COMS 414 - Prelim 1 Review Session

Vivek Vishnumurthy
vivi@cs.cornell.edu
Slides adapted from
Yejin Choi and
Daniel Williams(2003fa TA's)

< Processes & Threads >

- program V.S. process ?
- process V.S. thread ?
- kernel-level threads V.S. user-level threads ?

Multitasking (and multithreading…)

- What is a **system call**?
- What is a **context switch**?
- What **states** may a process be in?
- What is a **race condition**?
- What is **PCB**?

- Multiprocessor & MultiThreading…

< Synchronization >

- **Critical section**
- **Semaphore**
- **Monitor**

Critical Section

- **Critical section**
  1. **Entry section**
     {
     2. **Critical section**
     }
  3. **Exit section**
  4. **Remainder section**
- **Critical section design requirements**
  - **1. Mutual Exclusion**
  - **2. Progress**
  - **3. Bounded Waiting**

Semaphore

- **Synchronization solutions**
  - **Spinlock(busy waiting)**
  - **Counting semaphore/ Binary semaphore**
- **Semaphore**
- **Implementation Issues**
  - **Deadlock**
  - **Starvation**

Semaphore Primitives Implementation

Busy waiting :

```
Wait(S) {
    while(S<=0) ; // no-op
    S--;
    }

Signal(S) {
    S++;
    }
```

Block & Wakeup from CPU scheduler:

```
Wait(S) {
    S--;
    if( S<0 ) { block( );}
    }

Signal(S) {
    S++;
    if( S<=0 ) { wakeup( );}
    }
```

Synchronization

- Three levels of abstraction for concurrent programming:
  - Hardware instructions
    - Atomic HW instruction 'test_and_set'
  - O/S primitives
    - Semaphores
  - Programming language constructs
    - Monitors

Monitors

- **monitor**
- **condition variables**
- **condition variables** V.S. **semaphores**
  - Condition variable signal( )
    … resumes exactly only one suspended process
    … if no process suspended, no effect at all
  - Semaphore signal( )
    … always affect the state of the semaphore
    … by increasing resource counter

Sample Monitor Code (Readers&Writers from Class)

```
class ReadersNWriters:: monitor{
    condition readQ, writeQ;
    int nReadersInside = 0, nWritersInside = 0;
    int nReadersWaiting = 0, nWritersWaiting = 0;

    public void StartRead(){
        if(nWritersInside > 0 || nWritersWaiting > 0) {
                ++nReadersWaiting;
                //readers line up here if writers active/waiting
                readQ.wait();
                --nReadersWaiting;
        }
        ++nReadersInside;
        // once one reader gets in, this lets them all in
        readQ.signal();
    }

    public void EndRead(){
        if(--nReadersInside == 0)
                // if any writer is waiting the last reader lets it go
                writeQ.signal();
    }
```

Sample Monitor Code (Readers&Writers from Class)

```
    public void StartWrite()
    {
        if(nReadersInside > 0 || nWritersInside > 0) {
                ++nWritersWaiting;
                // wait if there is an active writer or reader
                writeQ.wait();
                --nWritersWaiting;
        }
        ++nWritersInside;
    }

    public void EndWrite()
    {
        --nWritersInside;
        // wake a reader if one is waiting, else a writer
        if(nReadersWaiting > 0)
                readQ.signal();
        else
                // a no-op if nobody is waiting
                writeQ.signal();
    }
}
```

Classic Synchronization Problems

- Bounded buffer problem
- Readers writers problem
- Dining philosophers problems

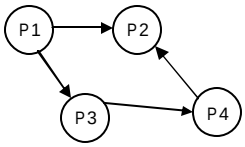
< Deadlock >

- Deadlock V.S. Livelock
- Necessary conditions
  - Mutual Exclusion, Hold and Wait, No Preemption, Circular Wait
- Resource Allocation Graph / Wait-For Graph
- Deadlock Prevention
  - Ensure one of 'necessary conditions' do not hold
- Deadlock Avoidance
  - Safe State, Resource-Allocation Graph Algorithm, Banker's Algorithms

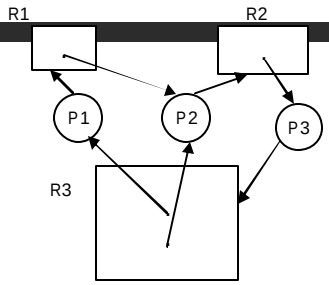
Livelock (/li:v'lok/ from www.hyperdictionary.com...)

