# COMP 252 Principles of Systems Software

## Fall semester 2000

Mid-Term Examination

Date: October 22, 2000 Time: 1:00pm-3:00pm

| | | |
|---|---|---|
| Name: SOLUTION | Student ID: | Email: |

## Instructions:

1. This examination paper consists of 8 pages and 6 questions.
2. Please write your name, student ID and Email on this page.
3. For each subsequent page, please write your student ID at the top of the page in the space provided.
4. Please answer all the questions within the space provided on the examination paper. You may use the back of the pages for your rough work.
5. Please read *each question very carefully* and answer the question clearly and to the point. Make sure that your answers are neatly written, readable and legible.
6. Show all the steps you use in deriving your answer, wherever appropriate.
7. For each of the questions assume that the concepts are known to the graders. Concentrate on answering to the point what is asked. Do not define or describe the concepts.

| Question | Points | Score |
|---|---|---|
| 1 | 10 | |
| 2 | 10 | |
| 3 | 20 | |
| 4 | 20 | |
| 5 | 20 | |
| 6 | 20 | |
| TOTAL | 100 | |

1a. Explain briefly the difference between the *job queue* and the *ready queue*.

- Job queue is keeping the jobs on the disk which are submitted to the system but which are not running yet.

-Ready queue is keeping processes/threads which are loaded into the memory and which have started their execution.

1b. Explain briefly the difference between *multiprogramming* and *time-sharing*.

In multiprogramming a process is holding a CPU until it needs some I/O operation or waits for an event.

In time-sharing a process can hold a CPU up to the size of the time-slice (quantum) or until it asks for an I/O (event) whichever comes first.

1c. Explain briefly the difference between *spinlock semaphores* and *semaphores with no busy-waiting*.

Spinlock semaphores use busy waiting. There is no context switch (except for the timer interrupt) when a process must wait on a lock. They are used when locks are expected to be held for a short time.

Semaphores without busy waiting put the calling thread/process on the waiting list if the of the sempahore was 0 at the moment when P() operation was invoked.

1d. Explain briefly the difference between the *user level thread* and the *traditional process*.

Multiple threads can execute in a single address space so that sharing of resources and data is easy. Creation and deletion of threads is much less expensive.

However, threads running in the same address space may interfere with one another if not properly synchronized.

2. Explain briefly the following concepts *in not more than two sentences* each.

   a) Mutual exclusion in the Critical Section Problem
   b) Aging
   c) Blocking a process
   d) Context switching
   e) Non-preemptive scheduling

a) If multiple processes need to acces a critical section, they can do this only one process at a time.

b) Process which has spent a lot of CPU time will be penalized by decreasing its priority. In order to prevent starvation of such a process, its priority will be also raised gradually in proportion to the time spent in the system.

c) process which has to wait for an I/O operation or for an event will be blocked, i.e. it will be put on the waiting queue for appropriate device or synchronization object.

d) Context switching is the action of copying of the contents of CPU registers into the PCB of the process which was running at the moment of context switch and loading the contents of PCB of the process which has to start running into CPU registers.

e) In non-preemptive scheduling, process (thread) which is currently running on the CPU can not be preepted by the higher priority process (thread) which arrives while its CPU burst was executed.

3. Consider a 3-level feedback queue with Round-Robin Scheduling, as shown in Figure 1.

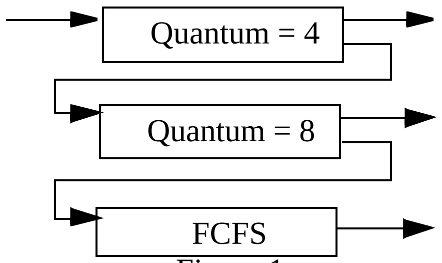


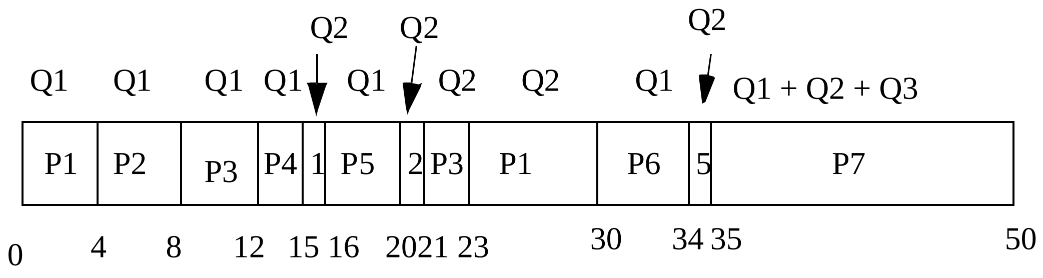
Figure 1.

| Process | P1 | P2 | P3 | P4 | P5 | P6 | P7 |
|---|---|---|---|---|---|---|---|
| CPU-burst | 12 | 5 | 6 | 3 | 5 | 4 | 15 |
| Arrival Time | 0 | 2 | 3 | 10 | 16 | 30 | 35 |

a) Construct a Gantt chart depicting the process scheduling for the set of processes specified in the above table of processes.

b) Calculate the average waiting time for the schedule constructed in part a.

