COMP530 Fall 2011 HW-3

Student Name: Duo Zhao PID: 720090360

1 Q1

1. Consider an implementation of the round-robin scheduling algorithm where the entries in the ready queue are pointers to the process control blocks (PCBs) of processes.

a) For a particular process, what would be the effect of putting two pointers to the process PCB in the ready queue? What would be the major advantages and disadvantages of this scheme?

It will have more time to execute in each round-robin period. Since all ready process are allocated equal access-time to CPU, if the process is listed in the queue twice, it will theoretically be granted twice as much the CPU time as other processes.

Advantage: It is easy to put more time priorities to some processes. It's extremely useful when some processes need higher priority.

Disadvantage: It need more efforts for duplicate PCB pointer maintenance. Insert(), Remove() must be execute twice. As the processing chance doubles, so does the context switch it incurs. Even the two pointers are consecutively in the queue, it may still need a context switch, at least a check.

b) How would you modify the basic round-robin algorithm (while keeping the basic structure of a round-robin scheduler) to achieve the same effect for processes without using duplicate pointers?

Rather than allocate each process a fixed time interval, allow the scheduler to set each process a different time slice, which based the priority of the process.

2 Q2

2. When implementing a Round-Robin scheduler:

a) When and why would one use a large quantum? Explain.

When the overall efficiency is highlighted.The smaller time quantum, the more jobs (or more frequent for a particular long job) will be suspended before it completes. Consequently, it will result in more context switch between processes and these interrupts may be time-consuming.

b) When and why would one use a small quantum? Explain.

When the wait time/response time is highlighted, especially when there are many short jobs. In the large-quantum scheme, it may wait a long time in the ready queue before the short job's turn to execute, since large time intervals are allocate to the processes before them. If the jobs in front of the short job are long-time jobs, the short job suffers.

3 Q3

3. An executive working in RTP is unhappy with the performance of his personal computer. He runs a large number of jobs/processes simultaneously but is particularly concerned about the performance of one job. An NC State intern, observing that the disk on the computer is constantly busy suggests replacing the disk with a faster model. The executive does this, however, the performance of his main job of interest degrades terribly. The UNC intern, a COMP 530 graduate, steps in and explains why this performance anomaly is observed.

What is happening in this computer system? Why did the performance of the executives main job decrease when a faster disk was installed? (Having impressed the executive, does the UNC intern share his new massive pay raise with his COMP 530 instructor?)

Under the instruction of the process scheduling algorithm in the operating system, the change of actual/estimated process information(arrival time, I/O time, remaining time, priority value...) may result in a different execution order.

Before the replacement of fast model disk, many processes with a disk access are supposed to I/O bound jobs due to the poor performance of the slow disk. Therefore, they spend a lot time in the waiting/blocked state rather in the ready queue for running, or they're likely not to be favored by the scheduling algorithm due to their long I/O period. Nevertheless, the main job which the executive cares do not rely on the access to the poor disk. As a CPU bound job, It enjoyed more chance in the ready queue or other advantage for running.

After the disk replacement, the I/O time for many jobs is significantly lowered. Consequently, many of them return to the ready queue more frequently, and compete with the executive's main job. Due to the sudden advent of numerous competitors in the ready queue, it's not so advantageous as before.

4 Q4

4. Suppose a scheduling algorithm (at the level of short-term CPU scheduling) favors existing processes (processes already in the system) that have used little processor time in the past.

a) Which class of processes will this policy favor? Explain.

The policy favors the I/O bound processes, since the a I/O bound job has less chance of CPU usage than a CPU bound job in the past. The scheduling algorithm will prefer I/O bound jobs.

b) Is starvation possible under this scheme? Explain.

This scheme won't result in starvation. Since the I/O bound jobs start increasing CPU time occupancy, the processor time allocates to CPU bound jobs will decrease. It will eventually arrive at a time when the CPU bound jobs acquire lower processor time than I/O bound jobs, when, according to the criteria of the algorithm, the CPU will alternate to favor CPU bound jobs. Therefore the CPU bound job won't suffer from starvation, since it is impossible for the job to be denied to CPU access.

c) How might this scheduling policy best be implemented? Describe briefly in words.

