hit the road, letting someone else run\}
- Wait(c) \{step out until condition satisfied\}
- Signal(c) \{if someone’s waiting, step out and let him run\}
Bounded buffer using (Hoare) monitors

Monitor bounded_buffer {
    buffer resources[N];
    condition not_full, not_empty;

    procedure add_entry(resource x) {
        if (array "resources" is full, determined maybe by a count)
            wait(not_full);
        insert "x" in array "resources"
        signal(not_empty);
    }

    procedure get_entry(resource *x) {
        if (array "resources" is empty, determined maybe by a count)
            wait(not_empty);
        *x = get resource from array "resources"
        signal(not_full);
    }
}
There is a subtle issue with that code...

• Who runs when the signal() is done and there is a thread waiting on the condition variable?

• **Hoare monitors**: `signal(c)` means
  - run waiter immediately
  - signaller blocks immediately
    - condition guaranteed to hold when waiter runs
    - but, signaller must **restore monitor invariants** before signalling!
      - cannot leave a mess for the waiter, who will run immediately!

• **Mesa monitors**: `signal(c)` means
  - waiter is made ready, but the signaller continues
    - waiter runs when signaller leaves monitor (or waits)
  - signaller need not restore invariant until it leaves the monitor
  - being woken up is only a hint that something has changed
    - signalled condition may no longer hold
    - must recheck conditional case
Hoare vs. Mesa Monitors

• Hoare monitors: if (notReady) wait(c)

• Mesa monitors: while (notReady) wait(c)

• Mesa monitors easier to use
  – more efficient: fewer context switches
  – directly supports broadcast

• Hoare monitors leave less to chance
  – when wake up, condition guaranteed to be what you expect
Runtime system calls for Hoare monitors

• **EnterMonitor(m)** \{guarantee mutual exclusion\}
  – if m occupied, insert caller into queue m
  – else mark as occupied, insert caller into ready queue
  – choose somebody to run

• **ExitMonitor(m)** \{hit the road, letting someone else run\}
  – if queue m is empty, then mark m as unoccupied
  – else move a thread from queue m to the ready queue
  – insert caller in ready queue
  – choose someone to run
Runtime system calls for Hoare monitors (cont’d)

- **Wait(c)** \{step out until condition satisfied\}
  - if queue m is empty, then mark m as unoccupied
  - else move a thread from queue m to the ready queue
  - put the caller on queue c
  - choose someone to run

- **Signal(c)** \{if someone’s waiting, step out and let him run\}
  - if queue c is empty then put the caller on the ready queue
  - else move a thread from queue c to the ready queue, and put the caller into queue m
  - choose someone to run
Runtime system calls for Mesa monitors

- EnterMonitor(m) \{\textit{guarantee mutual exclusion}\}
  - ... 
- ExitMonitor(m) \{\textit{hit the road, letting someone else run}\}
  - ... 
- Wait(c) \{\textit{step out until condition satisfied}\}
  - ... 
- Signal(c) \{\textit{if someone’s waiting, give him a shot after I’m done}\}
  - if queue c is occupied, move one thread from queue c to queue m
  - return to caller
• Broadcast(c) \{food fight!\}
  – move all threads on queue c onto queue m
  – return to caller
Monitor Summary

• Language supports monitors
• Compiler understands them
  – compiler inserts calls to runtime routines for
    • monitor entry
    • monitor exit
    • signal
    • Wait
  – Language/object encapsulation ensures correctness
    • Sometimes! With conditions you STILL need to think about synchronization
• Runtime system implements these routines
  – moves threads on and off queues
  – ensures mutual exclusion!