CS2106 Operating Systems
Tutorial 6
Synchronization II

1. [Semaphore] Consider three concurrently executing tasks using two semaphores S1 and S2 and a shared variable *x*. Assume S1 has been initialized to 1, while S2 has been initialized to 0. What are the possible values of the global variable *x*, initialized to 0, after all three tasks have terminated?

A	B	C
P(S2); P(S1); x = x*2; V(S1);	P(S1); x = x * x; V(S1);	P(S1); x = x + 3; V(S2); V(S1);

*Note: P(), V() are a common alternative name for Wait() and Signal() respectively.

2. [Semaphore] In cooperating concurrent tasks, sometimes we need to ensure that all N tasks reach a certain point in code before proceeding. This specific synchronization mechanism is commonly known as a **barrier**. Example usage:

```
//some code

Barrier( N );  //The first N-1 tasks reaching this point
               // will be blocked.
               //The arrival of the Nth task will release
               //  all N tasks.

//Code here only get executed after all N processes
// reached the barrier above.
```

Use semaphores to implement a **one-time use `Barrier()`** function **without using any form of loops.** Remember to indicate the variables declarations clearly.

3. [**Deadlocks**] We examine the stubborn villagers problem. A village has a long but narrow bridge that does not allow people crossing in opposite directions to pass by each other. All villagers are very stubborn, and will refuse to back off if they meet another person on the bridge coming from the opposite direction.

a. Explain how the behavior of the villagers can lead to a deadlock.

b. Analyze the correctness of the following solution and identify the problems, if any.

```
Semaphore sem = 1;

void enter_bridge()
{
   sem.wait();
}

void exit_bridge()
{
   sem.signal();
}
```

c. Modify the above solution to support multiple people crossing the bridge in the same direction. You are allowed to use a single shared variable and a single semaphore.

d. What is the problem with solution in (c)?

4. [**General Semaphore**] We mentioned that general semaphore (S > 1) can be implemented by using **binary semaphore** (S == 0 or 1). Consider the following attempt:

```
int count =  <initially: any non-negative integer>;
Semaphore mutex = 1;   //binary semaphore
Semaphore queue = 0;     //binary semaphore, for blocking tasks
```

```
GeneralWait() {
    wait( mutex );
      count = count - 1;
      if (count < 0) {
          signal( mutex );
          wait( queue )
      } else {
          signal( mutex );
      }
}
```

```
GeneralSignal() {
    wait( mutex );
    count = count + 1;
    if (count <= 0) {
        signal( queue );
    }
    signal( mutex );
}
```

Note: for ease of discussion, we allow the count to go negative in order to keep track of the number of task blocked on queue.

   a. The solution is **very close**, but unfortunately can still have **undefined behavior** in some execution scenarios. Give one such execution scenario to illustrate the issue. (hint: binary semaphore works only when its value S = 0 or S = 1).
   b. [Challenge] Correct the attempt. Note that you only need very small changes to the two functions.

5. (Discuss if time permits) [**Synchronization Problem – Dining Philosophers**] Our philosophers in the lecture are all left-handed (they pick up the left chopstick first). If we force **one of them** to be a right-hander, i.e. pick up the right chopstick before the left, then it is claimed that the philosophers can eat without explicit synchronization. Do you think this is a **deadlock free solution** to the dining philosopher problem? You can support your claim informally (i.e., no need for a formal proof).