BSS Practicum Week 2 Exercises

> - Andre Nanninga
> - 6-may-2014

Chapter 5

> *1.1 Explain why the term process scheduling is actually incorrect. (5.1)*

The Process Scheduler is actually scheduling threads of a process. The correct term should be Thread Scheduling

> *1.2 Explain the difference between a CPU scheduler and a dispatcher. (5.4)*

The CPU scheduler determines what process should be run next. The dispatcher will perform the task of switching to that process.

> *1.3 Suppose the following processes arrive for execution at the times indicated. Each process will run for the amount of time listed. In answering the questions, use non-preemptive scheduling, and base all decisions on the information you have at the time the decision must be made. (5.7)*
>
> | Process | Arrival time | Burst Time |
> |---|---|---|
> | P1 | 0.0 | 7 |
> | P2 | 0.5 | 3 |
> | P3 | 1.0 | 2 |
>
> Scheduling criteria, see page 197
>
> - Arrival Time (AT): The first time of submission of a process
> - Burst Time (BT): The time a process runs on a CPU
> - Turnaround Time (TT): The interval from the time of submission of a process to the time

> of completion
> - The sum of the periods spent waiting to get into memory, waiting in the ready queue, executing on the CPU, and doing I/O
> - Waiting Time (WT): The sum of the periods spent spent in the ready queue
>
> *A. ATT with FCFS*
>
> *What is the average turnaround time (ATT) for these processes with the First Come First Serve (FCFS) scheduling algorithm?*

Process	Arrival*	Start*	Finish*	Waiting time	Processing time	Total time
P1	0.0	0.0	7.0	0.0s	7.0s	7.0s
P2	0.5	7.0	10.0	6.5s	3.0s	9.5s
P3	1.0	10.0	12.0	9.0s	2.0s	11.0s
Average						**9.166...s**

Arrival, Start and Finish depict the time since startup

P1 will run first for a total execution time of *7.0 seconds*. During this time P2 and P3 have arrived and will be executed following the FCFS scheduling algorithm. P2 will start running at *7.0 seconds* after startup and it has waited for *6.5 seconds* by now. The total execution time of P2 is *9.5 seconds*. P3 will run after this and will has waited *9.0 seconds* and has a total execution time of *11.0 seconds*.

On average the execution time of the three processes is *9.166... seconds*.

> *B. ATT with SJF*
>
> *What is the average turnaround time (ATT) for these processes with the Shortest Job First (SJB) scheduling algorithm?*

Process	Arrival*	Start*	Finish*	Waiting time	Processing time	Total time
P1	0.0	0.0	7.0	0.0s	7.0s	7.0s
P2	0.5	9.0	13.0	8.5s	3.0s	11.5s
P3	1.0	7.0	9.0	6.0s	2.0s	8.0s

Average						**8.833...s**

P1 will run first for a total execution time of *7.0 seconds*. During this time both P2 and P3 have arrived and will be executed following the SJB scheduling algoritm. P3 is the short job waiting and will be executed first after P1 is finished. After *6.0 seconds* of waiting and *2.0 seconds* of execution P3 has a total execution time of *8.0 seconds*. P2 will run last after waiting *8.5 seconds* and running for *3.0 seconds* having a total execution time of *11.5 seconds*.

On average the execution time of the three processes is *8.833... seconds*.

> *C. ATT with SJB and idle*
>
> *The SJB algorithm is supposed to improve performance, but notice that we chose to run process P1 at time 0 because we did not know that two shorter processes would arrive soon. Compute what the average turnaround time will be if the CPU is left idle for the first 1 unit and then SJF scheduling is used. Remember that processes P1 and P2 are waiting during this idle time, so their waiting time may increase. This algorithm could be known as future-knowledge scheduling.*

Process	Arrival*	Start*	Finish*	Waiting time	Processing time	Total time
P1	0.0	6.0	13.0	6.0s	7.0s	13.0s
P2	0.5	3.0	6.0	2.5s	3.0s	5.5s
P3	1.0	1.0	3.0	0.0s	2.0s	2.0s
Average						**6.833...s**

Waiting for *1.0 seconds* before any execution means that all three process are present when starting. The shortest job, P3, will be executed first without any waiting time and a processing time of *2.0 seconds* making a total execution time of *2.0 seconds*. P2 will run after this after waiting for *2.5 seconds* with a processing time of *3.0 seconds* making a total execution time of *5.5 seconds*. Finally P1 will run after waiting *6.0 seconds* with a processing time of *7.0 seconds* totalling an execution time of *13.0 seconds*.

