CS 537
Lecture 7
Semaphores

Michael Swift

Locking Review

- Locking can be done by:
  - Software spin locks (Peterson's algorithm)
  - Hardware spin locks (test and set)
  - Disabling interrupts
  - Which is best, when?
- Locks protect **shared** variables

```
func(int * x)    // x may be global
  int y, z;
  y = *x + 2;
  z = y*5
  return z;
```

  - Locks protect access to *x, not y and z

Motivation for Semaphores

- Locks only provide mutual exclusion
  - Ensure only one thread is in critical section at a time
- May want more: Place ordering on scheduling of threads
  - Example: Producer/Consumer
    - Producer: Creates a resource (data)
    - Consumer: Uses a resource (data)
  - Example
    - ps | grep "gcc" | wc
  - Don't want producers and consumers to operate in lock step
    - Place a fixed-size buffer between producers and consumers
    - Synchronize accesses to buffer
    - Producer waits if buffer full; consumer waits if buffer empty

Semaphores

- semaphore = a synchronization primitive
  - higher level than locks
  - invented by Dijkstra in 1965, 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()
  - Plus sem_init(counter) to set first counter value

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
        add this process to S.L;
        block;

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

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

Semaphore Example

- What happens if sem is initialized to 2?
  - Scenario: Three processes call sem_wait(&sem)
- Observations
  - Sem value is negative --> Number of waiters on queue
  - Sem value is positive --> Number of threads that can be in c.s. at same time

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

Mutual Exclusion with Semaphores

- Previous example with locks:

```
Void deposit (int amount) {
  mutex_lock(&mylock);
  balance += amount;
  mutex_unlock(&mylocak);
}
```

- Example with semaphores:

```
Void deposit(int amount) {
  wait(&sem);
  balance += amount;
  signal(&sem);
}
```

- To what value should sem be initialized???

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

Producer/Consumer: Single Buffer

- Simplest case:
  - Single producer thread, single consumer thread
  - Single shared buffer between producer and consumer
- Requirements
  - Consumer must wait for producer to fill buffer
  - Producer must wait for consumer to empty buffer (if filled)
- Requires 3 semaphores
  - emptyBuffer: Initialize to ???
  - fullBuffer: Initialize to ???
  - mutex: Initialize to ???

Producer

```
While (1) {
       wait(&emptyBuffer);
       wait(&mutex);
       Fill(&buffer);
       signal(&mutex);
       signal(&fullBuffer);
}
```

Consumer

```
While (1) {
       wait(&fullBuffer);
       wait(&mutex);
       Use(&buffer);
       signal(&mutex);
       signal(&emptyBuffer);
}
```


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

```
semaphore mutex  ; controls access to readcount
semaphore wrt    ; control entry to a writer or first reader
int readcount    ; 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) 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) 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