CSE 451: Operating Systems
Spring 2011

Module 7
Semaphores and Monitors

John Zahorjan
zahorjan@cs.washington.edu
Allen Center 534
Last Time: Locks

- `acquire()`/`release()` operations
  - In complicated code, can be hard to get right
    - Some implementations provide “recursive locks”
    - Some implementations complain if one acquires, another releases
      - Some applications rely on one thread acquiring, another releasing

- Come in spinning and blocking varieties
  - Spin if you expect a short wait and there are multiple cores
  - Block if only one CPU/core or you expect long waits

- Blocking involves an update to a queue, i.e., a critical section
  - So, still need spin locks, if just to implement blocking locks
This Time: Other Synchronization Primitivies

- (Synchronization is a way of putting happens-before arcs into the thread graph)

- **Semaphores**
  - A generalization of blocking locks

- **Condition variables**
  - A way to wait for an event (while in a critical section)

- **Monitors**
  - Language (or convention)-based way to never forget to lock or unlock

- **Barriers**
  - Synchronize n threads in a single statement

- **Join**
  - Wait for a thread to terminate
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/down)
      - block until sem > 0, then subtract 1 from sem and proceed
    - **V(sem)** (signal/up)
      - 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 blocking lock

• **Counting semaphore**
  – Let N threads into “critical section,” not just one
    • Why? We'll see in a minute...
  – 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
Example: Bounded buffer problem

• AKA “producer/consumer” problem
  – there is a circular 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
Bounded buffer using semaphores (both binary and counting)

```plaintext
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 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!
**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

```
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?
  – when writer exits, if there is both a reader and writer
    waiting, which one goes next?
Semaphores vs. Locks

• Threads that are blocked at the level of program logic 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 application logic
Condition Variables

- **Basic operations:**
  - `Wait()`
    - wait until some thread does a signal AND release a lock, as an atomic operation
  - `Signal()`
    - if any threads are waiting, wake up one
    - (broadcast(): wake them all up)

- **`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()` whenever shared state changes from “bad” to “good”
Bounded-buffers with condition variables

```plaintext
var mutex: lock ; mutual exclusion to shared data
freeslot: condition ; there's a free slot
fullslot: condition ; there's a full slot

producer:
lock(mutex) ; get access to pointers
If ( buffer is full ) wait(freeslot);
  <add item to buffer>
signal(fullslot);
unlock(mutex) ; done with pointers

consumer:
lock(mutex) ; wait until there's a full buffer
If ( buffer is empty ) wait(fullslot) ; get access to pointers
  <remove item from buffer>
signal(freeslot);
unlock(mutex) ; use the item
```

Note: There is a subtle bug in this code!
The 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
    • The woken thread makes a mistake...

• NOT if (buffer is empty) wait(fullslot)

• INSTEAD while (buffer is empty) wait(fullslot)
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're 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 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
    - The restrictions of the style may lead to inefficiencies, however
  - Often language help is useful for this
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:
  – *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
Monitors Require Condition Variables

- Buffer is empty
- Now what?
Monitors and Java

• Monitors are a somewhat exotic language feature

• Java offers something a tiny bit like monitors
  – It should be clear to you that they're not monitors in the full sense at all!

• Every Java object contains an intrinsic lock

• The `sychronized` 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--;
  }
  ...
}
Synchronized Statements

• You can lock any Object, and have the lock automatically released when you leave the block of statements

• void foo(ArrayList list) {
  ...
  synchronized(list) {
    <manipulate the list>
  }
}
Barriers

• Sometimes you want (all) N threads to wait until they've all reached a synchronization point

• Example: NxM matrix vector multiply: \( C = AB \)
  
  ```c
  for (i=0; i<N; i++) {
      C[i] = 0;
      for ( j=0; j<M; j++) {
          C[i] += A[i][j] * B[j];
      }
  }
  ```

  Wait here until all threads have finished
As threaded code

- `barrier_init( multBarrier, N+1);`
  ```c
  for (i=0; i<N; i++) {
    thread_start( vectorMultiply, A, B, C, i, M);
  }
  barrier_wait(multBarrier);
  ```

(The italicized names are not the pthread names...)

- `void vectorMultiply(A,B,C,i,M) {
  C[i] = 0;
  for (j=0; j<M; j++) C[i] += A[i][j] * B[j];
  barrier_wait(multBarrier);
  ```
Join

• Sometimes you want to wait until a thread has terminated
  – That's what join() is for

• A common use:
  – Start N threads
  – Sit in a loop waiting for thread 1, then thread 2, then ...
    • It really doesn't matter much which one finishes first, you just wait in an arbitrary order

• Note: This is not quite the same as using a barrier
  – join() waits until threads have terminated, and so given up all their resources
  – A barrier is achieved before threads have terminated
Summary (in pictures)

lock()

unlock()

lock()

unlock()

Lock synchronization
Semaphore Synchronization

signal() or

wait() or

wait() or

signal()
Condition variable synchronization

- `signal()`
- `lock()`
- `Wait() [unlock()]`
- `[lock()]"
Barrier Synchronization

```
barrier()
barrier()
barrier()
```
Join Synchronization

join() → exit()

join() → exit()

join() → exit()