If the process from the lower priority queue is preemted with the process from the higher priority queue, it will go to the end of the same queue and it will be sheduled again until it spends the remaining time quantum.



w1= 0+ (15-4) + (23-16) = 18
w2 = (4-2) + (20-8) = 14
w3 = (8-3) + (21-12) = 14
w4 = (12- 10) = 2
w5 = (16-16) + (34-20) = 14
w6 = (30-30) = 0
w7 = (35-35) = 0

w = 72/7= 8.86

4. Consider the following solution to a two-process critical section problem using the procedure *Swap* as defined below:

```
------------------------------------------------------
procedure Swap (var a, b: boolean);
var temp: boolean;
begin
    temp := a;
    a := b;
    b := temp;
end;
------------------------------------------------------
repeat

    key := true;
    repeat
        Swap(lock, key);
    until key = false;

        critical section

    lock := false;

        remainder section

until false;
------------------------------------------------------
```

a) Assuming *Swap* to be *non-atomic*, does the given solution satisfy the *mutual exclusion* requirement? Justify your answer with a detailed example with step-by-step explanation. .

b) Assuming *Swap* to be *atomic*, does the given solution satisfy the *bounded waiting* requirement? Justify your answer with a detailed example with step-by-step explanation.

a) Suppose that lock was freee when P1 call Sawp(lock,key). If the context switch occurs after instruction temp:=a; and the other process P2 calls the Swap(lock,key) procedure, then P2 will get the acces to the critical section. Now suppose that P2 was context switched inside the critical section and P1 runs again. P1 will also gain to the critical section, so mutual exclusion is violated.
b) Bounded waiting is not satisfied. if Priority(P1)>priority(P2) then P1 can enter critical section arbitrary many times before P2 gets into it.

5. The Hong Kong marriage registry has implemented a hi-tech marriage registration scheme. The operation of this scheme can be described as follows. Marriage officers wait until a bride and a groom arrive, and a ready to get married. Once teh couple is ready, the officer takes them to sign the marriage register. The office has a single marriage register which must be accessed exclusively to register a mariage. The couple wait until the registration is completed, before leaving. Once the registration is completed, the officer goes back to wait for another couple. This operation can be described using multiple threads representing the officers, brides and grooms. Implement a proper coordination mechanism between the officers, brides and grooms using semphores. The order of which bride gets married to which groom is unimportant. For your convenience, a skeleton structure for the threads are given below: (20 points)

// Globeal variables and their initial values

Semaphore mutex; Initial value = 1;
Semaphore BrideReady; Initial value =0;
Sempahore Groomready; Initial value=0;
Sempahore BrideLeave; Initial value=0;
Semaphore GroomLeave; Initial value=0;

**Table 1:**

| | |
|---|---|
| Officer() { | Bride() { |
| while(TRUE) { | //Indicate Bride ready |
| //wait unti bride and grrom are ready | BrideReady->V(); |
| BrideReady->P(); | // Wait until marriage completed |
| GroomReady->P(); | BrideLeave->P(); |
| // Access marriage register to register marriage | } |
| mutex->P(); | |
| // access marriage register | Groom() { |
| mutex->V(); | // Indicate groom ready |
| // Complete marriage registration | GroomReady->V(); |
| BrideLeave->V(); | // Wait until marriage completed |
| GroomLeave->V(); | GroomLeave->P(); |
| } // end while | } |
| } | |

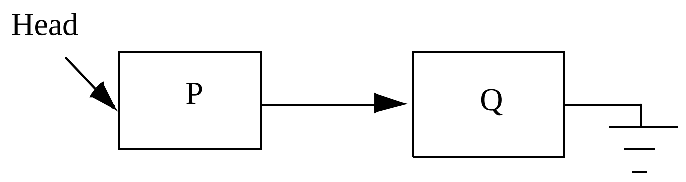
6. Consider the following two programs: (20 points):

**Table 2:**

| **Program P:** | **Program Q:** |
|---|---|
| int x,a,b; | int x,c,d; |
| | |
| main() { | main() { |
| x=0; | x=0; |
| // kernel threads | // PURE user threads |
| create kernel thread A; | create user thread C; |
| create kernel thread B; | create user thread D; |
| | |
| wait until threads A and B terminate; | wait until threads C and D terminate; |
| | |
| print a,b; | print c,d; |
| } | } |
| | |
| thread A() { | thread C() { |
| x=x+3; // statement (1) | x=x+4; // statement (5) |
| yield(); // kernel thread yield | yield(); // user thread yield |
| a=x+3; // statement (2) | c = x+4; // statement (6) |
| } | } |
| | |
| thread B() { | thread D() { |
| x=x+2; // statement (3) | x= x+1; // statement (7) |
| b= x+2; // statement (4) | d=x+1; // statement (8) |
| } | } |

*Note that Program P creates* ***kernel threads*** *and program Q creates* ***pure user threads****. Also, when a thread is created, it does not run immeditaely. It is first put into the corresponding ready queue.*

Two processes are forked to execute the programs P and Q respectively (P is forked first). before any of the processes start their execution, the kernel ready queue looks like this:



Both the kernel and the user thread library use FCFS scheduling policy.

Suppose during the program execution there are only two timer interrupts that force the kernel to do process/thread context switches. The two timer interrupts occur precisely between statements (3) and (4) and between statements (7) and (8) respectively. Besides these, threre are no other external interrupts nor exceptions.

a) Show the actual execution of the tsatements (i.e. Statement (1), (2) , etc.) and the changes in the kernel ready queue.

**Table 3:**

| Order | Kernel ready queue | Statement executed | process/Thread in CPU |
|---|---|---|---|
| 1 | head->P->Q->NIL | | |
| 2 | head->Q->A->B->NIL | | P |
| 3 | head->A->B->NIL | (5) | Q |
| 4 | head->A->B->NIL | (7) | Q |
| 5 | head->B->Q->NIL | (1) | A |
| 6 | head->Q->A->NIL | (3) | B |
| 7 | head->A->B->NIL | (8) | Q |
| 8 | head->A->B->NIL | (6) | Q |
| 9 | head->B->NIL | (2) | A |
| 10 | NIL | (4) | B |

b) What are the final values of the variables a, b c and d printed at the end of the programs ?

a=8
b=7
c=9
d=6