On average the execution time of the three processes is *6.833... seconds*.

> *1.4 Explain the term CPU burst. (5.3)*

An interval of time in which the CPU is processing a process.

> *1.5 Consider the following set of processes, with the length of the CPU burst given in milliseconds (5.8)*
>
> | Process | Burst Time | Priority |
> |---|---|---|
> | P1 | 8 | 3 |
> | P2 | 1 | 1 |
> | P3 | 3 | 3 |
> | P4 | 1 | 4 |
> | P5 | 5 | 2 |
>
> The process assumed to have arrived in the order P1, P2, P3, P4, P5, all at time 0.
>
> *A. Draw four Gantt charts that illustrate the execution of these processes using the following scheduling algorithms: FCFS, SJF, non-preemptive priority (a smaller priority number implies a higher priority) and RR (quantum = 1)*

FCFS

Process	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
P1	■	■	■	■	■	■	■	■										
P2									■									
P3										■	■	■						
P4													■					
P5														■	■	■	■	■

SJF

Process	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
P2	■																	
		■																

P4		■																
P3			■	■	■													
P5						■	■	■	■	■								
P1											■	■	■	■	■	■	■	■

Non-preemptive Priority

Process	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
P2	■																	
P5		■	■	■	■	■												
P1							■	■	■	■	■	■	■	■				
P3															■	■	■	
P4																		■

Round Robin (quantum 1)

Process	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
P1	■					■			■			■		■		■	■	■
P2		■																
P3			■				■			■								
P4				■														
P5					■			■			■		■		■			

> *B. What is the turnaround time of each process for each of the scheduling algorithms in part a?*

	FCFS	SJF	Non-preemptive Priority	Round Robin
P1	8	18	14	18

P2	9	1	1	2
P3	12	5	17	10
P4	13	2	18	4
P5	18	10	6	15

> *C. What is the waiting time of each process for each of these scheduling algorithms?*

	FCFS	SJF	Non-preemptive Priority	Round Robin
P1	0	10	6	10
P2	8	0	0	1
P3	9	2	14	7
P4	12	1	17	3
P5	13	5	1	10
Average	**8.4**	**3.6**	**7.6**	**6.2**

> *D. Which of the algorithms results in the minimum average waiting time (over all the processes)?*

The best average is SJF with *3.6*.

> *1.7 Which of the following scheduling algorithms could result in starvation? (5.9)*

Non-preemptive Priority, a process with a low priority could starve when new processes with higher priority are inserted before the low priority process has a chance to execute.

Chapter 6

> *2.1 Consider the two general approaches to handle critical sections in operating systems. Discuss the favor for the preemptive approach and the difficulties with SMP architectures. (6.1)*

With the use of preemptive scheduling you can more easier ensure that no process is starved and that every process is given some processing time in a timely matter. For the end user having feedback on a process is very important, seeing a process slowly but steadily progressing is favored above a process that does nothing for a long time and in one burst completes it's process.

> *2.2 Explain the differences between a counting semaphore and a binary semaphore. (6.2)*

A binary semaphore only takes two values, one to represent that the shared resource is occupied and one to represent that the shared resource is available. A counting semaphore can hold n values. This n represents how many threads can access a shared resources simulanteously.

> *2.3 Describe how a program can overcome the need for busy waiting. (6.3)*

A process block itself instead of busy waiting. This would put the process in the waiting queue associated with the semaphore.

