Lista 2

Chapter 5

5.2 A CPU-scheduling algorithm determines an order for the execution of its scheduled processes. Given *n* processes to be scheduled on one processor, how many different schedules are possible? Give a formula in terms of *n*.

Answer(book): n! (n factorial = n × n – 1 × n – 2 × ... × 2 × 1).

5.3 Consider a system running ten I/O-bound tasks and one CPU-bound task. Assume that the I/O-bound tasks issue an I/O operation once for every millisecond of CPU computing and that each I/0 operation takes 10 milliseconds to complete. Also assume that the context-switching overhead is 0.1 millisecond and that all processes are long-running tasks. Describe the CPU utilization for a round-robin scheduler when:

a)The time quantum is 1 millisecond

b) The time quantum is 10 milliseconds

Answer(book): (a) The time quantum is 1 millisecond: Irrespective of which process is scheduled, the scheduler incurs a 0.1 millisecond context-switching cost for every context-switch. This results in a CPU utilization of 1/1.1 * 100 = 91%. (b) The time quantum is 10 milliseconds: The I/O-bound tasks incur a context switch after using up only 1 millisecond of the time quantum. The time required to cycle through all the processes is therefore 10*1.1 + 10.1 (as each I/O-bound task executes for 1 millisecond and then incur the context switch task, whereas the CPU-bound task executes for 10 milliseconds before incurring a context switch). The CPU utilization is therefore 20/21.1 * 100 = 94%.

5.9 Which of the following scheduling algorithms could result in starvation?

1. First-come, first-served
2. Shortest job first
3. Round robin
4. Priority

Answer: Shortest job first and priority-based scheduling algorithms could result in starvation.

5.10 Suppose that a scheduling algorithm (at the level of short-term CPU scheduling) favors those processes that have used the least processor time in the recent past. Why will this algorithm favor I/O-bound programs and yet not permanently starve CPU-bound programs?

Answer: It will favor the I/O-bound programs because of the relatively short CPU burst request by them; however, the CPU-bound programs will not starve because the I/O-bound programs will relinquish the CPU relatively often to do their I/O.

5.12 Consider a variant of the RR scheduling algorithm in which the entries in the ready queue are pointers to the PCBs.

1. What would be the effect of putting two pointers to the same process in the ready queue?
2. What would be two major advantages and two disadvantages of this scheme?
3. How would you modify the basic RR algorithm to achieve the same effect without the duplicate pointers?

Answer:

1) In effect, that process will have increased its priority since by getting time more often it is receiving preferential treatment.
2) The advantage is that more important jobs could be given more time, in other words, higher priority in treatment. The consequence, of course, is that shorter jobs will suffer.
3) Alloc a longer amount of time to processes deserving higher priority, on other words, have two or more quantum possible in the RR scheme.

5.13 Consider the following set of processes, with the length of the CPU burst given in milliseconds:
Process Burst Time Priority ----

Process	Burst Time	Priority
P_1	10	3
P_2	1	1
P_3	2	3
P_4	1	4
P_5	5	2

The processes are assumed to have arrived in the order P1, P2, P3, P4, P5, all at time 0.

1. Draw four Gantt charts that illustrate the execution of these processes using the following scheduling algorithms: FCFS, SJF, nonpreemptive priority (a smaller priority number implies a higher priority), and RR (quantum= 1).
2. What is the turnaround time of each process for each of the scheduling algorithms in part a?
3. What is the waiting tim of each process for each of these scheduling algorithms?
4. Which of the algorithms results in the minimum average waiting time (over all processes)?

Answer: http://cs302.yolasite.com/resources/Assignment%202-modelanswer.pdf

5.14 The traditional UNIX scheduler enforces an inverse relationship between priority numbers and priorities: the higher the the lower the priority. The scheduler recalculates process priorities once per second using the following function:

Priority = (recent CPU usage *I* 2) + base

where base = 60 and *recent* CPU *usage* refers to a value indicating how often a process has used the CPU since priorities were last recalculated.

Assume that recent CPU usage for process P1 is 40, for process P2 is 18, and for process P3 is 10. What will be the new priorities for these three processes when priorities are recalculated? Based on this information, does the traditional UNIX scheduler raise or lower the relative priority of a CPU-bound process?

Answer: The priorities assigned to the processes are 80, 69, and 65 respectively. The scheduler lowers the relative priority of CPU-bound processes.

5.15 Discuss how the following pairs of scheduling criteria conflict in certain settings.

1. CPU utilization and response time
2. Average turnaround time and maximum waiting time
3. I/0 device utilization and CPU utilization

Answer:

1. CPU utilization and response time: CPU utilization is increased if the overheads associated with context switching is minimized. The context switching overheads could be lowered by performing context switches infrequently. This could, however, result in increasing the response time for processes.

