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.
Key: Race condition: the final result on some shared data depends on which process runs precisely when. A possible example: consider a ticket reservation system. Assume there is only one ticket left. A process enquires the system and knows there is one ticket left. But before it updates the database (i.e., decrease the ticket number to zero), another process also send a query to the system and knows there is still one ticket left. As a consequence, bother threads will book that ticket.

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.
Key: 1) It indicates that this Philo wants to eat. If it is not set, the neighbour may force him to eat even he does not want to eat.
2)No. Because in the procedure *test*, its neighbour can only eat if he is in thinking state.
3) *mutex* is for mutual exclusion. It protects the shared variable *sate*. *Array s[N]* keeps track of the states of philos. For example, when one philo wants to eat, he needs to know if his neighbours are eating.
4) Race condition. Notice that the procedure *test* is always protected by the mutex. Suppose no such protection. Therefor, two philos can call *test* at the same time. Now Philo 0 wants to eat so he calls *test*, and he finds no neighbour is eating. But before he updates his state to eating. CPU switches to Philo 1. Philo 1 also want to eat and he calls *test* as well. (Remember, if *test* is protected by mutex, Philo 1 has no chance to call *test*). Also Philo 1 finds that no neighbour is eating (Philo 0 has not updated his state yet) so he goes to eat. When CPU switches back to Philo 0, he will update his state and goes to eat as well. So both Philo 0 and Philo 1 are eating, but they are neighbours.

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.
Key: 1) User-space: less overhead. difficult to handle blocking system calls. Kernel-space: more overhead. Easy to handle blocking system calls.
2)No. If one thread blocks, all other threads within the same process will also be blocked by the kernel as well because kernel only knows process. It may lead to deadlock because the threads which are in the critical region are also blocked.

3.4 (2 points)

What is the advantage(s) for having different quantum sizes for different priority classes in multiple queues scheduling algorithm?
Key: Reduce the overhead caused by job/process switching. The jobs in low priority queues usually are CPU-intensive. Giving them large quantum can reduce the number of job switching, thus reduce the context switching overhead.

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;
     }
  }
```

Key:

```
semaphore s1=0, s2=0;

P1()
{
  while (TRUE)
  {
    compute A1;
    write(x);
    up(&s1);
    compute A2;
    down(&s2);
    read(y);
  }
}

P2()
{
    while (TRUE)
    {
       down(&s1);
      read(x);
       compute B1;
  write(y);
       up(&s2);
       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]);
}
```

Key: add the following statement after the declaration "int temp;"

```
if (one > two) return Add(two,one)
```

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)

Key:

- Shortest Remaining Time Nex ((14 - 2) + (64 - 6) + (18 - 12) + (34 - 0) + (41 - 32) + (91 - 24)) / 6 = 186 / 6 = 31 ms
- Round Robin (with a quantum of 5 ms): ((46 - 2) + (79 - 6) + (29 - 12) + (54 - 0) + (66 - 32) + (91 - 24)) / 6 = 289/ 6 = 48.167 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) ((53 - 2) + (41 - 6) + (57 - 12) + (18 - 0) + (91 - 32) + (84 - 24)) / 6 = 268/ 6 = 44.67 ms

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 entering 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.
key: a) No two G-threads can access the critical region at the same time.
b) Multiple R-threads and up to one G-thread may access the critical region at the same time, during which time no B-threads may access the critical region.
c) Multiple B-threads may access the critical region at the same time, during which time no R-thread or G-thread may access the critical region.

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++;
  unlock(m);
}

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

bf Key:

```
int man_count=0, woman_count=0;
mutex m;
condition man, woman;

void woman_enter {
   lock(&m);
   if ( man_count == 0 ) woman_count++;
   else wait(man,m);
   unlock(&m);
 }

void man_enter {
   lock(&m);
   if ( woman_count == 0 ) man_count++;
   else wait(woman,m);
   unlock(&m);
 }

void woman_leave {
   lock(&m);
   woman_count--;
   if (woman_count == 0 ) broadcast(woman);
   unlock(&m);
 }

void man_leave {
   lock(&m);
   man_count--;
   if (man_count == 0 ) broadcast(man);
   unlock(&m);
 }
```

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

Key: The lost-wakeup call problem. Before process A calls *sleep()*, process B calls *wakeup()*.

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.

Key: Calculate the request matrix first:

	Request
P1	0 1 0 0 2
P2	0 2 1 0 3
P3	1 0 3 0 0
P4	0 0 1 1 1

At first, only P4 can be marked if $x \geq 1$ and $y \geq 1$. Now A=(1 1 1+x 2 y). Now, if $y \geq 2$, we can mark P1, then P3 and P2. If $x \geq 2$, we can mark P3, then P1 and P2. Therefore, the smallest value of $x + y$ is 3.