CSI3131 – Operating Systems
Tutorial 5 – Process/Thread Synchronization - Solution

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.
   These operating systems provide different locking mechanisms depending on the application developers' needs. Spinlocks are useful for multiprocessor systems where a thread can run in a busy-loop (for a short period of time) rather than incurring the overhead of being put in a sleep queue. Mutexes are useful for locking resources. Semaphores and condition variables are more appropriate tools for synchronization when a resource must be held for a long period of time, since spinning is inefficient for a long duration.

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

   ***Mutual exclusion*: only one process can be in the critical section at any moment of time**
   ***Progress*: if some processes want to enter the CS and there is no process in the CS, eventually one of the processes will enter the CS.**
   ***Bounded Waiting*: (starvation free) a process waiting to enter the CS can be overtaken only finite number of times before being give access to the CS**

3. Why use semaphores (version 2, i.e. not spinlocks) when we have hardware instructions (test&set, xchng)?
   Solutions using hardware instructions involve busy waiting, while with semaphores we can block the process trying to get the CS and use the CPU to do other work. Also hardware instructions support mutual exclusion but require additional programming to meet the bounded waiting requirement. The semaphores are higher level constructs that allow more convenient programming.

4. Why use monitors when we have semaphores?
   The problem with semaphores is that the synchronization/mutual exclusion code is spread out over all processes accessing the shared variables. It is quite difficult to maintain, as an error in single process might cause disruption everywhere. Monitors are a higher level synchronization tool, which 1) encapsulates the shared variables, so that they can be accesses only through monitor methods; 2) provides automatic mutual exclusion to its functions/methods; and 3) contains all synchronization functions using conditional variables. This means that in order to ensure correctness in accessing the variable, it is enough to ensure correctness of the monitor.

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.

A sufficient number of customers may enter and buy baguettes while the baker is baking (executing bake()) to empty the shelf.

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

Baker is signaled to bake 100 baguettes. Before he can finish and add the baguettes to the shelf, another customer comes into the store and signals again to add baguettes to the shelf. The baker can add another 100 baguettes to the shelf before the number drops again to 50.

   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().)

/* solution for a */

```
      /* define your semaphores here using type SEMAPHORE*/
SEMAPHORE mutex;  /*only one thread/process can access the shelf */
SEMAPHORE needBaguettes;  /* to signal baker that baguettes are needed */
      /* executed by the baker , running all the time*/
bakeryBake()
{
   /* initialize the semaphores here with the call initSem(sem, value) */
   initSem(mutex,1);  /* only one thread/process can access the shelf */
   initSem(needBaguette,1);  /* set to 1,  to bake first batch */
   /* some initialization */
    onShelf = 0;
   for(;;)
   {
        wait(needBaguettes);
        bake();                  /* bake 100 new baguettes */
        wait(mutex);
        onShelf += 100;   /* add them to the shelf*/
        signal(mutex);
   }

/* invoked by a customer  - buying  x baguettes */
bakeryBuy(int x)
{
    if (x<=0)  /* just browsing, not buying */
        return;
    wait(mutex); /* only one customer at a time */
    if (x>onShelf)  /* cannot buy more then is available */
        x = onShelf;
    onShelf -= x;
    printf("Sold "+x+" baguettes.");
    if (onShelf < 50)
    {
        signal(needBaguettes);
        printf("Low on stock, ordering 100 more.");
    }
    signal(mutex);
}
```

/* solution for d */

```
      /* define your semaphores here using type SEMAPHORE*/
SEMAPHORE mutex;  /*only one thread/process can access the shelf */
SEMAPHORE needBaguette;  /* to signal baker that baguettes are needed */
int onOrder; /* set to TRUE when baguettes have been ordered
                and FALSE otherwise*/
      /* executed by the baker process/thread, running all the time*/
bakeryBake()
{
   /* initialize the semaphores here with the call initSem(sem, value) */
   initSem(mutex,1);  /* only one thread/process can access the shelf */
   initSem(needBaguette,1);  /* set to 1,  to bake first batch */
   /* some initialization */
    onShelf = 0;
   for(;;)
   {
        wait(needBaguette);
        bake();                  /* bake 100 new baguettes */
        wait(mutex);
        onShelf += 100;   /* add them to the shelf*/
        onOrder = FALSE;
        signal(mutex);
   }

/* invoked by a customer  - buying  x baguettes */
bakeryBuy(int x)
{
    if (x<=0)  /* just browsing, not buying */
        return;
    wait(mutex); /* only one customer at a time */
    if (x>onShelf)  /* cannot buy more then is available */
        x = onShelf;
    onShelf -= x;
    printf("Sold "+x+" baguettes.");
    if (onShelf < 50 && onOrder != TRUE)
    {
        signal(needBag);
        onOrder = TRUE;
        printf("Low on stock, ordering 100 more.");
    }
    signal(mutex);
}
```

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;
SEMAPHORE startRide;

