Lecture 6

March 2nd 2023

HW #3 Due Wed 3/8/2023

HW #4 Due Wed 3/22/2023

HW #5 Due Wed 3/29/2023

spring 3/13 - 3/17

Midterm will be on March-30-2023

Review on March 23rd

Sample Midterm will be posted.

Chapter 6 SynchronizationTools

Process synchronization

Review

OS is an application that manages other applications (Processes). It also manage hardware of the computer in such a way it is done in a secure and fair manner.

Hardware resources such as the Processor, Main Memory, Disk storage done in a very efficient manner that is, we would like the OS to serve as many processes as possible per unit time.

However, process do not run in silos. They share resources and they cooperate to fullfill certain task. This implies that OS must allow all sorts of traffic. If so, OS allowing multi traffic, it must also impose some kind of synchronization and Concurrency Handling.

Example:

Write a multithread Process that increments a single variable.

```
int tally = 0;                // shared  static
void main()
{
  tally = 0
  parabegin(total(), total());
  write(tally)
}

void total()
{
    tally = tally + 1;        // critical section /shared section
}
```

static: There is only one copy of the variable per class. So you can have as many objects as you would like, but there is only one copy of that. So all the objects you can instantiate as many as you would like, but there is only one location in memory called tally, that's accessible by all.

This tally is shared by both, so we need some kind of synchronization.

Q: What the answer of the example?

A: 0 , 1, 2

```
public class Main extends Thread {
  public static int tally = 0;  // static: shareable by everybody
  public static void main(String[] args){
    while (tally != 1)
    {
      tally = 0;
      Main thread1 = new Main();
      Main thread2 = new Main();
      thread1.start();
      thread2.start();
      thread1.stop();
      thread2.stop();
    }
    System.out.println("This code is outside of the thread and tally = " + tally);
  }

  public void run(){
    tally=tally+1;
    System.out.println("This code is running in a thread id " +
Thread.currentThread().getId());
  }
}
```

We want to impose process synchronization, the answer always must be `tally = 2` irrespective of the process speed.

Hence we must impose process synchronization. Must find a solution. But how do we know the solution is correct? Must have a criteria , what is it? Any solutions must satisfy the following conditions.

1. **Mutual exclusion**

   one and only one process can be in the critical section at any given time.

2. **Progress**

   progresses that are waiting to enter the critical section are the only processes that decide who goes next.

3. **Bounded Waiting**

   a process must wait only a limited time before it is granted access to the critical section.

1, 2, 3 The speed of the CPU or the process has no effect whatsoever on who goes next.

Nobody waits forever means nobody starves.

1. **What is the problem?**

   Exclusion access to critical section

2. **Under what conditions?**

   (1) Mutual Exclusion

   (2) Progress

   (3) Bounded waiting

3. **How to enforce them 1. and 2. both?**

   Three ways

   - **User Level**

     We have to program it.   (The most difficult)

     => Peterson Algorithm

   - **Hardware Level**

     Build hardware instructions that enforce it.

     => Example: disable interrupts

   - **Programming Level**

     Provide programming language constructs that enforce it.

     => Java, C# provide two very powerful programming constructs: semaphores, Monitors.

     semaphores: **fine** control of synchronization [complex]

     Monitor: broad synchronization

     Q: Which one is the most widely used of these two?

     A: Monitor. Because it is easier, you don't have to worry about the complex things. And the price of a little bit of losing efficiency. Who cares. It's just easier.

```
void Total()
{
  /*  protect it */
  tally = tall + 1;      /* control section */
}
```

We are going to code the protection.

Q: How?

A: Two threads or two process accessing tally at the same time. p0, p1

P0:

```
void Total()
{ while (true)
    {
      flag[0] = true;      /* p0 is asking to go next */
      turn = 1      /* courteous to p1 */
      while(flag[1] && turn == 1)
         {
            wait();               /* therefore it cannot be 1 and 0 at the same time */
          }                       /* somebody is gonna override the value of the other
*/
      /* critical section */
      tally = tally + 1;
      /* end of critical section */
      flag[0] = false;
    }
}
```

P1:

```
void Total()
{
    while (true)
    {
        flag[1] = true;      /* p1 wants to be first */
        turn = 0;              /* but it's being courteous to p0 */
        while (flag[0] and turn == 0)
        {
            wait();
        }
        /* critical section */
        tally = tally + 1;
        /* end of critical seciton */
        flag[1] = false;
    }
}
```

Q: Are we guaranteed that no matter the value of total = 2?

