CSI3131 – Operating Systems
Tutorial 5 – Process/Thread Synchronization

1. Give the reasons why Solaris, Windows XP, and Linux implement multiple synchronization mechanisms. Describe the circumstances under which they use spinlocks, mutexes, semaphores, and condition variables. In each case, explain why the mechanism is needed or desirable.

2. List the requirements a solution to Critical Section Problem must satisfy. Explain each one of them.

3. Why use semaphores (version 2, i.e. not spinlocks) when we have hardware instructions (test&set, xchng)?

4. Why use monitors when we have semaphores?

5. Consider simulating a bakery with a baker and several customers using the code provided on the next page. There is only one shelf with baguettes. When a customer walks in, he or she buys some baguettes by invoking a bakeryBuy() function. Since several customers can be present in the bakery, several invocations of bakeryBuy() can be executed simultaneously; note that all of them use the same shelf of baguettes (assume that one salesperson services the customers). The baker is represented with a thread/process that runs the bakeryBake() function that must also access the shelf.

   a) The incomplete code for the baker and for buying baguettes is provided on the next page. Use two semaphores to complete the code. The resulting code should work correctly even if there are several simultaneous executions of bakeryBuy(). The baker should bake baguettes (execute bake()) only at the beginning and when new baguettes are ordered from the bakeryBuy() function.

      To modify the code, all you have to do is
      - Specify the semaphores you are going to use (list their names) using the type SEMAPHORE, such as `SEMAPHORE sem1;`
      - Specify the initial value for each semaphore using the call initSem(), such as `initSem(sem1);`
      - place `wait()` and `signal()` calls in the code (ex: `wait(sem1), signal(sem1)`).

      Try to make the critical sections as small as possible, in order to limit unnecessary waiting.

   b) Assume that each customer buys at most 10 baguettes. Is it possible to have an empty shelf at least for a moment? Justify your answer.

   c) Describe a scenario resulting in more then 150 baguettes on the shelf.

   d) Modify your solution to question 1 so that at every moment there are at most 150 baguettes on the shelf. (For this solution, you can add statements different from signal() and wait().)

```
      /* define your semaphores here using type SEMAPHORE*/


      /* executed by the baker , running all the time*/
bakeryBake()
{
   /* initialize the semaphores here with the call initSem(sem, value) */


   /* some initialization */
    onShelf = 0;

   for(;;)
   {

        bake();                  /* bake 100 new baguettes */

        onShelf += 100;   /* add them to the shelf*/

   }

/* invoked by a customer  - buying  x baguettes */
bakeryBuy(int x)
{

    if (x<=0)  /* just browsing, not buying */

        return;

    if (x>onShelf)  /* cannot buy more then is available */

        x = onShelf;

    onShelf -= x;

    printf("Sold "+x+" baguettes.");

    if (onShelf < 50)

    {

        printf("Low on stock, ordering 100 more.");

    }

}
```

6. Consider simulating a roller coaster amusement ride with one car of capacity CAPACITY passengers. The passengers arrive, wait until the car is ready for boarding, then board (one by one). When the roller coaster car is full, it leaves for the ride. When it returns, the passengers leave the car (no leaving in the middle of the ride!). When all passengers have left the car, new passengers can start boarding.

Examine the following (that uses C-like pseudocode with calls similar to the previous question) solution for this problem:

```
/* shared variables and semaphores */
int onBoard = 0;
SEMAPHORE mutex;
SEMAPHORE boarding;
SEMAPHORE full;
SEMAPHORE empty;
SEMAPHORE leave;

/*initialize the semaphores*/
initSems()
{
   initSem(mutex,1);
   initSem(boarding, 0);
   initSem(full, 0);
   initSem(empty,0);
   initSem(leave,0);
}
```

Roller coaster car code:

```
while(true)
{
    signal(boarding); /* ready for boarding */
    wait(full);       /* wait until full */
    /* time for the ride */
    signal(leave);
    wait(empty);
    /* all passengers have left */
}
```

Passenger code:

```
while(true) {
    /* arrive at the ride, wait for roller coaster car */
    wait(boarding);
    wait(mutex);
    onBoard++; /* board the car */
    if (onBoard == CAPACITY) signal(full);
    else signal(boarding);
    signal(mutex);
    /* Enjoy the ride */
    printf("Yahoooooo, this ride is cool\n");
    wait(leave);
    wait(mutex);
    onBoard--; /* leave the car */
    if (onBoard > 0) signal(leave);
    else signal(empty);
    signal(mutex);
    /* enjoy the amusement park*/
    sleep(getRandomTime());
}
```

a) Is it possible that a passenger enjoys the ride before the roller coaster car has left (that is prints the message before the ride starts)? If no, justify why. If yes, correct the code – you may with to introduce a new semaphore or so. (No need to fully rewrite the code here, you may write/insert into the code provided on the previous page.)

b) Assume there are VIPs who would also like to take a ride. However a VIP is so important and has bodyguards, so he/she will use 5 spaces instead of 1. Moreover, VIPs don't like to wait: if VIP boards, the roller coaster car should leave even if it is nowhere near full. If there are less then 5 spaces left, the VIP cannot fit and the roller coaster car must leave (without the VIP) so it returns sooner and the VIP does not wait unnecessarily.

   Your task: Write the code for the VIP. Make sure that even if there are plenty of waiting passengers and the VIP process boards the roller coaster car first (assume that the VIP process has the same scheduling priority as the other processes).