CS377: Operating Systems and Systems Programming
Lecture 7a
Semaphores

Prof. Deepak Ganesan
http://www.cs.umass.edu/~dganesan/cs377
Goals for Today

• Continue with Synchronization Abstractions
  – Monitors and condition variables
• Readers-Writers problem and solution

Note: Some slides and/or pictures in the following are adapted from slides ©2005 Silberschatz, Galvin, and Gagne. Many slides generated from lecture notes by Kubiatowicz.
Where are we going with synchronization?

<table>
<thead>
<tr>
<th>Programs</th>
<th>Shared Programs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Higher-level API</td>
<td></td>
</tr>
<tr>
<td>Hardware</td>
<td>Load/Store  Disable Ints  Test&amp;Set Comp&amp;Swap</td>
</tr>
</tbody>
</table>

- We are going to implement various higher-level synchronization primitives using atomic operations
  - Everything is pretty painful if only atomic primitives are load and store
  - Need to provide primitives useful at user-level
Semaphores

• Semaphores are a kind of generalized locks
  – First defined by Dijkstra in late 60s
  – Main synchronization primitive used in original UNIX

• Definition: a Semaphore has a non-negative integer value and supports the following two operations:
  – P (or Down or Wait): an atomic operation that waits for semaphore to become positive, then decrements it by 1
  – V (or Up or Signal): an atomic operation that increments the semaphore by 1, waking up a waiting P, if any
  – Analogy: Think about semaphore value as the number of empty chairs at a table…
Semaphores Like Integers Except

- Semaphores are like integers, except
  - No negative values
  - Only operations allowed are P and V – can’t read or write value, except to set it initially
  - Operations must be atomic
    - Two P’s together can’t decrement value below zero
    - Similarly, thread going to sleep in P won’t miss wakeup from V – even if they both happen at same time
Two uses of semaphores

• Mutual Exclusion (initial value = 1)
  – Also called “Binary Semaphore”.
  – Can be used for mutual exclusion:
    ```
    semaphore.P();
    // Critical section goes here
    semaphore.V();
    ```

• Scheduling constraint (e.g. initial value > 1)
Two uses of semaphores

• Signaling events (initial value = 0)
  – Example: suppose you had to implement ThreadJoin which must wait for thread to terminate:
    
    Initial value of semaphore = 0
    ThreadJoin {
        semaphore.P();
    }
    ThreadFinish {
        semaphore.V();
    }

```
Interrupt
ISR

binary semaphore

Task

```
Producer-consumer with a bounded buffer

- **Problem Definition**
  - Producer puts things into a shared buffer
  - Consumer takes them out
  - Need synchronization to coordinate producer/consumer

- Don’t want producer and consumer to have to work in lockstep, so put a fixed-size buffer between them
  - Need to synchronize access to this buffer
  - Producer needs to wait if buffer is full
  - Consumer needs to wait if buffer is empty

- **Example: Coke machine**
  - Producer can put limited number of cokes in machine
  - Consumer can’t take cokes out if machine is empty
Semaphores: notes

• Notice: shared object (coke machine) separate from threads (delivery person, students, faculty). Shared object coordinates activity of threads.

• Common confusion when using semaphores
  – students often try to do the synchronization within the threads’ code.
  – No, the synchronization happens within the shared objects. “Let the shared objects do the work.”
Correctness constraints

- Synchronization problems have semaphores represent two types of constraint
  - mutual exclusions
  - wait for some event

- When you start working on a synchronization problem, first define the mutual exclusion constraints, then ask “when does a thread wait”, and create a separate synchronization variable representing each constraint
Correctness constraints for solution (v1)

• Correctness Constraints:
  – Consumer must wait for producer to fill slots, if empty (scheduling constraint)
  – Producer must wait for consumer to make room in buffer, if all full (scheduling constraint)

• General rule of thumb:
  Use a separate semaphore for each constraint
  – Semaphore fullSlots; // consumer’s constraint
  – Semaphore emptySlots; // producer’s constraint
Solution to Bounded Buffer (v1)

Semaphore fullSlots = 0; // Initially, no coke
Semaphore emptySlots = bufSize;
    // Initially, num empty slots

Producer(item) {
    emptySlots.P();        // Wait until space
    insert item into buffer;
    fullSlots.V();        // Tell consumers there is
                         // more coke
}

