Counting Semaphore

1. Consider a semaphore S that uses blocking implementation. Suppose at the current time instant, a) 2 processes are holding S having successfully executed Wait(S) previously, and b) the blocked process list for S contains 4 Process Control Blocks (PCBs).

a) What is the value of S at the current time instant?

b) Suppose the value of S changes to -1 after executing **Signal(S) five times and Wait(S) two times** in some order. What is the minimum intermediate value that S can have during these operations? Justify your answer.

c) What is the largest value that S can ever have? Justify your answer.

Counting Semaphore

a) The current value of S is -4. This is because there are 4 PCBs in the blocked list of S.

b) Minimum value of S is -6 when the 2 Wait(S) operations are performed first.

c) The largest value that S can ever have is 2, because at most 2 processes can hold S simultaneously. This is true because the blocked list of S is non-empty when 2 processes have successfully executed Wait(S).

Multiplayer Game

2. Alice is playing a game that involves pairing apples with oranges. The game produces apples and oranges in separate baskets at random instants. For Alice to get points, she must **pair exactly 1 apple with 2 oranges** by picking them from the respective baskets. Each basket is protected by a semaphore; A for the apple basket and O for the orange basket. To access a basket, Alice must first acquire the corresponding semaphore.

Complete the code in the below figure to help Alice to play this game. You should only use the operations Wait(A), Wait(O), Signal(A) and Signal(O) in the boxes provided. You may use any number of these operations (including zero) in each of the boxes. **Your solution must be deadlock-free even in the presence of other players who may be executing a different code.**

```
[ 1 ]
// Pick 1 apple
[ 2 ]
If (atleast 2 oranges in basket) {
** Pick 2 oranges and pair with apple. **
[ 3 ]
} else {
[ 4 ]
** Drop 1 apple back in basket **
[ 5 ]
}
```

Multiplayer Game

```
[ 1: Wait(A) ]
// Pick 1 apple
[ 2: Signal(A)
     Wait(O) ]
If (atleast 2 oranges in basket) {
** Pick 2 oranges and pair with apple. **
[ 3: Signal(O) ]
} else {
[ 4: Signal(O)
     Wait(A) ]
** Drop 1 apple back in basket **
[ 5: Signal(A) ]
}
```

1. First acquire an apple and release semaphore A.
2. If there are at least 2 oranges available, then acquire them and release semaphore O.
3. Else release semaphore O, drop the apple back and release semaphore A.

- **There is no deadlock, because the two semaphores are never acquired together (no nesting).**

Semaphore Implementation

3. Describe how *Wait*(*S*) and *Signal*(*S*) of a semaphore can be implemented using a *TestAndSet* instruction, given the below semaphore structure definition.

***Hint**: The semaphore value and the process queue L are shared variables among different processes when those processes access the semaphore.*

```
typedef struct {
        int value;
        struct process *L;
} semaphore
```

Where is the Critical Section?

🟨 Critical section
🟩 Remainder section

```
//Current implementation of Wait(S)
S.value--;
if (S.value < 0) {
        block( );   { Add current process to S.L;
                    { Put current process to waiting state;
}

//Current implementation of Signal(S)
S.value++;
if (S.value <= 0) {
    wakeup(P);   { Remove a process P from S.L;
                 { Add process P to ready queue;
}
```

Not entire block() and wakeup() functions are in the critical section.

Semaphore Implementation

Shared variable: Boolean lock=false;

***wait*(*S*)**

```
while TestAndSet(lock);
S.value = S.value –1;
if (S.value < 0) {
        add current process to S.L;
        lock = false;
        Put current process to waiting state;
}
else lock = false;
```

Semaphore Implementation (Cont')

***Signal*(*S*)**

```
while Test-and-Set(lock);
S.value = S.value +1;
if (S.value <= 0) {
        remove a process P from S.L;
        lock = false;
        Add process P to ready queue;
}
else lock = false;
```