Programming excersise

> *3.1 Servers can be designed to limit the number of open connections. For example, a server may wish to have only N socket connections open at any point in time. After N connections have been made, the server will not accept another incoming connection until an existing connection is released. In the source code available on Wiley PLUS, there is a program named TimedServer.java that listens to port 2500. When a connection is made (via telnet or the supplied client program TimedClient.java), the server creates a new thread that maintains the connection for 10 seconds (writing the number of seconds remaining while the connection remains open). At the end of 10 seconds, the thread closes the connection. Currently, TimedServer.java will accept an unlimited number of connections. Using semaphores, modify this program so that it limits the number of concurrent connections. (6.4)*

TimedServer.java

```
import java.net.*;
import java.io.*;
import java.util.concurrent.*;

public class TimedServer {

  public static final int PORT = 2500;
```

```
  public static final int LIMIT = 2;

  public static void main(String[] args){
    try {
      ServerSocket server = new ServerSocket(PORT);
      Semaphore semaphore = new Semaphore(LIMIT);

      while(true) {
        semaphore.acquire();

        Socket socket = server.accept();

        Thread worker = new Thread(new Worker(socket, semaphore));
        worker.start();
      }
    }
    catch(Exception e) {
      System.err.println(e.getMessage());
      System.exit(0);
    }
  }
}
```

Worker.java

```
import java.net.*;
import java.io.*;
import java.util.concurrent.*;

public class Worker implements Runnable {
  private int timer = 10;

  private Socket socket;
  private Semaphore semaphore;

  public Worker(Socket socket, Semaphore semaphore) {
    this.socket = socket;
    this.semaphore = semaphore;
  }

  public void run() {
    try {
      PrintWriter writer = new PrintWriter(socket.getOutputStream(), true);

      while(timer > 0) {
        writer.println("Sleeping for " + timer + " more second(s).");
```

```
        Thread.sleep(1000);

        timer--;
      }
    }
    catch(Exception e) {
      System.err.println(e.getMessage());
    }
    finally {
      semaphore.release();

      try {
        socket.close();
      }
      catch(Exception e) {
        System.err.println(e.getMessage());
      }
    }
  }
}
```

> *3.2 Assume that a finite number of resources of a single resource type must be managed. Processes may ask for a number of these resources and — once finished — will return them. As an example, many commercial software packages provide a given number of licenses , indicating the number of applications that may run concurrently. When the application is started, the license count is decremented. When the application is terminated, the license count is incremented. If all licenses are in use, requests to start the application are denied. Such requests will only be granted when an existing license holder terminates the application and a license is returned.The following Java class is used to manage a finite number of instances of an available resource. Note that when a process wishes to obtain a number of resources, it invokes the decreaseCount() method. Similarly, when a process wants to return a number of resources, it calls increaseCount().*
>
> *a. Identify the data involved in the race condition.*

The integer `availableResources`. When this is data is not synchronized between threads the value may not be up to date when one thread causes this variable to be decreased.

> *b. Identify the location (or locations) in the code where the race condition occurs.*

In both the `decreaseCount` and `increaseCount` can a race condition occur. Any modification on a shared unsynchronized variable can be the cause of a race condition. Both these methods do a modification on the variable `availableResources`.

> *c. Using Java synchronization, fix the race condition. Also modify decreaseCount() so that a thread blocks if there aren't sufficient resources available.*

```java
public class Manager {
  public static final int MAX_RESOURCES = 5;
  private Integer availableResources = MAX_RESOURCES;

  // Decrease availableResources by count resources.
  // return 0 if sufficient resources available,
  // otherwise return -1
  public int decreaseCount(int count) {
    synchronized(availableResources) {
      try {
        if(availableResources < count) {
          availableResources.wait();
        }

        availableResources -= count;
      }
      catch(Exception e) { }

      return 0;
    }
  }

  /* Increase availableResources by count resources. */
  public void increaseCount(int count) {
    synchronized(availableResources) {
      availableResources += count;

      try {
        availableResources.notify();
      }
      catch(Exception e) { }
    }
  }
}
```