2. Average turnaround time and maximum waiting time: Average turnaround time is minimized by executing the shortest tasks first. Such a scheduling policy could, however, starve long-running tasks and thereby increase their waiting time.

3. I/O device utilization and CPU utilization: CPU utilization is maximized by running long-running CPU-bound tasks without performing context switches. I/O device utilization is maximized by scheduling I/O-bound jobs as soon as they become ready to run, thereby incurring the overheads of context switches.

Chapter 6

6.9 Servers can be designed to limit the number of open connections. For example, a server may wish to have only *N* socket connections at any point in time. As soon as *N* connections are made, the server will not accept another incoming connection until an existing connection is released. Explain how semaphores can be used by a server to limit the number of concurrent connections.
Describe a mechanism for enforcing memory protection in order to prevent a program from modifying the memory associated with other programs.

Answer: Semaphores can be used by a server to limit the number of concurrent connections. This can be accomplished by initializing the semaphore 'availableConn' to a value 'N'. When a new connection is opened, the semaphore value should be decremented by executing wait (availableConn) to decrement the value of currently available connections. Similarly, when one connection is closed, then post (availableConn) can be executed indicating that there is one more socket available for a connection.

6.10 Why do Solaris, Lil1.ux, and Windows XP use spinlocks as a syn- chronization mechanism only on multiprocessor systems and not on single-processor systems?

Answer: Solaris, Linux, and Windows 2000 use spinlocks as a synchronization mechanism only on multiprocessor systems. Spinlocks are not appropriate for single-processor systems because the condition that would break a process/thread out of spinlock could be obtained only by executing a different process. If the process is not relinquishing the processor, other processes do not get the opportunity to set the program condition required for the first process/thread to make progress. In a multiprocessor system, other processes execute on other processors and thereby modify the program state in order to release the first process/thread from the spinlock

6.11 Show that, if the wait () and signal () semaphore operations are not executed atomically, then mutual exclusion may be violated.

Answer: A wait operation atomically decrements the value associated with a semaphore. If two wait operations are executed on a semaphore when its value is 1, if the two operations are not performed atomically, then it is possible that both operations might proceed to decrement the semaphore value, thereby violating mutual exclusion.

6.12 Show how to implement the wait() and signal() semaphore operations in multiprocessor environments using the TestAndSet () instruction. The solution should exhibit minimal busy waiting.

Answer:

```
int guard = 0;
int semaphore value = 0;
wait() {
        while (TestAndSet(&guard) == 1);
        if (semaphore value == 0) {
                atomically add process to a queue of processes waiting for the semaphore and set guard to 0;
        } else {
```

```
                semaphore value--;
                guard = 0;
        }
}
signal() {
        while (TestAndSet(&guard) == 1);
                if (semaphore value == 0 && there is a process on the wait queue)
                        wake up the first process in the queue of waiting processes
                else
                        semaphore value++;
        guard = 0;
}
```

6.17 Explain why implementing synchronization primitives by disabling interrupts is not appropriate in a single-processor system if the synchronization primitives are to be used in user-level programs.

Answer: If a user-level program is given the ability to disable interrupts, then it can disable the timer interrupt and prevent context switching from taking place, thereby allowing it to use the processor without letting other processes to execute.

Chapter 7

7.3 Consider a system consisting of four resources of the same type that are shared by three processes, each of which needs at most two resources. Show that the system is deadlock free.

Answer(book): Suppose the system is deadlocked. This implies that each process is holding one resource and is waiting for one more. Since there are three processes and four resources, on process must be able to obtain two resources. This process requires no more resources and therefore it will return its resources when done.

7.7 Consider the following resource-allocation policy. Requests for and releases of resources are allowed at any time. If a request for resources cannot be satisfied because the resources are not available, then we check any processes that are blocked waiting for resources. If a blocked process has the desired resources, then these resources are taken away from it and are given to the requesting process. The vector of resources for which the blocked process is waiting is increased to include the resources that were taken away.

For example, consider a system with three resource types and the vector Available initialized to (4,2,2). If process P0 asks for (2,2,1), it gets them. If P1 asks for (1,0,1), it gets them. Then, if P0 asks for (0,0,1), it is blocked (resource not available). If P2 now asks for (2,0,0), it gets the available one (1,0,0) and one that was allocated to P0 (since P0 is blocked). P0's Allocation vector goes down to (1,2,1), and its Need vector goes up to (1,0,1). a. Can deadlock occur? If you answer "yes," give an example. If you answer "no," specify which necessary condition cannot occur. b. Can indefinite blocking occur? Explain your answer.

