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.

ANS:

Semaphore S1 is initialized to 1, which means it acts like a mutex lock and ensures mutually exclusive access to variable *x* for all three processes.

Because semaphore S2 has been initialized to 0, process A can execute only after it receives signal V(S2) from process C, which is after variable *x* was accessed by process C. V(S2) is in critical section guarded by S1 -> process A will certainly not proceed before C sends signal V(S1).

How many possible executions do we have? The number of executions is equal to the number of permutations of A, B, and C in which A happens after C. Which is 3!/2=3:

B-C-A -> x = 6
C-A-B -> x = 36
C-B-A -> x = 18
Possible outcomes are 6, 18, and 36.

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.

ANS:

```
int arrived = 0;  //shared variable
Semaphore mutex = 1;  //binary semaphore to provide mutual exclusion
Semaphore waitQ = 0;  //for N-1 process to blocks


Barrier( N ) {
   wait( mutex );
   arrived ++;
   signal( mutex );

   if (arrived == N )
         signal( waitQ )

   wait( waitQ );
   signal( waitQ );
}
```

[Note to instructors] Point out that this is not a **reusable barrier**, as the **waitQ** may have undefined value afterwards, e.g. when task interleave just before the "if (arrived == N)" □ multiple tasks can execute the first "signal(waitQ)" resulting in >0 waitQ at the end. Note that this does not affect correctness.

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)?

ANS:

a. Two villagers on different sides of the bridge trying to cross at the same time will lead to a deadlock.

b. The problem with this solution is that it allows only a single villager to cross at a time. A second villager crossing the bridge in the same direction cannot walk behind the first one and instead needs to wait for the first one to exit the bridge.

c. Let's use a single variable `crossing`, whose value is originally `0`. A positive value indicates the number of villagers currently crossing in one direction, and a negative value indicates the number of villagers currently crossing in the other direction.

```
Semaphore mutex=1;
int crossing = 0;

void enter_bridge_direction1()
```

```
{
   bool pass=false;
   while(!pass){
      mutex.wait();
      if(crossing>=0){
        crossing++;
         pass=true;
      }
      mutex.signal();
   }
}


void enter_bridge_direction2()
{
   bool pass=false;
   while(!pass){
      mutex.wait();
      if(crossing<=0){
        crossing--;
         pass=true;
      }
      mutex.signal();
   }
}


void exit_bridge_direction1()
{
    mutex.wait();
    crossing--;
    mutex.signal();
}

void exit_bridge_direction2()
{
    mutex.wait();
    crossing++;
    mutex.signal();
}
```

d. The problem with this solution is that it allows the villagers crossing in one direction to indefinitely starve the villagers crossing in the other direction.

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

ANS:

a. The issue is task interleaving between "`signal(mutex)`" and "`wait(queue)`" in `GeneralWait()` function. Consider the scenario where count is 0, two tasks A and B execute `GeneralWait()`, as task A clears the "`signal(mutex)`", task B gets to executes until the same line. At this point, count is -2. Suppose two other tasks C and D now executes `GeneralSignal()` in turns, both of them will perform `signal(queue)` due to the count -2. Since `queue` is a binary semaphore, the 2nd `signal()` will have undefined behavior (remember that we cannot have S = 2 for binary semaphore).

b. Corrected version:

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

```
GeneralWait() {                   GeneralSignal() {
    wait( mutex );                    wait(mutex);
    count = count -1;                 count = count + 1;
```

`    if (count < 0) {`	`    if (count <= 0) {`
`        signal(mutex);`	`        signal(queue);`
`        wait(queue)`	`    } else {` **//else added**
`    }` **//else removed**	`        signal(mutex);`
`    signal(mutex);`	`    }`
`}`	`}`

Using the same execution scenario in (a), task D will not be able to do the 2nd signal(queue) as the mutex is not unlocked. Either Task A or B can clears the wait(queue), then signal(mutex) allowing task D to proceed. At this point in time, the queue value has settled back to 0 ⍰ no undefined signal(queue).

Reference: D. Hemmendinger, "A correct implementation of general semaphores", Operating Systems Review, vol. 22, no. 3 (July, 1988), pp. 42-44.

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).

ANS:

The claim is TRUE. Informal argument below.

For ease of discussion, let's refer to the right-hander as **R**.

If **R** grabbed the right fork then managed to grab the left fork THEN
R can eat -> not a deadlock.

If **R** grabbed the right fork but the left fork is taken THEN
The left neighbor of R has already gotten both forks -> eating -> eventually release fork.

If **R** cannot grabbed the right fork THEN
The right neighbor of R has taken its left fork. Worst case scenario: all remaining left-hander all hold on to their left fork. However, the left neighbor of R will be able to take its right fork because R is still trying to get its right fork.