- When two or more processes continuously change their state in response to changes in the other process(es) without doing any useful work.
- This is similar to deadlock in that no progress is made but differs in that neither process is blocked or waiting for anything.
- A human example of livelock would be two people who meet face-to-face in a corridor and each moves aside to let the other pass, but they end up swaying from side to side without making any progress because they always move the same way at the same time.

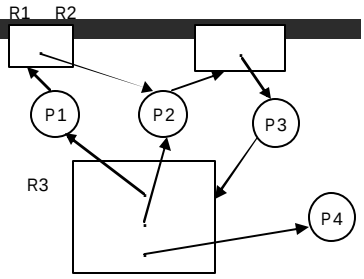
Wait-For Graph



Resource-allocation graph #1



Resource-allocation graph #2



Deadlock Prevention

Make sure that one of the necessary conditions does not hold:

1. **Mutual exclusion**
2. **Hold and Wait**
3. **No Pre-Emption**
4. **Circular Wait**

< How to Prepare Prelim >

- Make sure to review homework problem sets.
- Practice writing synchronization code on your own.
- Rather than reading every single line of the text book, find out key ideas and topics, and try to explain them in your own words.

- http://www.cs.cornell.edu/Courses/cs414/**2003fa**/
- http://www.cs.cornell.edu/Courses/cs414/**2002fa**/

Practice #1 – Mutual Exclusion using Test&Set (from Text)

```
bool lock = false; //this is global-shared among all processes

CS_ENTER{
    waiting[i] = true;
    bool key = true;
    while(waiting[i] && key){
          key = Test_And_Set(lock);
    }
    waiting[i] = false;
}
CSLeave{
    int j=(i+1)%n;
    while(j!=i && !waiting[j]){
          j= (j+1)%n;
    }
    if(j==i)
          //No process wants to get in!
          lock = false;
    else
          waiting[j] = false;
}
```

Provides Mutual Exclusion, Progress, and Bounded Waiting!

Synchronization Practice #2

HW3 – Problem 3

Due to last minute funding cuts, Ellison University's dorms only have a single shared bathroom on each floor, even though the dorms are coed. Using semaphores, solve the unisex bathroom problem. Specifically, design procedures **GuyEnters(), GuyLeaves(), GirlEnters(), GirlLeaves()** such that: (a) there are never more than 3 people in the bathroom, (b) if a guy is in, girls can't enter and vice- versa, and (c) If a guy is waiting and girls are inside, the next person to get in will be a guy, and vice versa (all of these 3 properties should hold at the same time).

Synchronization Practice #2 - Answer

```
Semaphore Mutex = 1, TheLine = 1, OpenTheDoor = 1, RoomLimit = 3;
Integer GuyCount = 0, GirlCount = 0;

GuyEnters() {
    // "separate" logic to respect room capacity limit
    wait(RoomLimit);
    wait(TheLine);
    // Only one person gets here at a time! Others line up on "TheLine"
    wait(Mutex);
    // Mutex needed because of people leaving
    if(GuyCount++ == 0 || GirlCount > 0) {
          signal(Mutex);
          // If we get here, the bathroom is currently empty (or will be soon)
          // Notice that while waiting on this semaphore anyone else who shows
          // up gets stuck waiting on TheLine and won't get past that spot until
          // this person (who could be a guy or a girl, since this section of
          // the code looks the same in both cases) is actually in the bathroom.
          wait(OpenTheDoor);
    }else
          signal(Mutex);
    // Now we can let another person get off TheLine
    signal(TheLine);
}
```

Synchronization Practice #2 – Answer…

```
GuyLeaves()
{
    signal(RoomLimit);
    wait(Mutex);
    // Think of this next line as "last person out closes the door"
    if(--GuyCount == 0)
          signal(OpenTheDoor);
    signal(Mutex);
}
```

Last Verdict…