CSC 369H1 F
OPERATING SYSTEMS

UNIVERSITY OF TORONTO
FALL 2003

Midterm Test

NO AIDS ALLOWED

Please **PRINT** in answering the following requests for information:

Family Name: _____

Given Names: _____

Student Number: |_ _ _| |_ _ _| |_ _ _|

Login (@cdf): _____

Notes to students:

1. This test lasts for 50 minutes and consists of 50 marks. Budget your time accordingly.
2. This test has 7 pages and 5 questions; Check that you have all pages before starting.
3. **Write in pen. No pencils.**
4. Write your answers on this "question and answer" paper, in the spaces provided. Be concise. In general, the amount of space provided is an upper bound on the "size" of answer that is expected. If necessary, use space where available and provide explicit pointers.
5. In general, state your assumptions and show your intermediate work, where appropriate.
6. Do **not** go beyond here until instructed to do so. Write your student number at the top of each succeeding page once you get going.

Question	Marks
1	
2	
3	
4	
5	
Total	

VERSION A

../continued

1. [10 marks, 2 each] True/False

Circle "T" if the statement is *always* true. Otherwise circle "F".

a) In paging systems, external fragmentation cannot occur. **T**

b) Race conditions cannot occur on a uniprocessor. **F**

c) SJF can be implemented as a priority algorithm, where the priority is determined by the arrival time of the job. **F**

d) A process in the *Ready* state can only transition to *Running* or *Exit* states. **T**

e) The two-phase locking protocol guarantees that concurrent transactions are deadlock-free. **F**

../continued

2. [9 marks; 3 each]

Define the following terms in the context of this course:

(a) Starvation:

A condition in which a process is blocked indefinitely because another process is always given preference. May occur because of priority scheduling, or unfair synchronization algorithms.

(b) Page Frame

A fixed-size block of physical memory used in paging systems to hold parts of a process's address space. Frames are identical; any frame can be used to hold any page for any process.

(c) Turnaround Time

The difference between the time at which a process arrives, and the time at which it completes.

../continued

3. [10 marks; 2 each] Measurements of a certain system have shown that the average process runs for a time T before blocking on I/O. A process context switch requires a time S, which is effectively wasted (overhead). The performance measure of interest is CPU efficiency, defined as the ratio of useful CPU time over total CPU time. For round-robin (RR) scheduling with a quantum of length Q, *give* a formula for CPU efficiency in each of the following cases (be as specific as possible, give a range if appropriate):

a) $Q = \infty$

No involuntary context switches will occur. Each process will pay 1 context switch per CPU burst. Useful = T, Total = T + S

Efficiency = T / (T + S)

b) $Q > T$

As long as the quantum Q is larger than T, then no involuntary context switches will occur. Same as (a).

Efficiency = T/(T+S)

c) $S < Q < T$

Average process will have T/Q context switches per CPU burst, each with a cost S. Useful time is still T, Total is T + S*T/Q.

Efficiency = T/(T+ST/Q) = 1/(1 + S/Q)

Since S < Q, S/Q is < 1 and efficiency falls in the range (0.5, 1) or 50-100%

d) $Q = S$

Starting with the equation from C, the efficiency is now exactly 0.5

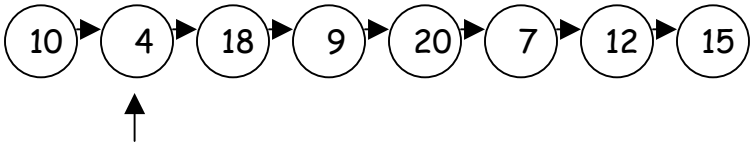
e) Q nearly 0 (or tending to 0)

Again use the equation from C, but as Q approaches zero, S/Q approaches infinity and the efficiency approaches 0.

../continued

4. [10 marks; breakdown as given below]

Consider a dynamic partitioning system in which memory consists of the following holes, sorted by increasing memory address (all sizes are in Kilobytes):

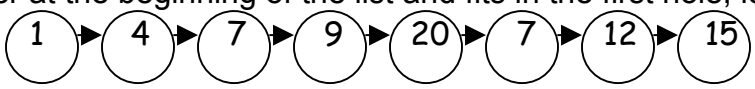


Hole created by last allocation

(a) **[8 marks]** Suppose a new process requiring 11 kB arrives, followed by a process needing 9 kB of memory. **Show** the list of holes **after both** these processes are placed in memory for each of the following algorithms (start with the original list of holes for each algorithm).

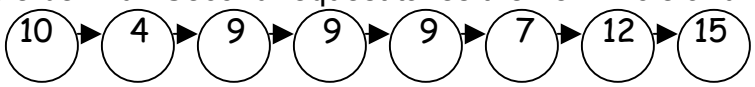
i) First Fit:

The first request fits in hole of size 18, leaving behind a 7kB hole. The second requests starts over at the beginning of the list and fits in the first hole, leaving behind a 1kB hole.