Consumer() {
    fullSlots.P();        // Check if there’s a coke
    get item from buffer;
    emptySlots.V();     // tell producer need more
    return item;
}
Correctness constraints for solution

• Correctness Constraints:
  – Consumer must wait for producer to fill slots, if empty (scheduling constraint)
  – Producer must wait for consumer to make room in buffer, if all full (scheduling constraint)
  – Only one thread can manipulate buffer queue at a time (mutual exclusion)

• General rule of thumb:
  Use a separate semaphore for each constraint
  – Semaphore fullSlots; // consumer’s constraint
  – Semaphore emptySlots; // producer’s constraint
  – Semaphore mutex; // mutual exclusion
Full Solution to Bounded Buffer

Semaphore fullSlots = 0;  // Initially, no coke
Semaphore emptySlots = bufSize;  
                           // Initially, num empty slots
Semaphore mutex = 1;  // No one using machine

Producer(item) {
    emptySlots.P();  // Wait until space
    mutex.P();  // Wait until slot free
    Enqueue(item);
    mutex.V();
    fullSlots.V();  // Tell consumers there is
                   // more coke
}

Consumer() {
    fullSlots.P();  // Check if there’s a coke
    mutex.P();  // Wait until machine free
    item = Dequeue();
    mutex.V();
    emptySlots.V();  // tell producer need more
    return item;
}
Discussion about Solution

Semaphore fullSlots = 0;
Semaphore emptySlots = bufSize;
Semaphore mutex = 1;

Producer(item) {
    emptySlots.P();
    mutex.P();
    Enqueue(item);
    mutex.V();
    fullSlots.V();
}

Consumer() {
    fullSlots.P();
    mutex.P();
    item = Dequeue();
    mutex.V();
    emptySlots.V();
    return item;
}

• Is order of P’s important?
  – Yes! Can cause deadlock
• Is order of V’s important?
  – No, except that it might affect scheduling efficiency
• What if we have 2 producers or 2 consumers?
  – Do we need to change anything?
Solution for Bounded Buffer using Locks only

```java
int fullSlots = 0; // Initially, no coke
Lock lock = free; // No one using machine

Producer(item) {
    lock.Acquire();
    if (fullSlots == bufSize) {
        lock.Release();
        return false; // No room for coke
    }
    Enqueue(item); // add new coke
    fullSlots++;
    lock.Release();
    return true;
}

Consumer() {
    lock.Acquire();
    if (fullSlots == 0) {
        lock.Release();
        return null; // no coke
    }
    item = Dequeue(); // get coke
    fullSlots--;
    lock.Release();
    return item;
}
```
Semaphore implementation (interrupts)

typedef struct {
    int count;
    queue q; /* queue of threads waiting on this semaphore */
} Semaphore;

void P(Semaphore s)
{
    Disable interrupts;
    if (s->count > 0) {
        s->count -= 1;
        Enable interrupts;
        return;
    }
    Add(s->q, current thread);
    sleep & enable int();
}

void V(Semaphore s)
{
    Disable interrupts;
    if (isEmpty(s->q)) {
        s->count += 1;
    } else {
        thread = RemoveFirst(s->q);
        wakeup(thread); /* ready q */
    }
    Enable interrupts;
}
Semaphore implementation (test&set)

typedef struct {
    int count, lock;
    queue q; /* queue of threads waiting on this semaphore */
} Semaphore;

void P(Semaphore s)
{
    while (TAS(&lock, 1) != 0);
    if (s->count > 0) {
        s->count -= 1;
        Enable interrupts;
        return;
    }
    Add(s->q, current thread);
    sleep & set lock=0;
}

void V(Semaphore s)
{
    while (TAS(&lock, 1) != 0);
    if (isEmpty(s->q)) {
        s->count += 1;
    } else {
        thread = RemoveFirst(s->q);
        wakeup(thread);
    }
    lock=0;
}
Summary: Semaphores

• Synchronization problems have two types of constraints
  – mutual exclusions
  – wait for some event

• Semaphores: Like integers with restricted interface
  – Two operations:
    » \( P() \) : Wait if zero; decrement when becomes non-zero
    » \( V() \) : Increment and wake a sleeping task (if exists)
    » Can initialize value to any non-negative value
  – Use separate semaphore for each constraint

• Two uses of semaphores
  – Mutual Exclusion (initial value = 1)
  – Signaling/Scheduling constraint (initial value = 0)