Comp 530 Homework 3
Name: Yee Lyn Chan (Alexis)

10/05/2011

Collaborated with Rebecca Lovewell

Discussed question 6 with Stephanie Zolayvar

Question 1

a) The process will have two time slices per round in the round-robin ready queue to run (other processes have 1 time slice per round). It will be allocated twice the amount of time that is allocated to other processes (2 quanta instead of 1 quantum).
   Advantages:
   i. That particular process' turnaround time, i.e. the interval from the time of process submission to the time of completion, when it is granted 2 quanta per round will be shorter than its turnaround time when it is granted 1 quantum per round. If the user cares greatly about the turnaround time of that particular process, then this is an advantage.
   ii. If a scheduler needs to assign a higher priority to a process without changing its mechanism, this is an easy way to implement it.

   Disadvantages:
   i. The other processes will have to wait for an extra quantum in between turns. Their turnaround time will be longer than if that particular process did not have 2 quanta of time slice per round. This will be a disadvantage if the user cares greatly about the turnaround time of one of the other processes instead of the process with the two quanta of time slices.
   ii. There is an overhead of context switch time involved in scheduling a process's pointer twice compared to just increasing the time alloted to it to 2 quanta (if the scheduler is using this as a method to assign higher priority to the process).

b) **Method 1:**
   In round robin scheduling, the timer is set to raise an interrupt at the end of 1 quantum of time slice and the process that is executing (if it has not released the CPU voluntarily) will be interrupted and a context switch will be executed. The process will be put at the tail of the ready queue.
   To achieve the same result as the scenario in 1 (a) without using duplicate pointers:
   i. For each round in the round-robin, we schedule the process pointer only once.
   ii. For every other occurrence of timer interrupt for that particular process, instead of putting that particular process at the end of the ready queue, the round-robin scheduler should put the process at the head of the ready queue. (And for the rest of the timer interrupts for that particular process, the scheduler should put the process at the end of the ready queue). The net effect of this is that particular process is executed for 2 quanta of time slices every round.

   Method 2:
   Alternately, the scheduler can also just increase the quantum of that particular process to 2 quanta.

Question 2

a) A large quantum should be used if
   i. **the context switching time is large**
   In general the quantum should be very much larger than the context switching time. Otherwise, the ratio of context switches to the number of commands executed be very high. The throughput of the processor goes down because the processor will be spending most of the time performing context switches.
   ii. **a First-Come-First-Serve effect is desired**
   with a very large quanta, there is a higher probability that each process voluntarily

relinquishes control of the CPU before its quantum expires. In which case, processes run in a First-Come-First-Serve order. (Operating Systems Concepts, Sliberschatz, pg 165).

b) A small quantum should be used if
   i. **the context switching time is very small and high interactivity is desired**
   as long as the quantum is larger than the context switching time, using a smaller quantum will reduce the amount of time that a process waits in between rounds. This will allow the process to produce some output at shorter intervals, giving the user a higher perception of interactivity.

Question 3

Since the disk is constantly busy, we can assume that there might be some processes in the computer that are IO-bound. These processes do not spend a lot of time executing instructions on the CPU – they spend most of the time performing IO instructions (reading from the disk, for example). The job that the executive is concerned about could be a CPU-bound process. When disk is replaced with a faster model, the amount of time it takes to read from and write to the disk is drastically reduced, so the IO-bound processes return to the system queue sooner. This means that more context switches will take place as the IO processes return from read/write operations and the CPU-bound process is interrupted more frequently before it manages to complete its job. This increases the turnaround time for the CPU-bound process, and the executive observes that the job's performance has deteriorated.

Question 4

a) I/O bound processes (Interactive processes) –since these processes spend little time running CPU instructions and most of the time waiting for response from I/O devices, so they exit the ready queue quicker than CPU bound processes. Therefore, they have a "track record" of having used little processor time in the past.
b) Yes. A CPU-bound process would have a history of using the processor for a long time in the past and will be assigned a higher priority value (higher priority value means "lower priority" – means closer to the tail of the queue). If there are several I/O bound processes also running on the same processor, and none of the processes are terminating, then the I/O bound processes will be continuously kept at the head of the queue. The CPU-bound process may never be executed as long as the I/O bound processes do not terminate.
c) The scheduler should also decrease the priority value (increase the priority) of a process as the process "ages" i.e. a process' priority increases as the amount of time lapse since the process was submitted to the ready queue increases.
d) No this is not a good policy for a time-sharing system because jobs that are not highly-interactive (i.e CPU bound) will have lower priority. But a CPU-bound job's performance may be very important to the user.

