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 “wait” and “signal”)
    • P(sem) (wait)
      – block until sem > 0, then subtract 1 from sem and proceed
    • V(sem) (signal)
      – 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; 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
      – might as well let the “V-ing” thread continue execution
    • otherwise (when no threads are waiting on the sem), increment sem
      – the signal is “remembered” for next time P(sem) is called

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
  – Logically equivalent to a lock with blocking rather than spinning

• Counting semaphore
  – Allow up to N threads continue (we’ll see why in a bit …)
  – 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

Binary semaphore 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

Example: Bounded buffer problem

• AKA “producer/consumer” problem
  – there is a circular buffer in memory with N entries (slots)
  – 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 slots (all empty to start)
full: semaphore = 0         ; count of full slots (none full to start)

producer:
P(empty) ; block if no slots available
P(mutex) ; get access to pointers
<add item to slot, adjust pointers>
V(mutex) ; done with pointers
V(full)      ; note one more full slot

consumer:
P(full)      ; wait until there's a full slot
P(mutex) ; get access to pointers
<remove item from slot, adjust pointers>
V(mutex) ; done with pointers
V(empty) ; note there's an empty slot
```

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

```plaintext
var mutex: semaphore = 1 ; controls access to readcount
wrt: semaphore = 1 ; control entry for a writer or first reader
readcount: integer = 0 ; number of active readers

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

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

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?
      • so?
  – when writer exits, if there is both a reader and writer waiting,
    which one goes next?

Semaphores vs. Spinlocks

• Threads that are blocked at the level of program logic (that is, by
  the semaphore P operation) 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
  – and they are not implemented by the application programmer

Abstract implementation

• P(wait(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(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(signal(sem)) is called
  • release "real" mutual exclusion
  • [the "V-ing" thread continues execution, or may be preempted]
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 before manipulating shared state?
- Could locks be implemented in exactly the same way? That is, “software locks” that you acquire and release, where the underlying implementation involves moving descriptors to/from a wait queue?

Condition Variables

- Basic operations
  - Wait()
  - Wait until some thread does a signal and release the associated lock, as an atomic operation
  - Signal()
  - If any threads are waiting, wake up one
  - Cannot proceed until lock re-acquired
- Signal() is not remembered
  - A signal to a condition variable that has no threads waiting is a no-op
- Qualitative use guideline
  - You wait() when you can’t proceed until some shared state changes
  - You signal() when shared state changes from “bad” to “good”

Bounded buffers with condition variables

- producer:
  - lock(mutex)
  - if [no slots available] wait(freeslot);
  - [add item to slot, adjust pointers]
  - signal(fullslot);
  - unlock(mutex)

- consumer:
  - lock(mutex)
  - if [no slots have data] wait(fullslot);
  - [remove item from slot, adjust pointers]
  - signal(freeslot);
  - unlock(mutex);
  - [use the item]

The possible bug

- Depending on the implementation …
  - Between the time a thread is woken up by signal() and the time it re-acquires the lock, the condition it is waiting for may be false again
  - Waiting for a thread to put something in the buffer
  - A thread does, and signals
  - Now another thread comes along and consumes it
  - Then the “signalled” thread forges ahead …
- Solution
  - Not
    - if [no slots available] wait(fullslot)
  - Instead
    - While [no slots available] wait(fullslot)
  - Could the scheduler also solve this problem?

Problems with semaphores, locks, and condition variables

- They can be used to solve any of the traditional synchronization problems, but it’s easy to make mistakes
  - they are essentially shared global variables
  - can be accessed from anywhere (bad software engineering)
  - there is no connection between the synchronization variable and the data being controlled by it
  - No control over their use, no guarantee of proper usage
    - Condition variables: will there ever be a signal?
    - Semaphores: will there ever be a V()?
    - Locks: did you lock when necessary? Unlock at the right time? At all?
- Thus, they are prone to bugs
  - We can reduce the chance of bugs by “stylizing” the use of synchronization
  - Language help is useful for this

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 is (essentially) a class in which every method automatically acquires a lock on entry, and releases it on exit – it combines:
    - shared data structures (object)
    - procedures that operate on the shared data (object methods)
    - 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
    - Prevents ambiguity about what the synchronization variable protects
  - 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

Proc A

Proc B

Proc C

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...

Problem: Bounded Buffer Scenario

• Buffer is empty
  • Now what?

Solution?

• Monitors require condition variables
• Operations on condition variables (just as before!)
  – 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
    – “Hoare” monitor: wakeup immediately, signaller steps outside
    • if no waiting threads, signal is lost
    – this is different than semaphores: no history!
  – broadcast(c)
    • wake up all waiting threads

Problem: Bounded Buffer Scenario

• Buffer is full
  • Now what?

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);
        resource “x” = get resource from array “resources”
        signal(not_full);
    }
}
**Problem: Bounded Buffer Scenario**

- Buffer is full
- Now what?

**Bounded Buffer Scenario with CV’s**

- Buffer is full
- Now what?

**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)

- EnterMonitor and ExitMonitor are inserted automatically by the compiler.
- This guarantees mutual exclusion for code inside of the monitor.

**Bounded buffer using (Hoare) monitors**

```c
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 signal/leave 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
- **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)** (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, 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

Monitors and Java

- Java offers something a bit like monitors
  - It should be clear that they’re not monitors in the full sense!
- Every Java object contains an intrinsic lock
- The `synchronized` keyword locks that lock
- Can be applied to methods, or blocks of statements
Synchronized methods

- Atomic integer is a commonly provided (or built) package
- `public class atomicInt {
   int value;
   public atomicInt(int initVal) {
      value = initVal;
   }
   public synchronized postIncrement() {
      return value++;
   }
   public synchronized postDecrement() {
      return value--;
   }
   ...
}

Monitor Summary

- Language supports monitors
- Compiler understands them
  - Compiler inserts calls to runtime routines for
    - monitor entry
    - monitor exit
  - Programmer inserts calls to runtime routines for
    - 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!