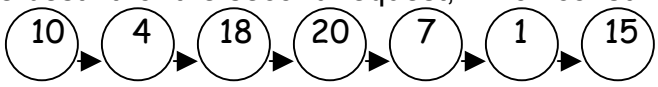
ii) Worst Fit:

The largest available hole is always used. First requests takes the 20kb hole and leaves a 9kB hole behind. Second request takes the 18kB hole and leaves a 9kB hole behind.



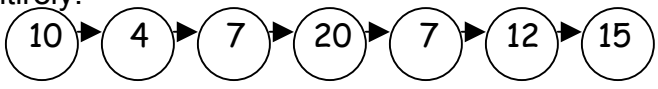
iii) Best Fit:

The 12kB hole is the best fit for the first request, leaving a 1kB hole behind. The 9kB hole is the best fit for the second request, which consumes the hole entirely.



iv) Next Fit:

Search begins where the last search finished. The 4kB hole was created by the last allocation, so that is where we start. The first request is placed in the 18kB hole, leaving a 7 kB hole behind. The search for the second request begins at this point and finds a 9kB hole which is consumed entirely.



(b) **[2 marks]** Keeping the hole list sorted by size can make some of the allocation algorithms faster. **State** one advantage of keeping the list sorted **by address** instead.

When blocks are freed, we would like the ability to *coalesce* adjacent free blocks into a single larger block. This is much easier if the free block list is sorted by address (we just have to look at the preceding and following block on the list once we find the insertion point for the newly freed block). Otherwise, all free blocks have to be examined to determine if coalescing is possible or not.

../continued

5. [11 marks]

Following an all-candidates meeting at city hall, politicians and reporters need to take an elevator that can only carry exactly 3 people at a time down to the ground floor to exit the building. (The elevator can come back up again empty). The elevator must carry 2 politicians and 1 reporter on each trip—politicians don't want to be outnumbered by reporters, and reporters don't want politicians to leave without answering any questions. There should be no unnecessary waiting: politicians and reporters should not wait if there is enough of them ready to form a "safe" elevator load.

A politician calls the procedure *PoliticianArrives* and a reporter calls the procedure *ReporterArrives* when they are ready to use the elevator. The procedures arrange the arriving politicians and reporters into "safe" elevator loads; once a full load is ready, one thread calls *EnterElevator* and after the call to *EnterElevator*, all three threads can leave. Assume that the *EnterElevator* procedure correctly blocks the caller until the elevator is ready for another load of passengers. Assume also that there are exactly twice as many politicians as reporters at the meeting.

Below is a semaphore-based implementation, using Posix semaphores and pthreads to represent politicians and reporters. Assume the semaphore waiting queues are FCFS.

```
/* Semaphores are initialized to 0 */
sem_t politician_ready;
sem_t reporter_ready;

void *PolitionArrives() {
   politician_ready.post();
   reporter_ready.wait();
   pthread_exit(0);
}
```

```
void *ReporterArrives() {
   politician_ready.wait();
   politician_ready.wait();
   EnterElevator();
   reporter_ready.post();
   reporter_ready.post();
   pthread_exit(0);
}
```

(a) **[2 marks] State** whether the implementation <u>works</u> (it is correct) or <u>does not work</u> (it is either always wrong or there exists an execution scenario where it is wrong).

Does not work.

(b) **[9 marks] Justify** your answer to (a) by **either** (i) **proving** that the implementation is correct, **OR** (ii) **showing** an execution scenario where the implementation is not correct and suggest a way to fix it (use the next page for your answer).

../continued

Execution scenario, assuming initial value of semaphores (Px indicates Polititian thread x, Rx indicates Reporter thread x):

P1: politician_ready.post() ← polititian_ready semaphore value = 1
P2: politician_ready.post() ← polititian_ready semaphore value = 2
P1: reporter_ready.wait() ← reporter_ready semaphore value = 0, P1 blocks
P2: reporter_ready.wait() ← reporter_ready semaphore value = 0, P2 blocks
R1: politician_ready.wait() ← decrements politician_ready value to 1 and returns
R2: politician_ready.wait() ← decrements politician_ready value to 0 and returns
R1: politician_ready.wait() ← R1 blocks, even though there are 2 politicians ready to leave

At this point we have violated the specification that says there should be no unnecessary waiting.

We can fix this with a third semaphore, one_reporter, initialized to 1. Each reporter should call one_reporter.wait() *before* waiting for the politicians, and one_reporter.post() before calling EnterElevator. This guarantees that one reporter can be matched with a pair of politicians before another reporter starts to be matched up. (This limits the rate at which elevator car loads can be formed, but since we only have a single elevator, this isn't a serious problem.)

Total marks = (50)
End of test

../continued