Lecture 7

03.09.2023

Synchronization

Chapter 6 Process Synchronization

Important: 6.1, 6.2, 6.3, 6.6, 6.7

Read: 6.4, 6.5, 6.10

Skip: 6.8, 6.9

Chapter 7

Important: 7.1, 7.4,

Read: 7.2, 7.3, 7.5

Chapter 8

Important: 8.1, 8.2, 8.3, 8.4, 8.5, 8.6, 8.7

Read: 8.8, 8.9

HW #3

1. will be posting on Canvas sample Java program as how to do the Hw3
2. might adjust Hw3 due date

write a class CPU for CPU simulation.

```
public class CPU{
  public static void run(Task task, int slice){
    system.out.println("will run task", task);
  }
}
```

For this Hw, you will need to write the following classes.

1. `FCFS.Java`
2. `SJF.Java`
3. `RR.Java`

4. `Priority.Java`
5. `Priority.Java`

Each of these classes must implement the `Algorithm.java` interface (which will be provided)

`Algorithm.java`:

```
public interface Algorithm{
  public abstract void schedule()

  public abstract Task pickNexttask()
}
```

`Task.java`:

```
public class Task{
  private int burst;
  private int priority
  private int tid;
  private string name;
}
```

`Driver.java`

```
public class Driver{
  public static void main(String[] args){
    /* Usage Java Driver FCFS [schedule]
    /* T1 4 20
    /* T2 3 26
    /* T4 5 15 */
    /* Reading the input */
    queue[] = T1, 4, 7  // -> Task Object

  }
}
```

Queue is a list Task.

```
List <Task> queue = new ArrayList<Task>
  queue = [T1, 4, 2,
          T2, 3, 5,
          ....
          ]
```

Initialization of tasks:

- hand code / read from a file

```
string choice = args[0].toUppearCase();
switch(choice)
  case "FCFS" =
    schedule = new FCFS(queue);
    break
  case "SJF" =
    schedule = new SJF(queue);
    break
      ...
```

1. Write a Driver that will create a list of Tasks where these tasks are either hand-code in the program or read from a file whose name is passes as an argument in the command line.
2. Read the scheduling algorithm from the command line where the scheduling algorithm is one of the following:

   FCFS, SJF, PRI, RR, PRI-RR

   Must create a small class called Task that has Task name, Task id, priority, CPU burst. We also need a CPU simulation. A class for Task, a class for CPU, a class for FCFS, a class for SJF, a class for the main Driver which has the main method. (FCFS, SJF, ..., must implemented two methods found in Algorithm interface.)

`FCFS`

```
public class FCFS implements Algorithm{
  private list<Task> queue
    FCFS(list<Task> queue){
    queue =
  }

  public void schedule(){
    for each (Task t in queue)
      CPU.run(t);
  }
}
```

The Dept has assigned a grader for this class. Will send the grader infor via Canvas.

Synchronization

Process Synchronization:

Why? We decided to design OS that allows traffic in all directions. That is, OS allows processes to run concurrently and share common resources. Trade-off is that process may get deadlock and may be modify shared resources at the same time. Hence OS must impose some kind of Synchronization must be fair to all processes irrespective of their priority. To do so the OS must address the critical section problem and also must synchronize the use of the limited or finite shared resources.

How?

1. User level: **Peterson Algorithm**
2. Handware Level: **Handware instructions**
3. Programming language Level: **Programming concepts**
    - **Semaphores**
        - Binary Semaphores

          is used to address the critical section problem

          (talk this last time, we use to solve the CS problem)
        - Counting Semaphores

          to manage the finite and limited resources.
    - Monitors

Counting Semaphores

A binary semaphore is integer variable of 0,1 along with two operations

`SemWait(s)`, `SemSigned(s)`

On the other hand a counting Semaphores is an integer variable s that can be with a value of n and two operations associated with t `SemWait(s)` and `SemSigned(s)`.

We are done with binary Semaphore. How, examine counting Semaphore.

Example: Producer / Consumer problem:

They are one or more producers generating some type of data and placing these in a buffer. There is a single consumer that is taking items out of the buffer one at a time.

Goal: is to make sure to prevent overlap of buffer operations. That's only one agent (producer or the consumer may accessthe buffer at any one time. The chanllege is to make sure that the producer will not try to add data into the buffer if it is full and the consumer will not worry to try remove data from the buffer when it is empty.

Producer:

```
while(true)
{
  /* add item */
  b[in] = item           -------> shared
    in = in + 1
}
```

Consumer:

```
While (true)
{
  while (in <= out)
    wait;

  w = b[out]                     ---------> shared
  out = out + 1
  consume(w)
}
```

shared, since it is shared, we have critical sections. We have **many** producers and one consumer.

safe guard the buffer

- Binary Semaphores

  complex: because we are using a very simple construct which is simple
- Counting Semaphores

  less complex: because this counting semaphores are very sophisticated

Program

```
/* Goal to solve the producer Consumer problem using Binary Semaphores */
int n;
binary-semaphores s = 1;
binary-semaphores delay = 0;
```

Producer

```
/* multiple producers */
while (true)
{
  produce(); /* produce a new item */
  SemWait(s)       ----------> May I have a permission to enter the critical section
  append();  /* add to the buffer */
  n = n + 1; /* keep track # of item in buffer */

  SemSigned(s)   -----> Thank you so much, I am leaving, that's what the critical
section is.
}
```

Consumer

```
{
    SemWait(delay) /* make sure buffer is not empty */
    take() /* Consume somehting from a buffer */
}
```