The shortest served time first algorithm can be implemented preemptively. The processes in the ready queue are sequenced according to the total past CPU served time of each existing process. The shorter CPU used time it possess, the more close to head (higher priority) it is in the queue. The longer CPU used time it possess, the more close to tail (lower priority) it is in the queue. As the running process is executing, its CPU used time is increasing. Some preemptive policies may implement here,

(1)When the CPU used time of the running process ($T_{running}$) excesses some level of the CPU used time of process in the head of the ready queue (T_{head}), such as $T_{running} > a \cdot T_{head} + b$ ($a > 1, b > 0$). $a > 1$ is to avoid the infinitely frequent swap of the running process and the head process. $b > 0$ is to avoid the running process is preempted each time a new process arrives.

(2) Implement a round-robin policy. Each selected process was executed in a fixed time in CPU, and then make a context switch, insert it back into the ready queue with the given sorting policies.

d) Would this be a good policy to use in a time-sharing system? Explain.

No. Time-sharing system requires equal opportunity to each process. Each process is supposed to enjoy a short response time. However, the provided algorithm favors the processes

with less CPU used time. For the existing long-time process, it will waiting a long time, when all other processes either excess its CPU used time, is blocked or terminates, before its turn to re-execute. Especially when there is numerous relative short-time processes creation to the ready queue. The existing long-time process will be heavily delayed.

5 Q 5

5. Consider the producer/consumer solution using a shared counter that we developed in class:

```
1  ================================
2  process Producer
3    var c : char
4  begin
5    loop
6      <produce a character  c >
7      while count = n do
8        NOOP
9      end while
10     buffer[nextIn] := c
11     nextIn := nextIn+1 mod n
12     count  := count + 1
13   end loop
14 end Producer
15 ================================
```

```
1  ================================
2  process Consumer
3    var c : char
4  begin
5    loop
6      while count = 0 do
7        NOOP
8      end while
9      c := buffer[nextOut]
10     nextOut := nextOut+1 mod n
11     count   := count - 1
12     <consume a character  c >
13   end loop
14 end Consumer
15 ================================
```

2

a) What exactly was the problem with this solution?

No initial values are assigned to the shared variables, char buffer[] and int count.

It does not guarantee mutual exclusion for the shared variable count. This violation of the correctness condition may result in misrepresentation of usage of count. Race condition may occur while the producer is executing "count := count + 1" and the consumer is executing "count := count - 1", either instruction is composed of three atomic operations rather than one. It may have trouble when operations in one process interleave with those of another process. For instance, if the execution sequence is

MOV R_1 @count //producer loads count to $register_1$

MOV R_2 @count //consumer loads count to $register_2$

ADD R_1 1

SUB R_2 1

MOV @count R1

MOV @count R2

The count will decrease by 1 as a result of data inconsistency. However, the correct anticipated result would be that count remain the same.

b) Ultimately we eliminated the counter and replaced it with a pair of counting semaphores. However, what if instead we simply required the operating system to perform non-preemptive scheduling of the producer and consumer processes? Would non-preemptive scheduling have made the counter-based solution above correct? Explain.

For a multi-processor/multi-core environment, the above code does not function well obviously. In spite of non-preemptive scheduling, the producer and consumer may execute in separate processors in parallel.

For a single processor environment, although one running process cannot be interrupted and therefore mutual exclusion is guaranteed by the non-preemptive scheduling, it may not guarantee bounded waiting. For instance, if the scheduler selects consumer process first(with initial value of count =0) , the consumer process will be infinitely busy-waiting at while-NOOP section, which makes no progress at all. Meanwhile, the processor can never be granted a chance for executing in this case. A deadlock situation occurs and the producer suffers from CPU starvation.

6 Q6

6. Write a solution to the producer/consumer problem that uses (only) binary semaphores for both condition synchronization and mutual exclusion.

Since there are several ways to implement a general semaphore with binary semaphores, it is possible to implement binary semaphores directly to solve producer/consumer problem. A general idea is to associate an integer variable to each of emptyBuffer and fullBuffer.

General idea:

The value pool of emptyBuffers and filledBuffers are {0,1,2, ..., n}

use a int counter to represent the value of the counting semaphore and use a binary semaphore to represent if the value is non-zero.

