CSE 451: Operating Systems
Winter 2003

Lecture 8
Semaphores and Monitors

Hank Levy
levy@cs.washington.edu
412 Sieg Hall
Semaphores

• semaphore = a synchronization primitive
  – higher level than locks
  – invented by Dijkstra in 1968, as part of the THE os

• A semaphore is:
  – a variable that is manipulated atomically through two operations, signal and wait
  – wait(semaphore): decrement, block until semaphore is open
    • also called P(), after Dutch word for test, also called down()
  – signal(semaphore): increment, allow another to enter
    • also called V(), after Dutch word for increment, also called up()
 Blocking in Semaphores

• Each semaphore has an associated queue of processes/threads
  – when wait() is called by a thread,
    • if semaphore is “available”, thread continues
    • if semaphore is “unavailable”, thread blocks, waits on queue
  – signal() opens the semaphore
    • if thread(s) are waiting on a queue, one thread is unblocked
    • if no threads are on the queue, the signal is remembered for next time a wait() is called

• In other words, semaphore has history
  – this history is a counter
    – if counter falls below 0 (after decrement), then the semaphore is closed
      • wait decrements counter
      • signal increments counter
Hypothetical Implementation

type semaphore = record
    value: integer;
    L: list of processes;
end

wait(S):
    S.value = S.value - 1;
    if S.value < 0
        then begin
            add this process to S.L;
            block;
        end;

signal(S):
    S.value = S.value + 1;
    if S.value <= 0
        then begin
            remove a process P from S.L;
            wakeup P
        end;

wait()/signal() are critical sections!
Hence, they must be executed atomically with respect to each other.
Two types of semaphores

• **Binary** semaphore (aka mutex semaphore)
  – guarantees mutually exclusive access to resource
  – only one thread/process allowed entry at a time
  – counter is initialized to 1

• **Counting** semaphore (aka counted semaphore)
  – represents a resources with many units available
  – allows threads/process to enter as long as more units are available
  – counter is initialized to N
    • N = number of units available
Example: bounded buffer problem

• AKA producer/consumer problem
  – there is a buffer in memory
    • with finite size $N$ entries
  – a producer process inserts an entry into it
  – a consumer process removes an entry from it

• Processes are concurrent
  – so, we must use synchronization constructs to control
    access to shared variables describing buffer state
Bounded Buffer using Semaphores

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

consumer:
wait(full) ; wait until there’s a full buffer
wait(mutex) ; get access to pointers
  <remove item from buffer>
signal(mutex) ; done with pointers
signal(empty) ; note there’s an empty buffer
  <use the item>
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
wrt: semaphore ; control entry to a writer or first reader
readcount: integer ; number of readers

write process:
  wait(wrt) ; any writers or readers?
  <perform write operation>
  signal(wrt) ; allow others

read process:
  wait(mutex) ; ensure exclusion
  readcount = readcount + 1 ; one more reader
  if readcount = 1 then wait(wrt) ; if we’re the first, synch with writers
  signal(mutex)
  <perform reading>
  wait(mutex) ; ensure exclusion
  readcount = readcount - 1 ; one fewer reader
  if readcount = 0 then signal(wrt) ; no more readers, allow a writer
  signal(mutex)
Readers/Writers notes

• Note:
  – the first reader blocks if there is a writer
    • any other readers will then block on mutex
  – if a 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
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
  – another (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 processes that invoke those procedures
• Monitor protects the data from unstructured access
  – guarantees only access data through procedures, hence in legitimate ways
A monitor

- A waiting queue of processes trying to enter the monitor.
- At most one process in the monitor at a time.
- Operations (procedures) and shared data.
Monitor facilities

• Mutual exclusion
  – only one process can be executing inside at any time
    • thus, synchronization implicitly associated with monitor
  – if a second process tries to enter a monitor procedure, it
    blocks until the first has left the monitor
    • more restrictive than semaphores!
    • but easier to use most of the time

• Once inside, a process may discover it can’t
  continue, and may wish to sleep
  – or, allow some other waiting process to continue
  – condition variables provided within monitor
    • processes can wait or signal others to continue
    • condition variable can only be accessed from inside monitor
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 wait queues
  – signal(c)
    • wake up at most one waiting process/thread
    • if no waiting processes, signal is lost
    • this is different than semaphores: no history!
  – broadcast(c)
    • wake up all waiting processes/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);
        add "x" to 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
  – signaller blocks immediately
    • condition guaranteed to hold when waiter runs
    • but, signaller must restore monitor invariants before signalling!

• Mesa monitors: signal(c) means
  – waiter is made ready, but the signaller continues
    • waiter runs when signaller leaves monitor (or waits)
    • condition is not necessarily true when waiter runs again
  – signaller need not restore invariant until it leaves the monitor
  – being woken up is only a hint that something has changed
    • must recheck conditional case
Examples

• 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
Condition Variables and Mutex

• Yet another construct:
  – condition variables can be used with mutexes

    pthread_mutex_t mu;
    pthread_cond_t co;
    boolean ready;
    void foo( ) {
      pthread_mutex_lock(&mu);
      if (!ready)
        pthread_cond_wait(&co, &mu);
      ...
      ready = TRUE;
      pthread_cond_signal(&co);  // unlock and signal atomically
      pthread_mutex_unlock(&mu);
    }

• Think of a monitor as a language feature
  – under the covers, compiler knows about monitors
  – compiler inserts a mutex to control entry and exit of processes to
    the monitor’s procedures