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

Roller coaster car code:

```
while(true)
{
    signal(boarding); /* ready for boarding */
    wait(full);       /* wait until full */
    /* time for the ride */
    for(i=0; i<onBoard; i++)
       signal(startRide);
    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);
    wait(startRide);
    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.)

Yes, it is possible, as the passenger starts to enjoy the ride right after boarding, even if the car is still waiting for other passengers. One possible solution (provided on the previous page) is to have the car signal to passengers that it has left, and the passengers wait to leave before they start enjoying. There are several ways to accomplish that, one possibility is to have the car signal as many times as there are passengers. Another is to have the car signal only once, and have each passenger signal to the next passenger. In this latter case, the last passenger should not signal, to ensure that the number of signals and the number of waits on the "startRide" semaphore in one ride cycle are equal. (The variable counting the passengers "onboard" or the constant CAPACITY can be used for this function.)

Note that in both solutions the roller coaster car might finish the ride before all passengers have enjoyed the ride (executed their printed statement). Another semaphore might be used to tell the car that all passengers have printed their message.

A curiosity: The mutex semaphore is used to ensure the mutual exclusive access to the onBoard variable share among all processes/threads. This is a very natural use of mutexes for shared variables. But if you look at the logic of how the other semaphores are being used (boarding, leave) you will note that the mutex semaphore is note required.

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).

First attempt - a simple solution (note that the code is simply the Passenger code with a few changes):
VIPcode:

```
while(true)
{
    /* arrive to the ride, wait for the roller coaster car */
    wait(boarding);
    wait(mutex);
    if (onBoard > CAPACITY-5)
    {
         signal(full);  /* not enough space */
         signal(mutex);
    }
    else
    {
        onBoard+= 5;   /* board */
        signal(full); /* and have the car leave */
        signal(mutex);
        for(i=0; i<5; i++) wait(startRide);
        /* enjoying the ride */
        printf("Uhmmmm, this ride is ok\n");
        wait(leave);
        wait(mutex);
        onBoard-=5; /* leave the car */
        if (onBoard > 0) signal(leave);
        else signal(empty);
        signal(mutex);
       /* Go check out the rest of the amusement park */
        sleep(getVIPRandomTime());
    }
}
```

Notes:

- **in order for enjoying to work properly, the roller coaster car code must use onBoard to count the signals to startRide and not CAPACITY.**
- **Both in a) and here the mutex semaphore is not needed, as the boarding and leave semaphores ensure that at any moment only one passenger/VIP accesses the onBoard variable.**

Solution for the case of really important VIPs, that should not be overtaken by other passengers (if the scheduler that decides who is awaken after signal() does not like the VIPs). The idea is that the passenger waits until there are no VIPs and only then tries to board. It can happen that a VIP arrived while the passenger at the head of the line was waiting on the boarding semaphore, in such case the passenger releases the boarding semaphore without boarding and waits again on the noVIPs semaphore. (Note that this is not an ideal solution since the passenger at the head of the line is sent to the back of the line by the VIP).

New semaphores:

```
SEMAPHORE noVIPs;
```

Initialize semaphore:

```
initSem(noVIPs, 1)
```

New variables:

```
vips = 0; /* the number of VIPs waiting for boarding */
```

Passenger code:

```
while(true)
{
    while(true)
    {
        wait(noVIPs); /* wait until there are no VIPs */
        wait(boarding);
        if (vips>0)    /* vips appeared, don't board */
           signal(boarding);
        else
        {
        /* if there are still no VIPs, tell other passengers
           about it and proceed with  actual boarding */
           signal(noVIPs);
           break;
        }
    }
    /* the rest is the same as before */
    wait(mutex);
    onBoard++; /* board the car */
    if (onBoard == CAPACITY) signal(full);
    else signal(boarding);
    signal(mutex);
    wait(startRide);
    /* enjoying 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);
       /* Go check out the rest of the amusement park */
    sleep(getRandomTime());
}
```

VIPcode:

```
while(true)
{
    /* arrive to the ride, wait for the roller coaster */
    wait(mutex);
    vips++; /* tell the passengers there is a VIP waiting */
    signal(mutex);
    wait(boarding);
    wait(mutex);
    if (onBoard > CAPACITY-5)
    {
         signal(full);  /* not enough space */
         signal(mutex);
    }
    else
    {
        onBoard+= 5;   /* board */
        if (--vips == 0) /* this VIP is not waiting anymore */
           signal(noVIPs);/*if no more VIPs waiting, signal it*/
        signal(full); /* and have the car leave */
        signal(mutex);
        for(i=0; i<5; i++) wait(startRide);
        /* enjoying the ride */
        printf("Uhmmmm, this ride is ok\n");
        wait(leave);
        wait(mutex);
        onBoard-=5; /* leave the car */
        if (onBoard > 0) signal(leave);
        else signal(empty);
        signal(mutex);
       /* Go check out the rest of the amusement park */
        sleep(getVIPRandomTime());
    }
}
```