BinarySemaphore IsThereEmptyBuffers;

// when IsThereEmptyBuffers==0 means emptyBuffers==0;

// when IsThereEmptyBuffers==1 means emptyBuffers may pick up values from {1,2,3,...n};

BinarySemaphore IsThereFilledBuffers;

// when IsThereFilledBuffers==0 means filledBuffers==0;

// when IsThereFilledBuffers==1 means filledBuffers may pick up values from {1,2,3,...n};

BinarySemaphore mutex;

// For the mutual exclusion of the shared counter;

```
Shared variables for both the producer and the consumer process:
#define N BUFFER_SIZE
BinarySemaphore IsThereFilledBuffers = 0;
BinarySemaphore IsThereEmptyBuffers = 1;
BinarySemaphore mutex = 1;
char buffer[N] = "";  // N is the buffer size
int count = 0;        //Values from{0,1,2,...n}
int nextIn = 0;
int nextOut = 0;
```

```
==========
Process Producer
loop
        IsThereEmptyBuffers.down();
        buffer[nextIn] = 'c';
        nextIn = (nextIn+1)%n;  // +1 and then mod n
        mutex.down();
                count++;
        mutex.up();
        if(count<n){
                IsThereEmptyBuffers.up();  //count not full, up it.
        }
```

```
13          IsThereFilledBuffers.up();
14  end loop
15  ================================
16  Process Consumer
17  loop
18          IsThereFilledBuffers.down();
19          data = buffer[nextOut];
20          nextOut = (nextOut+1)%n;   // +1 and then mod n
21          mutex.down();
22                  count--;
23          mutex.up();
24          if(count>0){
25                  IsThereFilledBuffers.up();    // count not empty, up it.
26          }
27          IsThereEmptyBuffers.up();
28  end loop
29  ================================
```

Alternatively,

```
1   ================================
2   Process Producer
3   loop
4           if(count >= n-1){       // count = {n-1,n} enter
5                   IsThereEmptyBuffers.down();
6           }
7           buffer[nextIn] = 'c';
8           nextIn = (nextIn+1)%n;   // +1 and then mod n
9           mutex.down();
10                  count++;
11          mutex.up();
12          IsThereFilledBuffers.up();
13  end loop
14  ================================
15  Process Consumer
16  loop
17          if(count <= 1){       // count = {0,1} enter
18                  IsThereFilledBuffers.down();
19          }
20          data = buffer[nextOut];
21          nextOut = (nextOut+1)%n;   // +1 and then mod n
22          mutex.down();
23                  count--;
24          mutex.up();
25          IsThereEmptyBuffers.up();
26  end loop
27  ================================
```

7 Q7

7. What hardware support is necessary to implement a secure operating system (an operating system wherein user programs can only access their data and the system is immune from user programs crashing or hanging)? Some say no hardware support is needed! Consider whether it is possible to construct a secure operating system for a computer with no special support for the operating system (e.g., no dual-mode operation). Is it possible to do so? Explain in detail. (Hint: The answer is yes, its possible. There are actually several ways to do this.)

If no hardware support is needed. Hardware should support the dual model operation. The operating system should be able to identify and manage the User-mode and Administrator-mode operations. Also, the system contains a table of the access level of resources(user/administrator). When a user entered the system. The system should have a mechanism to differentiate a user process and a system process, such as using a PID number $(PID < M) -- > SystemProcess$, $(PID > M) -- > UserProcess$. The system grants different access authority to user process and system process. The system checks every process for its authority level, before executing its instructions. When a user process makes a over-authority system call, it will be denied by the operating system, and make an appropriate feedback(e.g. return an value of error) to the violating user process.

8 Q8

8. A scheme for implementing a context switch was presented in class wherein interrupts were disabled while performing a context switch to ensure mutual exclusion held.

```
1 context_switch(queue : system_queue)
2 var next : process_id
3 begin
4   DISABLE_INTS
5   insert_queue(queue,runningProcess) // Line A
6   next := remove_queue(readyQueue)  // Line B
7   dispatch(next)                    // Line C
8   ENABLE_INTS
9 end context_switch
```

a) Why is mutual exclusion required in this code? What might go wrong if mutual exclusion was not guaranteed?

