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
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: 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 notes

• Note:
  – the first reader blocks if there is a writer
  – any other readers will then block on mutex
  – if a waiting writer exists, last reader to exit signals waiting writer
  – 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 programming language construct that supports controlled access to shared data
  – synchronization code added by compiler, enforced at runtime
  – why does this help?
• 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
• Monitor protects the data from unstructured access
  – guarantees data can only be accessed through procedures, hence in legitimate ways

A monitor

Monitor facilities

• Mutual exclusion
  – only one thread can be executing inside at any time
  – thus, synchronization implicitly associated with monitor
  – if a second thread tries to execute a monitor procedure, it blocks until the first has left the monitor
• Once inside, a thread may discover it can’t continue, and may wish to sleep
  – or, allow some other waiting process to continue
  – condition variables provided within monitor
    • threads can wait, or can signal others to continue
    • condition variables can only be accessed from within monitor
    • a thread that waits “steps outside” the monitor
    • what happens to a thread that signals depends on the precise monitor semantics that are used

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
  – signal(c)
    • wake up at most one waiting thread
    • if no waiting threads, signal is lost
  – broadcast(c)
    • wake up all waiting threads
Bounded buffer using monitors

Monitor bounded_buffer {
    buffer resources[N];
    condition not_full, not_empty;
    procedure add_entry(resource x) {
        while(array "resources" is full)
            wait(not_full);
        insert "x" in array "resources"
        signal(not_empty);
    }
    procedure get_entry(resource *x) {
        while (array "resources" is empty)
            wait(not_empty);
        *x = get resource from array "resources"
        signal(not_full);
    }
}

Two kinds of monitors

- Hoare monitors: signal(c) means
  - run waiter immediately
  - signaler blocks immediately
  - condition guaranteed to hold when waiter runs
    - can use "if" rather than "while" in previous example
  - but, signaler must restore monitor invariants before signaling!

- Mesa monitors: signal(c) means
  - waiter is made ready, but the signaler continues
  - waiter runs when signaler leaves monitor (or waits)
  - condition is not necessarily true when waiter runs again
    - must use "while" as in previous example
  - 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 vs. Mesa

- 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

A monitor is a language feature

- 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
  - ensures mutual exclusion!