CMPT-300 Day: Mid-Term Examination

1 Time

16:30-17:20, Friday, Oct. 17th, 2003.

2 Directions

1. It is a closed book examination.
2. This examination has 3 pages and 6 questions with 27 marks. You need to answer all of these questions.
3. Please write all your answers on examination booklet(s). Do NOT write your answers on the question sheet(s).
4. Please write your answers clearly so I can follow your explanations.
5. Questions are not given in the order of difficult. Marks for each question are indicated in parentheses. Please spend your time accordingly.

3 Questions

3.1 (3 points)

What machine (hardware) features are needed to support protected operating system kernels? Please list three and briefly explain each of them.
Key:1. CPU working mode bit: indicating whether CPU is in the kernel or user mode
2. clock interrupt: preventing a user's process running for long time
3. base/limit registers: preventing the kernel in main memory from being accessed by other processes

3.2 (3 points)

A distributed system using mailboxes has two IPC primitives, *send* and *receive*. The latter primitive (*receive*) specifies a process to receive from, and blocks if no message from that process is available, even though messages may be waiting from other processes. There are no shared resources, but processes need to communicate frequently about other matters. Is deadlock possible? Please validate your answer in brief.
Key: Yes, it is possible. For example, process A is waiting for B's message, B is waiting for C's message, while C is waiting for A.

3.3 (3 points)

Please answer "Yes" or "No" to each of the following statements. You need NOT to explain your answers.

1. If the quantum of a Round-Robin scheduling algorithm is large enough, the algorithm will behave like first-come first-served (FCFS) algorithm.
2. Neither FCFS nor shortest job first scheduling algorithm has starvation problem.
3. Response time is not a goal of a scheduling algorithm in batch systems.

Key: 1)Yes 2)No 3)Yes

3.4 (4 points)

Given the three thread states: running, ready, and wait, state all possible thread transitions and show an example in brief of why a thread would follow such a transition.
Key:
Ready→Running: chosen to run by OS Scheduler
Running→Ready: time-sliced out or called Yield()
Running→Wait: I/O requests or wait on another thread
Waiting→Ready: I/O completion

3.5 (6 points)

```
BOOL flag[2]=FALSE; //shared variable, initial values are false

Process 0                              Process 1
{                                      {
```

```
  while(TRUE)
  {
    flag[0] = TRUE;
    while (flag[1]);
    critical-section;
    flag[0] = FALSE;
    remainder-section;
  }
}
```

```
  while(TRUE)
  {
     flag[1] = TRUE;
     while (flag[0]);
     critical-section;
     flag[1] = FALSE;
     remainder-section;
   }
}
```

Read the two processes above and answer the following questions:

1. Can the implementation achieve mutual exclusion? Please explain in brief.

2. What is the major problem(s) of this implementation? Please explain in brief.

Key: 1) Yes, it does achieve mutual exclusion. For example, suppose process 0 is inside the critical region, then flag[0] is true. Therefore, process 1 cannot enter the critical region.
2) Deadlock. Suppose process 0 executes "flag[0]=TRUE" and CPU switches to process 1 and executes "flag[1]=TRUE". Now we have a deadlock.

3.6 (8 points)

Consider the late-night pizza problem. A group of students are studying together. The students can study only while eating pizza. Each student executes the following loop: while (true) { pick up a slice of pizza; study while eating the pizza }. If a student finds that the pizza is gone, the student goes to sleep until another pizza arrives. The student who takes the last slice of pizza phones Satisfactions at PizzaHat to order another pizza before eating the pizzar. Each pizza has S slices. Assume initially there are alread S slices of pizza for students.

Write pseudo-code to synchronize the student threads and the pizza delivery thread. Your solution should avoid deadlock and phone Satisfactions (i.e., wake up the delivery thread) exactly once each time a pizza is exhausted. No piece of pizza may be consumed by more than one student. You may use the primitives provided below.

Primitive Semantics

- **lock(Mutex m):** acquires the lock on m; other threads invoking lock on m before it is unlocked are appended to the waiting queue on m and put to sleep.

- **unlock(Mutex m):** releases the lock on m; removes one thread(if any) from the waiting queue on m and awakens it.

- **wait(Condition cond, Mutex m):** appends the calling thread to the waiting queue on the condition variable *cond*; automatically unlocks *m*; the calling thread is then put to sleep.
- **signal(Condition cond):** removes one of the threads(if any) from the waiting queue on *cond* and awakens it.
- **broadcast(Condition cond):** removes all threads(if any) from the waiting queue on *cond* and awakens them

Key: (It is ok that some students use semaphores.)

```
mutex m;
Condition delivery, pizza;
int slice_num=S;
```

```
student_pick_pizza
{
  lock(m);
  while ( slice_num == 0 )
    wait(pizza, m);
  slice_num--;
  if ( slice_num == 0 )
    signal(delivery);
  unlock(m);
}
```

```
Satisfaction
{
   wait(delivery);
   lock(m);
   slice_num += S;
   for (i=0; i<S; i++) signal(pizza);
   unlock(m);
 }
```