When context switch is executing, no other operations on the running process and ready queue should be allowed, which may result in failure of context switch, or race condition occurs. For instance, if one context switch interleaves with another and executes in the following sequence:

Context Switch 1: Line A $->$ Context Switch 2: Line A
$->$Context Switch 1: Line B$->$ Context Switch 2: Line B
$->$Context Switch 1: Line C$->$ Context Switch 2: Line C

The current running process will inserted into queue twice, which will result in two PCBs to the process in the queue, which is troublesome for maintenance at a later time (e.g. when execute remove(), it must be removed twice.)

The first two processes at the head of the ready queue will be removed. However, only the latter one is actually dispatched for running, the former removed-from-ready-queue process is missing.

b) What would be the pros and cons of using binary semaphores to ensure mutual exclusion instead of disabling and enabling interrupts?

PROS:

(1) In a multi-processor environment with a shared CPU ready queue, Semaphore-type context switch guarantees there is at most one context switch across all CPUs. On the contrary, interrupts disabling only exclusion at one CPU.

(2) When errors during context switching occur, it is easy to handle in the semaphore scheme. The supervisor call, scheduler-induced preemptions still work for it. However, if errors occurs while interrupts disabled, there is little way to terminate or restore it.

CONS:

(1)In a multi-processor environment with Per-CPU ready queue, There could only be one context switch in all CPUs at the same time. However, each processor only need to have its own context switch exclusion, that is to say, context switches in separate processors are allowed to happen simultaneously. However, Semaphore-type context switch fails to achieve this.

(2) Semaphore-type exclusion could only guarantee the exclusion of other context switch operations. However, the ready queue may be shared by many operations other than context switch. If these kinds of operations exist, semaphore is hard to guarantee an exclusion to non-common variables.

(3)If it is guaranteed to make an exclusion to all other operations, the shared variable(this semaphore) must be included in all PCBs and programing codes. Then it will be a great challenge to the current computer architecture to achieve this.

9 Q9

9. The COMP 530 Teaching Assistant holds office hours twice a week in his office. His office can hold 2 persons: 1 TA and 1 student. Outside the office are 4 chairs for waiting students. If there are no students waiting to see the TA, the TA plays Gears of War. If a student arrives at the TAs office and the TA is playing games, the student loudly clears his throat and the TA invites the student in and begins helping her. If a student arrives at the office and the TA is busy with another student, the student waits in a chair outside the TAs office until the TA is free. If the arriving student finds all the chairs occupied, then he leaves.

Using semaphores, write two process, $student_i$ and TA, that synchronize access to the TAs office during his office hours. These processes will have approximately the following structure

```
process TA
  loop
        <Entry protocol to synchronize with a student>
        <Advise a student>
        <Exit protocol>
  end loop
end TA
```

```
process student_i
        <Entry protocol to synchronize with the TA>
        <Get advice or leave>
        <Exit protocol>
end student_i
```

(Note that although you are writing one student process, assume multiple instances of the process are active simultaneously.)

```
1  Shared variables:
2          #define CHAIRS 4
3          Semaphore students = 0; // # of students waiting  {0,1,2,3,4}
4          Semaphore TAs = 0;       // # of TAs waiting for students, {0,1}
5          Semaphore mutex =1;      //  mutual exclusion, binary {0,1}
6          int waiting = 0;         // # students are waiting
7
8  process TA
9           loop
10                 students.down();  // if students = 0,
11                                   //execute a handler to promote TA to play games
12                 mutex.down();
13                         waiting--;
14                         TAs.up();
15                 mutex.up();
16                 toAdvise();
17         end loop
18 end TA
19
20 process student_i
21         mutex.down();      //enter the critical section
22         if(waiting<CHAIRS)     //check if seats are available
23         {
24                         waiting++;
25                         students.up();   //clear throat & stop TA from playing
26                                 //if student from 0 to 1, clear throat
27                 mutex.up();
28                 TAs.down();
29                 BeAdvised();  //
30         }
31         else
32         {
33                 mutex.up();
34                 _exit();   // no seats and leave
35         }
36 end student_i
```