Concordia University
Computer Science & Software Engineering
Faculty of Engineering and Computer Science

COMP 346 – Fall 2017
Theory Assignment 4

Tianlin Yang
40010303

Due Date	**By 11:59pm Friday, November 23, 2018**
Format	Assignments must be typed and submitted online to Moodle system. **Scanned hand-written assignments will be discarded**.
Late Submission:	none accepted
Purpose:	The purpose of this assignment is to help you learn the overview of computer operating systems and Input and output mechanisms.
CEAB/CIPS Attributes:	Design/Problem analysis/Communication Skills

Question # 1

[10 marks] Consider the following preemptive priority-scheduling algorithm based on dynamically changing priorities. Larger priority numbers indicate higher priority (e.g., priority 3 is higher priority than priority 1; priority 0 is higher priority than priority -1, etc.). When a process is waiting for the CPU in the ready queue, its priority changes at the rate of α per unit of time; when it is running, its priority changes at the rate of β per unit of time. All processes are assigned priority 0 when they enter the ready queue. Answer the following questions:

(a) What is the scheduling algorithm that results when $\beta > \alpha > 0$? Explain.

(b) What is the scheduling algorithm that results when $\alpha < \beta < 0$? Explain.

Answer[i]:

(a) Since processes start at 0, any processes that have been in the system (either running or waiting) have higher priority. Therefore, new processes go to the back of the queue. When a process runs, its priority keeps increasing at the rate of beta, which is more of an increase than for the processes in the ready queue. Therefore, every time the process has timer runout, it goes to the front of the ready queue and gets dispatched again. This is the equivalent of FCFS.

(b) This time, any new processes entering the system have higher priority than any old ones, since priority starts at zero and then becomes negative when the process waits or runs. New processes go in at the front of the queue. When a process runs or waits, its priority decreases, with the waiting processes decreasing faster than the running process. This is a LIFO (last-in-first-out) algorithm. The question didn't promise that this would be an algorithm you would want to use in real life.

Question # 2

[10 marks] Somebody proposed a CPU scheduling algorithm that favors those processes that used the least CPU time in the recent past. For calculating the CPU time of a process in the recent past, a time window of size π is maintained and the recent CPU time used by a process at time T is calculated as the sum of the CPU times used by the process between time T and T- π. It is argued that this particular scheduling algorithm (a) will favor I/O-bound processes, and (b) will not permanently starve CPU-bound processes. Do you agree/disagree with (a) and (b)? Explain.

Answer[ii]:

(a) It will favor the I/O-bound processes, so (a) is correct.

Because the I/O-bound processes frequently request of the short CPU burst, the least CPU time will be reached compare with other heavy and none-I/O-bound processes.

(b) Also correct, the CPU-bound processes will not starve.

Because the I/O-bound processes will frequently release the CPU impropriate to waiting I/O, therefore the starvation could be avoided.

Question # 3

[10 marks] Consider a variant of the round robin (RR) scheduling algorithm in which the entries in the ready queue are pointers to the Process Control Blocks (PCBs), rather than the PCBs. A malicious user wants to take advantage and somehow, through a security loophole in the system, manages to put two pointers to the PCB of his/her process with the intention that it can run twice as much. Explain what serious consequence(s) it could have if the (malicious) intention goes undetected by the OS.

Answer[iii]:

Process appear twice in the ready queue and is scheduled twice as often as other processes. Advantage is implement priorities. But it will overheads in maintaining pointers; same number of context switches. Adaptive quantum for each process. A higher priority process can use up to 2/3/4/etc. quantum of time over the single quantum for normal processes.

Question # 4

4 [10 marks] Consider the version of the dining philosopher's problem in which the chopsticks are placed in the center of the table and any two of them can be used by a philosopher. Assume that requests for chopsticks are made one chopstick at a time. Describe a simple rule for determining whether a particular request can be satisfied without causing deadlock given the current allocation of chopsticks to philosophers.

Answer[iv]:

When a philosopher makes a request for their first chopstick, do not satisfy the request only if there is no other philosopher with two chopsticks and if there is only one chopstick remaining.

Assume that there are N dining philosophers. There are a total of N chopsticks in the center at the beginning. So Available=N. Max=2 since each philosopher needs 2 chopsticks to finish. Each request can ask for only one chopstick. We can use Banker's rule for this as follows. Grant a request for a chopstick from a Philosopher Pi only when:

1. The number of chopsticks in the center is ≥1, AND

2. There is at least one philosopher who can finish (have 2 chopsticks) after Pi's request is granted. Illustration: Let there be 4 Philosophers. So, there are 4 chopsticks in the center. Available=4. Right now, no one is holding a chopstick. P1 makes a request for a chopstick. Let us check the rule. Available ≥1 and after the request is granted Available=3.

With this, P1, P2, P3, or P4 can finish. The rule is satisfied and hence request may be granted.

Question # 5

[10 marks] Consider Q.4 above. Assume now that each philosopher needs three chopsticks to eat. Resource requests are still issued one at a time. Describe some simple rules for determining whether a particular request can be satisfied without causing deadlock given the current allocation of chopsticks to philosophers.

Answer[v]:

When a philosopher makes a request for a chopstick, allocate the request if:

1) the philosopher has two chopsticks and there is at least one chopstick remaining,

2) the philosopher has one chopstick and there are at least two chopsticks remaining,

3) there is at least one chopstick remaining, and there is at least one philosopher with three chopsticks,

4) the philosopher has no chopsticks, there are two chopsticks remaining, and there is at least one other philosopher with two chopsticks assigned.

Question #6

[10 marks] Consider a system consisting of *m* resources of the same type being shared by *n* processes. A process can request or release only one resource at a time. Show that the system is deadlock free if the following two conditions hold:

1. The maximum need of each process is between 1 resource and *m* resources.
2. The sum of all maximum needs is less than $m + n$.

Answer[vi]:

Suppose N = Sum of all Needi, A = Sum of all Allocationi, M = Sum of all Maxi. Use contradiction to prove.

Assume this system is not deadlock free. If there exists a deadlock state, then A = m because there's only one kind of resource and resources can be requested and released only one at a time. From condition b, N + A = M < m + n. So we get N + m < m + n. So we get N < n. It shows that at least one process i that Needi = 0. From condition a, Pi can release at least 1 resource. So there are n-1 processes sharing m resources now, condition a and b still hold. Go on the argument, no process will wait permanently, so there's no deadlock.

Reference

[i] http://codex.cs.yale.edu/avi/os-book/OSE2/practice-exer-dir/6-web.pdf

[ii] https://www.cs.princeton.edu/courses/archive/fall11/cos318/lectures/L7_CPUScheduling.pdf

[iii] https://www.cs.princeton.edu/courses/archive/fall11/cos318/lectures/L7_CPUScheduling.pdf

[iv] https://quizlet.com/98417175/chapter-7-recommended-exercises-flash-cards/

[v] https://faculty.psau.edu.sa/filedownload/doc-6-pdf-78ea22b6de9c90b6baaa8328518c05c8-original.pdf

[vi] https://www.cs.du.edu/~dconnors/courses/comp3361/assignments/sol3.txt