Question 5

a) Both the Producer and the Consumer process are editing the variable count and operate on shared memory (i.e. the buffer). The section where count is updated is a critical section and is not atomic. Therefore, implementation shown here is not mutually exclusive.

   For example, the Producer executes the line **nextIn := nextIn + 1 mod n** but was interrupted by Consumer before it could increment count by 1. Consumer then reads a character from buffer and decrements count. Since count has not been incremented by Producer, this means that count is 1 less than the number of characters actually in the buffer. When count reaches n-1, the number of characters in the buffer could be equal n, in which case Producer will overwrite the character at index 0, which might not have been consumed by the Consumer.

Basically by updating shared memory in a critical section without mutual exclusion, the code runs the risk of producing characters that are not consumed by the consumer.

b) The critical section will be mutually exclusive between the producer and consumer processes. But if the Producer does not voluntarily release control of the CPU to the process scheduler, it will be stuck in the NOOP while loop waiting for the full buffer to be emptied before producing the next character. However, the full buffer could not be emptied because the Consumer process could not run since the Producer process could not be preempted by the scheduler.
Another likely case is that the Consumer keeps blocking in the while loop waiting for the empty buffer to fill up with a character for its consumption and does not voluntarily release control of the CPU. The empty buffer could not be filled because the Producer process could not run as the Consumer process could not be preempted by the scheduler.

Question 6
Shared memory
Binary semaphore notFull = 1
Binary semaphore notEmpty = 0
Binary semaphore countMutex = 1
Buf   // Shared buffer for storing characters

Private to Producer
nextIn = 0 // index of next spot in buffer to fill

Private to Consumer
nextOut = 0 // index of next spot in buffer to empty

Producer

```
Start loop
        <Produce character 'c'>
        notFull.down()                  // Acquire mutual exclusion for producing if buffer is not full
        buf[nextIn] = input             // Put a character into buffer
        nextIn = nextIn + 1 mod n       // Increase index of next spot in buffer to fill
        countMutex.down()               // Acquire lock to count
                count=count + 1     // Increase count because we put a character into buffer

                if (count < n)          // As long as count is less than n we can be sure that the buffer
                        notFull.up()    // is not full so we release the lock on notFull so that producer
                endif                   // can keep producing without having to wait for Consumer to
                                        // unlock it
        countMutex.up()                 // Release lock on count
        notEmpty.up()               // Release lock on notEmpty because we put a character in the buffer
End loop
End Producer
```

Consumer

```
Start loop
        notEmpty.down()                    // Acquire mutual exclusion for consuming if buffer is not empty
        data = buf[nextOut]                // Remove a character from buffer
        nextOut = nextOut + 1 mod n        // Increase index of next spot in buffer to empty
        <consume data>
        countMutex.down()                    //Acquire lock to count
                count= count - 1        // Decrease count because we consumed a character from buffer

                if (count >0)               // As long as count is more than 0 we can be sure that the buffer
                        notEmpty.up()     // is not empty so we release the lock on notEmpty so
                end                         //  that consumer
                                            // can keep producing without having to wait for Producer to
                                            // unlock it
        countMutex.up()                    // Release lock on count
        notFull.up()             // Release lock on notFull because we removed a character from the buffer
End loop
End Consumer
```

Question 7

Most processors provide a protected instruction set for system procedures that cannot be accessed by user processes. A bit in the processor status word indicates the current "mode" of instruction execution in the processor – an instruction can either be in "user mode" or "system mode". An instruction in "system mode" can access protected registers and memory that cannot be accessed by an instruction in "user mode". This is called the dual mode operation. Interrupts and system calls are used to switch from user to system mode to allow the user to request that the system execute protected instructions on its behalf (such as reading/writing to IO).

