Semaphores

Semaphores are a synchronization primitive that offers a higher level of abstraction than locks. They were invented by Ed Dijkstra in 1968 as part of the THE operating system.

A semaphore is a variable that can be manipulated through two operations: P (for "wait") and V (for "signal").

- **P(sem)**
  - Block until sem > 0, then subtract 1 from sem and proceed.

- **V(sem)**
  - 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 and place it on the ready queue.
  - If no threads are waiting on the associated queue, sem is incremented.
  - The signal is "remembered" for next time P(sem) is called.

Semaphores thus have history.

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 and 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-ring" thread continues execution or is preempted]
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)
  – 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 do a P before manipulating shared state?
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

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?

Bounded buffer using semaphores (both binary and counting)

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!

Readers/Writers using semaphores
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?
  - does this cause any problems?
  - when writer exits, if there is both a reader and writer waiting,
    which one goes next?

Semaphores vs. Locks

- Threads that are blocked 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
  
- 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 than 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

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 full
  • 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
  - \( \text{wait}(c) \)
    - release monitor lock, so somebody else can get in
    - wait for somebody else to signal condition
    - thus, condition variables have associated wait queues
  - \( \text{signal}(c) \)
    - wake up at most one waiting thread
    - if no waiting threads, signal is lost
    - this is different than semaphores: no history!
  - \( \text{broadcast}(c) \)
    - wake up all waiting threads

A monitor (including CVs)

Bounded buffer using (Hoare) monitors

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

  produce(resource x) {
    if (array "resources" is full)
      wait(not_full);
    insert "x" in array "resources"
    signal(not_empty);
  }

  consume(resource *x) {
    if (array "resources" is empty)
      wait(not_empty);
    *x = get resource from array "resources"
    signal(not_full);
  }
}

Readers and Writers
(stolen from Cornell ☺)

Monitor ReadersNWriters {
  int WaitingWriters, WaitingReaders, NReaders, NWriters;
  Condition CanRead, CanWrite;

  Void BeginWrite() {
    if(NWriters == 1 || NReaders > 0)
      ++WaitingWriters;
      wait(CanWrite);
      --WaitingWriters;
  }

  Void EndWrite() {
    NWriters = 0;
    if(WaitingReaders)
      Signal(CanRead);
    else
      Signal(CanWrite);
  }

  Void BeginRead() {
    if(NWriters == 1 || WaitingWriters > 0)
      ++WaitingReaders;
      wait(CanRead);
      --WaitingReaders;
  }

  Void EndRead() {
    if(--NReaders == 0)
      Signal(CanWrite);
  }
}
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

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);
  }
}

Question: who runs when the signal() is executed and there is a thread waiting on the condition variable?

Hoare monitors: signal(c) means
- run waiter 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
- signalled condition may no longer hold
- must re-check conditional case

There are two kinds of Monitors

- Hoare monitors: if (notReady) wait(c)
- Mesa monitors: while (notReady) wait(c)

Hoare vs. Mesa Monitors

- Hoare 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 c is empty, then mark c as unoccupied
  – else move a thread from queue c 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 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) (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)
  – ...

• 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!