Q: Will this guarantee Mutual exclusion?

A: Yes

Bit Operation

x, y

XOR: one but the other

$x \bigoplus y = xy' + yx'$

$x = y = xy + x'y'$

1: true 0:false

p1	p0	shared		
flag[1]	flag[0]	turn	Is it possible	(who gets access)
0	0	0	x (because one of flags has to be true)	
0	0	1	x (same reason as above)	
0	1	0	x (p1:flag[1] = 0 means p1 has already executed its critical section, which means the turn must be 1)	
0	1	1	√ (p0:flag[0] = 1 means p0 go next, for p1)	p1 is done, p0 is waiting
1	0	0	√ (p1:flag[1] =1 means p1 wants to go, turn could be 0 because it's going to set it to basically 0)	p0 done, p1 is waiting
1	0	1	x (code flag[1] = true; turn = 0;)	
1	1	0	√	p0 is in, p1 is waiting
1	1	1	√	p1 is in, p0 is waiting

University is beautiful because of symmetric.

```
Do
{
    entry condition
    critical section

    exit condition
} while true
```

The algorithm that we just implemented is called **Peterson** algorithm

```
boolean flag [2];
int turn;
void p0()
{
    while (true){
        flag[0] = true;
        turn = 1;
        while (flag [1] && turn == 1) /* do nothing */
        /* critical section */;
        flag [0] = false;
        /* remainder */;
    }
}

void p1()
{
    while (true){
        flag [1] = true;
        turn = 0;
        while (flag[0] && turn == 0) /* do nothing */
        /* critical section */;
        flag [1] = false;
        /* remainder */
    }
}
```

f[1]	f[0]	Turn	
0	0	0	Impossible
0	0	1	Impossible
0	1	0	Impossible
0	1	1	p0 w, p1 done
1	0	0	p0 done, p1 w
1	0	1	Impossible
1	1	0	p0, p1 waiting
1	1	1	p1, p0 waiting

Three ways to enforce synchronization

1. **user level**

   above we just did it.

   only for theoretical purposes

2. **hardware level**

   has limited use

3. **programming language constructs**

   most popular

2, 3 Reading assignments: suffice to say that one simple solution is to disable all interrupts using hardware mechanism before entering a critical section and enable interrupts once done. (These works only on uniprocessor)

3 Programming Language Constructs

- Semaphore:

  Two types:

  - Binary Semaphore
  - Counting Semaphore

  Q: What is semaphore?

  A: A German word like he said in Spanish it's basically a traffic light it's like basically a switch on or off. But once you put your finger on that switch and you try to turn it on. Nobody can interrupt you in the middle of it advisor so once you put your finger on that switch the OS is not going to be able to interrupt you. So it is a none it is an atomic operation even though it could be multi-steps.

  Binary Semaphore and Counting Semaphore are atomic operation even though they can be multi-step. Cannot be interrupted. OS will enforce the non-interruption.

  Q: Binary Semaphore, counting Semaphore which one you think will be used to enforce Mutual exclusion, like to shield or to the a critical section problem?

Definition of Semaphore: An integer value used for signaling among processes. Only three operations maybe performed on a semaphore all of which are atomic :

1. Initialize
2. Increment
3. Decrement. These decrement operation may result in the blocking of a process, and the increment operation may result in the unblocking of a process.

Binary Semaphore:

a semaphore that takes on only the value of 0 or 1 (analogy to a light switch).

In this case Binary semaphore are very appealing to solving the critical section problem.

How does a semaphore look like:

Binary semaphore:

```
value: {0, 1}
queue: Queue(list)

void SemWait(binary Semaphore s)
{
    if (s.value == 1) then
        s.value == 0
    else{
        1. palce the process in s.queue
        2. block this process
    }
}

void SemSignalB(Semaphore s)
{
    if (s.queue is empty())
        s.value = 1
    else
    {
        1. remove a process from s.queue;
        2. place the process on the ready list
    }
}/* end of the Binary Semaphore Definition */
```

Solution the critical section using Binary Semaphores

P0:

```
p0
Semaphore s = 1;
while (true)
{
    semwait(s)
    tally = tally + 1
    SemSignal(s)
}

p1:
while(true)
{
    semwait(s)
    tally = tally + 1;
    SemSignal(s)
}
```