CS414 Assignment#1 : Solutions

(For question 3, we assume the priority changes as *p+t'*x*, since this is what most of you wanted. The solution is very similar if the priority changes as *p*t'*x*. Another assumption in question 3 is that a higher number => a higher priority)

1) The modified algorithm will not be able to solve the critical section problem. It violates the mutual exclusion condition in some cases. A case where it is violated is: Suppose there are 2 processes, **P1** and **P2**, and both reach line 5 at the same time. So both **P1** and **P2** read the values of *number[1] = number[2] = 0*. If **P2** first chooses a number, that is *number[2] = 1*. **P1** doesn't choose *number[1]* now, but delays the computation for some time. So **P2** is able to enter the critical section as *number[1] = 0*. Now **P1** computes *number[1] = 1*, since it had read a value of *number[2] = 0*. It then gets into the while loop in line 9, and is able to enter the critical section since *(number[1], 1)* < *(number[2], 2)*. Hence we have both **P1** and **P2** in the critical section at the same time. This violates the mutual exclusion condition and hence the variable *choosing* is required in the Bakery Algorithm, to solve the CS problem.

2) Semaphore Solution:
This solution is for the case when there is more than one barber!! You do not need a counting semaphore for barbers otherwise.

```
#define CHAIRS 10       // number of chairs available

   semaphore mutex=1, customers=0, barbers=0;
   int waiting = 0;        // #customers waiting

   void barber()
   {
      while(true)
      {
        P(customers);      // wait for a customer to arrive
        P(mutex);          // acquire lock on 'waiting'
        waiting--;
        V(mutex);           // release lock
        V(barbers);         // barber is now ready to cut hair
        cut_hair();
      }
   }

   void customer()
   {
      P(mutex);             // acquire lock on 'waiting'

      if (waiting >= CHAIRS)
      {
         V(mutex);          // no chairs are left so customer leaves
      }
      else
      {
         waiting++;
         V(customers);
         V(mutex);

         P(barbers);        // wait until we can get served
         get_hair_cut();
      }
   }
```

Monitor Solution:

```
type barbershop = monitor

 var waiting : integer; // customers waiting to be cut
 var customers : condition; // barber waiting for customer
 var barber : condition; // customer waiting for barber

 procedure entry get-customer() // called by the barber
   begin
     if waiting=0
       then customers.wait; // sleeps if there are none
     waiting := waiting-1; // one less customer waiting
     barber.signal; // frees a waiting customer
 end seek-customer;

 procedure entry get-haircut() // called by a customer
   begin
     if waiting < chairs // is there a free chair
       then
       {
         waiting := waiting+1;
         customers.signal; // signal barber
         barber.wait; // wait for barber to get free
       }
 end get-haircut;

 begin    // initialization
  waiting = 0;
 end
end barbershop;
```

3) When x<0, then priority goes down when a process is waiting in the ready queue. Similarly if y<0, then the priority goes down when the CPU is serving it. Let's see what happens when:
   a) (x<0 and y<0): If x>y then our scheduling algorithm works fine. If a process has been given more CPU time, its priority goes down, and hence other processes that are waiting, get time on the CPU. On the other hand if x>>y, then this results in frequent context switching. If x<y, then a process will never let go the CPU until completion. This results in something similar to an FCFS scheduling algorithm. The obvious disadvantages are that it is bad for time-sharing and could result in starvation.
   b) (x>0 and y>0): If x>y then our scheduling algorithm works fine. If a process has been given more CPU time, its priority goes up, but at a slower rate compared to waiting processes, and hence other processes that are waiting get time on the CPU. On the other hand if x>>y, then this results in frequent context switching. If x<y, then a process will never let go the CPU until completion. This results in something similar to an FCFS scheduling algorithm. The obvious disadvantages are that it is bad for time-sharing and could result in starvation.
   c) (x<0 and y>0): The priority would go down when waiting in the ready queue and go up when being served by the CPU. This is a very dangerous scheduling algorithm!! Not only do processes not relinquish the CPU once they acquire it, the other processes also lose priority. The disadvantages become more prominent in multiprocessor scheduling, where we will not be able to achieve a time-sharing system. Also since priorities cannot drop below 0, an x<<0 would be worse, since the priority difference between processes would begin to decrease.
   d) (x>0 and y<0): This is the ideal parameter, since a process loses priority when being served by the CPU and gains when waiting. But again an x>>0 or a y<<0 would be disadvantageous since this would result in frequent context switching. So while these parameters are good for a time-sharing system, it isn't good for a real time system.