Answer:
a. Deadlock cannot occur because preemption exists.

b. Yes. A process may never acquire all the resources it needs if they are continuously preempted by a series of requests such as those of process C.

7.9 Compare the circular-wait scheme with the deadlock-avoidance schemes (like the banker's algorithm) with respect to the following issues:

a. Runtime overheads

b. System throughput

Answer: Deadlock-avoidance scheme tends to increase the runtime overheads due to the cost of keep track of the current resource allocation. However, a deadlock-avoidance scheme allows for more concurrent use of resources than schemes that statically prevent the formation of deadlock. In that sense, a deadlock-avoidance scheme could increase system throughput.

7.10 Consider the following snapshot of a system:

	Allocation	Max	Available
	A B C D	A B C D	A B C D
P0	0 0 1 2	0 0 1 2	1 5 2 0
P1	1 0 0 0	1 7 5 0	
P2	1 3 5 4	2 3 5 6	
P3	0 6 3 2	0 6 5 2	
P4	0 0 1 4	0 6 5 6	

Answer the following questions using the banker's algorithm:

a. What is the content of the matrix Need?

The values of *Need* for processes *P*0 through *P*4 respectively are (0, 0, 0, 0), (0, 7, 5, 0), (1, 0, 0, 2), (0, 0, 2, 0), and (0, 6, 4, 2)

b. Is the system in a safe state?

Yes. With *Available* being equal to (1, 5, 2, 0), either process *P*0 or *P*3 could run. Once process *P*3 runs, it releases its resources which allow all other existing processes to run.

c. If a request from process P1 arrives for (0,4,2,0), can the request be granted immediately?

Yes it can. This results in the value of *Available* being (1, 1, 0, 0).One ordering of processes that can finish is *P*0, *P*2, *P*3, *P*1, and *P*4

7.12 Consider a computer system that runs 5,000 jobs per month with no deadlock-prevention or deadlock-avoidance scheme. Deadlocks occur about twice per month, and the operator must terminate and rerun about 10 jobs per deadlock. Each job is worth about $2 (in CPU time), and the jobs terminated tend to be about half-done when they are aborted. A systems programmer has estimated that a deadlock-avoidance algorithm (like the banker's algorithm) could be installed in the system with an increase in the average execution time per job of about 10 percent. Since the machine currently has 30-percent idle time, all 5,000 jobs per month could still be run, although turnaround time would increase by about 20 percent on average. a. What are the arguments for installing the deadlock-avoidance algorithm? b. What are the arguments against installing the deadlock-avoidance algorithm?

Answer: An argument for installing deadlock avoidance in the system is that we could ensure deadlock would never occur. In addition, despite the increase in turnaround time, all 5,000 jobs could still run. An argument against installing deadlock avoidance software is that deadlocks occur infrequently and they cost little when they do occur.

7.13 Consider the deadlock situation that could occur in the dining philosophers problem when the philosophers obtain the chopsticks one at a time. Discuss how the four necessary conditions for deadlock indeed hold in this setting. Discuss how deadlocks could be avoided by eliminating any one of the four conditions

Answer:

• Mutual Exclusion: When a philosopher picks up one chopstick, it cannot be shared with others. If they could be shared this situation would not exist and would prevent any deadlocks from occurring.

• Hold and Wait: When the philosopher tries to pick up a chopstick, he only picks up one at a time. If he could pick up both chopsticks at one time then a deadlock condition could not exist.

• No preemption: Once a philosopher picks up a chopstick, it cannot be taken away from her. If it could, then a deadlock condition could not exist.

• Circular Wait: Because all of the philosophers are sitting in a round table and each philosopher has access to the chopsticks next to them, if a philosopher picks up one chopstick he will affect the philosopher sitting next to him. and the philosopher on that side can also affect the philosopher sitting next to her in the same manner. This holds true all the way around the table. If one of the philosophers in the table could pick up a chopstick that another philosopher never needed, a deadlock condition would not exist.

7.14 What is the optimistic assumption made in the deadlock-detection algorithm? How can this assumption be violated?

Answer: The optimistic assumption is that there will not be any form of circular wait in terms of resources allocated and processes making requests for them. This assumption could be violated if a circular wait does indeed occur in practice.

7.16 Is it possible to have a deadlock involving only a single process? Explain your answer.

Answer: It is not possible to have a deadlock involving only one single process. The deadlock involves a circular “hold-and-wait” condition between two or more processes, so “one” process cannot hold a resource, yet be waiting for another resource that it is holding.