Yes it is possible to implement a secure operating system without special hardware support. This requires safe programming practices. For example, in Linux, most of the operating system commands are implemented by forking off a new process and calling exec() to execute the file containing the instructions of the new process. This wipes the memory image belonging to the parent process and replaces it with the memory image of the child process. Therefore the child process cannot access or modify the parent memory's image, ensuring safety of the parent process. If a child process fails, it fails without also killing the parent process or other processes in the system.

Another method to ensure security without having hardware based support is to ensure safe programming practices via compiler-based enforcement. "A language implementation might provide standard protected procedures to interpret software capabilities that would realize the protection policies that could be specified in the language." (Operating Systems Concepts, Sliberschatz, pg 551).

Question 8

a) Mutual exclusion is required because the system queue and ready queue are shared by different processes. If mutual exclusion is not guaranteed, the modification of the shared memory by a process is not atomic – the process could be interrupted in the middle of operations on the shared memory and another process would start operating on the partially modified shared memory resulting in errors. For example, a process could be removed from the readyQueue at **next := remove_queue(readyQueue)** , at which point the context_switch could be interrupted. For example, if another **context_switch(ready_queue)** is called, the previous runningProcess (not "next") is inserted again into "queue". The "next" is replaced by the next process from the readyQueue and that process is dispatched. This means that one process is not dispatched when it should have been because the value of “next” is rewritten with the value of another process id.

b) **Pros of using binary semaphores instead of disabling and enabling interrupts**:
We do not unnecessarily disable interrupts. If disabling/enabling interrupts is used to ensure mutual exclusion of the critical section in a process, the process has sole control of the CPU for a longer amount of time (compared to using a TST type binary semaphore) during which no other process can run. Interrupts are important to allow the operating system to switch to higher priority processes such as handling a user shutdown request (or cancelling the launching of a missile, for example). Disabling interrupts would prevent the operating system from responding to important events as soon as possible.

Cons of using binary semaphores instead of disabling and enabling interrupts:
There is higher waiting time involved in using binary semaphores instead of disabling interrupts as the process has to wait for the semaphore lock to unlock the critical section. Whereas a process disabling interrupt would get into the critical section without having to wait.

Question 9

```
Shared memory
Counting semaphore  emptyRoomQueue = 5            //We assume counting semaphore is
                                                 // implemented using OS kernel which implements a
                                                 //queue
Binary semaphore  stopPlayingGearsOfWar = 0
Binary semaphore  countMutex = 1
emptySeats = 4
Binary semaphore emptyRoom = 1

TA

Start loop

        stopPlayingGearsOfWar.down()             // Call binary semaphore down function
                                           //which spins until a student appears
                                           // This is equivalent to the TA playing Gears of War until a
                                           // student clears his throat to inform the TA that he needs
                                           // advice

        countMutex.down()                   // Once stopPlayingGearsOfWar is released by Student
                                           // or TA
        emptyRoomQueue.up()                 // TA releases a spot on the empty seat queue by inviting
                                           // the student in to the office

        emptySeats  = emptySeats + 1       // So the number of empty seats increases

        countMutex.up()
         <advise student>
End loop
End TA

Student
```

```
        countMutex.down()                      //Acquire lock to count of empty seats
        if (emptySeats <= 0)                   // Technically emptySeats cannot be negative.
                countMutex.up()                // But if emptySeats <= 0, there are no empty seats left
                exit Student                   // so the student sadly leaves without waiting
        end
        emptySeats = emptySeats – 1            //Student sits down, so he takes up 1 empty seat outside
        emptyRoomQueue.down()                  // Student seats outside, waiting for an empty room.
        countMutex.up()                   // Release lock on count
        emptyRoom.down()                  // Student waits for empty room before clearing throat
        stopPlayingGearsOfWar.up()        // Student loudly clears her throat
                                          // which the TA has been waiting
                                          // for (i.e. the TA was waiting for this binary semaphore)
        <gets advice>
        emptyRoom.up()               // Student leaves the office
End Student
```