2013 - 2014 Thema 2.1 ITSD

Practicum Week 2 Exercises

The number between parenthesis at the end of each question is a reference for the lecturer.

Exercises

Chapter 5

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

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

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 Fist Come First Server (FCFS) scheduling algorithm?

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

c) ATT with SJB and idle
The SJF 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.

Example table

P	AT	0	0,5	1	2	3	4	5	6	7	8	9	10	11	12	13	14	BT	WT	TT
P1																				
P2																				
P3																				
Pn																				
																Total		0,00	0,00	0,00
																ATT		0,00	0,00	**0,00**

1.4 Explain the term CPU burst. (5.3)

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)
b) What is the turnaround time of each process for each of the scheduling algorithms in part a?
c) What is the waiting time of each process for each of these scheduling algorithms?
d) Which of the algorithms results in the minimum average waiting time (over all the processes)?

Example table

Example table TT of each process

TT	**FCFS**	**SJF**	**Prio**	**RR**
P1				
P2				
P3				
P4				
P5				
Total	0	0	0	0
Average	**0**	**0**	**0**	**0**

Example table WT of each process

WT	**FCFS**	**SJF**	**Prio**	**RR**
P1				
P2				
P3				
P4				
P5				
Total	0	0	0	0
Average	**0**	**0**	**0**	**0**

1.7 Which of t he following scheduling algorithms could result in starvation? (5.9)
a. First-come, first-served
b. Shortest job first
c. Round robin
d. Priority

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)

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

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

Programming Exercise

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)

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() .

```
public class Manager {
    public static final int MAX RESOURCES = 5;
    private int availableResources = MAX RESOURCES;

   /**
    * Decrease availableResources by count resources.
    * return 0 if sufficient resources available,
    * otherwise return -1
    */

   public int decreaseCount(int count) {
        if (availableResources < count) return -1;
        else {
            availableResources -= count;
            return 0;
    }

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

However, the preceding program segment produces a race condition. Do the following:
a. Identify the data involved in the race condition.
b. Identify the location (or locations) in the code where the race condition occurs.
c. Using Java synchronization, fix the race condition. Also modify decreaseCount() so that a thread blocks if there aren’t sufficient resources available.