Semaphores

- Semaphore = a synchronization primitive
  - higher level than locks
  - invented by Dijkstra in 1968, as part of the THE operating system
- A semaphore is:
  - a variable that is manipulated atomically through two operations, \( P(sem) \) (wait) and \( V(sem) \) (signal)
  - \( P \) and \( V \) are Dutch for "wait" and "signal"
  - Plus, you get to say stuff like "the thread p's on the semaphore"
  - \( P(sem) \): block until \( sem > 0 \), then subtract 1 from \( sem \) and proceed
  - \( V(sem) \): add 1 to \( sem \)

Two types of semaphores

- **Binary semaphore (aka mutex semaphore)**
  - guarantees mutually exclusive access to resource (e.g., a critical section of code)
  - only one thread allowed entry at a time
  - \( sem \) is initialized to 1
- **Counting semaphore**
  - represents a resource with many units available
  - allows threads to enter as long as more units are available
  - \( sem \) is initialized to \( N \)
    - \( N \) = number of units available
- We'll mostly focus on binary semaphores

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

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" (\( \leq 0 \)), place thread on associated queue, run some other 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)
    - if no threads are waiting on the associated queue, increment \( sem \)
    - the signal is "remembered" for next time \( P(sem) \) is called
    - might as well let the "V-ing" thread continue execution

Implementation

- \( P(sem) \)
  - acquire "real" mutual exclusion
  - if \( sem \) was "available" (\( >0 \)), decrement \( sem \)
  - release "real" mutual exclusion; let thread continue
  - if \( sem \) was "unavailable" (\( \leq 0 \)), place thread on associated queue and release "real" mutual exclusion; run some other thread
- When \( V(sem) \) is called by a thread
  - 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
  - might as well let the "V-ing" thread continue execution
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 finite size $N$ entries
  – a producer thread inserts entries into it
  – a consumer thread removes entries from it
• Threads are concurrent
  – so, we must use synchronization constructs to control access to shared variables describing buffer state

Example: Bounded buffer problem (both binary and counting)

```
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 spared you a repeat of the clip-art!
Note 2: I have elided all the code concerning which is the first full buffer, which is the last full buffer, etc.
Note 3: Try to figure out how to do this without using counting semaphores!

Example: Readers/Writers
• Basic problem:
  – object is shared among several processes
  – some read from it
  – others write to it
• We can allow multiple readers at a time
  – why?
• We can only allow one writer at a time
  – why?

Readers/Writers using semaphores

```
var mutex: semaphore ; controls access to readcount
clear: semaphore ; control entry for a writer or first reader
readcount: integer ; number of active readers

writer:
P(clear) ; any writers or readers?
<perform write operation>
V(clear) ; allow others

reader:
P(mutex) ; ensure exclusion
readcount = readcount + 1 ; one more reader
if readcount = 1 then P(clear) ; if we're the first, sync with writers
V(mutex) ; <perform reading>
P(mutex) ; ensure exclusion
readcount = readcount – 1 ; one fewer reader
if readcount = 0 then V(clear) ; no more readers, allow a writer
V(mutex)
```

Readers/Writers notes
• Note:
  – the first reader blocks if there is a writer
  – any other writers will then block on mutex
  – if a waiting writer exists, last reader to exit signals waiting writer
  – can new readers get in while writer is waiting?
  – when writer exits, if there is both a reader and writer waiting, which one goes next is up to scheduler
Semaphores vs. locks

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

- 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
  - a better approach: use programming language support

Monitors

- A monitor is a software module that 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
  - protects the data from unstructured access
- Synchronization code (calls to synchronization routines in the thread package) is added by compiler
  - why does this help?
- Addresses the key usability issues that arise with semaphores

Monitor facilities

- "Automatic" mutual exclusion
  - only one thread can be executing inside at any time
    - thus, synchronization "comes for free" with monitor
  - if a second thread tries to execute a monitor procedure, it blocks until the first has left the monitor
- Condition variables
  - once inside, a thread may discover it can’t continue, and may wish to block (or allow some other waiting thread to continue)
    - it can wait on a condition variable, or signal others to continue
      - condition variables can only be accessed from within monitor
      - a thread that waits "steps outside" the monitor (onto a wait queue associated with that condition variable)
      - what happens to a thread that signals depends on the precise monitor semantics that are used – "Hoare" vs. "Mesa"

Condition variables

- A place to wait; sometimes called a rendezvous point
- 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
    - (ignore for now)
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);
}
}

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

• 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

Two kinds of monitors: Hoare and Mesa

• Hoare monitors: signal(c) means
– run water immediately
– signaler blocks immediately
  • condition guaranteed to hold when waiter runs
  • but, signaler 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 signaler continues
  • waiter runs when signaler leaves monitor (or waits)
  • signaler need not restore invariant until it leaves the monitor
  – being woken up is only a hint that something has changed
  • must recheck conditional case
• Hoare monitors
  – if (notReady)
  • wait(c)
• Mesa monitors
  – while(notReady)
  • wait(c)

• Mesa monitors easier to use
  – more efficient
  – fewer switches
  – directly supports broadcast
• Hoare monitors leave less to chance
  – when wake up, condition guaranteed to be what you expect

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) \textit{(food fight!)}
  – move all threads on queue c onto queue m
  – return to caller

Summary
• Language supports monitors
• Compiler understands them
  – compiler inserts calls to runtime routines for
    • monitor entry
    • monitor exit
    • signal
    • wait
• Runtime system implements these routines
  – moves threads on and off queues
  – \textit{ensures mutual exclusion!}