CMPT-300 Day: Assignment 2

1 The deadline

Due at 4:30pm, Oct. 24th, Friday.

2 Where to submit

Please drop all your answers into the assignment drop-in box outside the room ASB 9838. I put a sign on the box which is reserved for our course (day section). There is another box there designated for the evenig section of the same course. So be careful.

3 Questions

In this assignment, you have **11** problems which count for **60** marks. You need to answer all of these questions. Please write your answers clearly.

3.1 (3 points)

What is race condition? Except for the *printer daemon* example demonstrated in the text book, figure out one example of your own for a potential race condition and describe it in one paragraph.

3.2 (8 points)

Read the solution for dinning philisophers' problem in textbook and answer the following questions:
(1) Why is the state variable set to HUNGRY in the function take_forks?
(2) Can we set the state variable to THINKING after (instead of before) the two calls to the function test in the function put_forks? Justify you answer.
(3) What is the semaphore *mutex* used for? What is the semaphore array *s[N]* used for?
(4) Some one argues that we do not need to use the semaphore mutex. Explain if we do not use it, what will happen.

3.3 (4 points)

(1) We know that the threads can be implemented in user space or in kernel space. What are the main advantages and disadvantages of for each of these two implementations?
(2) Now suppose that threads are implemented in user space. Can two threads in the same process synchronize using a kernel semaphore? Justify your answer using 1-2 sentences.

3.4 (2 points)

What is the advantage(s) for having different quantum sizes for different priority classes in multiple queues scheduling algorithm?

3.5 (4 points)

Suppose we have two processes P1 and P2 and need to synchronize their behaviors as follows

- *read(x)* can only be executed after *write(x)* is executed.
- *read(y)* can only be executed after *write(y)* is executed.

Please modify the pseudo-codes of P1 and P2 as shown below using semaphores to achieve the synchronization required.

```
P1()
{
  while (TRUE)
  {
   compute A1;
   write(x);
   compute A2;
   read(y);
  }
}
```

```
P2()
{
    while (TRUE)
    {
       read(x);
       compute B1;
       write(y);
       compute B2;
     }
  }
```

3.6 (4 points)

Suppose you have a multi-threaded application in which threads may execute the following procedure asynchronously. The procedure has a bug, in that it may lead to deadlock. Fix it so that it will never cause deadlock (under the assumption that the mutexes are used only by this one routine) without affecting the locking granularity (i.e., do not reduce or increase the size of critical region) and without making unnecessary changes.

```
const int ASIZE=16;   //shared by threads
```

```
mutex locks[ASIZE];   //shared by threads
int A[ASIZE];         //shared by threads

void Add(int one, int two)
{
    int temp;
    mutex_lock(&locks[one]);
    mutex_lock(&locks[two]);
    temp = A[one];
    A[one] = A[one]+A[two];
    A[two] = A[two] + temp;
    mutex_unlock(&locks[one]);
    mutex_unlock(&locks[two]);
}
```

3.7 (6 points)

Consider the following set of processes with their CPU times in milliseconds, arrival times in milliseconds and initial priorities. For example, process P1 arrives 2 milliseconds after the start of the system, has priority 3 and needs the CPU for 12 milliseconds.

Process	Burst Time	Arrival Time	Priority
P1	12	2	10
P2	23	6	20
P3	4	12	0
P4	18	0	30
P5	7	32	15
P6	27	24	10

For each of the scheduling algorithms listed below, please calculate the average turnaround time for this set of processes. Please show your work to receive full marks.

- Shortest Remaining Time Nex
- Round Robin (with a quantum of 5 ms):
- Non-preemptive Priority (higher number means higher priority; priority of a process increases by 1 for each millisecond in ready but not in running)

3.8 (9 points)

Suppose we have three kinds of threads: R-threads, G-threads and B-threads in a system. Each thread calls a *enter region* procedure before enterting its critical region and calls a *leave region* procedure after leaving its critical region. For example, a R-thread will call *R_enter* before entering and call *R_leave* after leaving. Similarily, G-threads and B-threads will call *G_enter, G_leave* and *B_enter, B_leave*, respectively.

The pseudo-codes of these six procedure calls are provided below. Also the explaination of synchoronization primitives used in codes are presented. Notice every primitive is atomatic.

Question: Please explain what synchronization(s) are achieved among three kinds of threads.

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

Pseudo-codes for the procedure calls

```
Mutex m;  //shared by all threads
Condition c_R, c_G, c_B; // condition variables shared by all threads
int n_R=0, n_G=0, n_B=0; //shared by all threads
```

```
void R_enter
{
  lock(m);
  while(n_B != 0)
    wait(c_R, m);
  n_R++;
  unlock(m);
}
```

```
void R_leave
{
  lock(m);
  n_R--;
  if((n_R == 0) && (n_G == 0))
    broadcast(c_B);
  unlock(m);
}
```

```
void G_enter
{
  lock(m);
  while((n_B != 0) || (n_G != 0))
    wait(c_G, m);
  n_G++;
```

```
void G_leave
{
  lock(m);
  n_G--;
  if (n_R == 0)
    broadcast(c_B);
```

```
  unlock(m);
}
```

```
  else
    signal(c_G);
  unlock(m);
}
```

```
void B_enter
{
  lock(m);
  while((n_R != 0) || (n_G!= 0))
    wait(c_B, m);
  n_B++;
  unlock(m);
}
```

```
void B_leave
{
  lock(m);
  n_B--;
  if(n_B == 0){
  {
     broadcast(c_R);
     signal(c_G);
  }
  unlock(m);
}
```

3.9 (12 points)

Please use the primitives provided in the last question to solve the problem Q50, in page 157 text book. Notice you only need to write pseudo-code for the procedures. And you can only use the primitives provided in the last question.

3.10 (4 points)

The code shown below for two threads A and B exhibits a well-known synchronization flaw. Briefly outline a scenario in which this code would fail, and the outcome of that scenario.

```
int x; //shared variable
void A
{
  while(1){
    sleep();
    access x;
   wakeup(B);
  }
}
```

```
void B
{
   while(1) {
     access x;
     wakeup(A);
     sleep();
   }
}
```

3.11 (4 points)

A system has four processes and five allocatable resources. The current allocation and maximum needs are as follows:

	Allocated	Maximum
P1	1 0 2 1 1	1 1 2 1 3
P2	2 0 1 1 1	2 2 2 1 4
P3	1 1 0 1 3	2 1 3 1 3
P4	1 1 1 1 0	1 1 2 2 1

`The available resource vector is: A=(0 0 x 1 y).`

Please determine the smallest value of $(x+y)$ for which this is a safe state using Banker's Algorithm. Please show your calculations.