Qn1a) S value = -4 ✓

b) Minimum S value = -6. This occurs when Wait(S) executes 2 times when the blocked process list for S already contains 4 PCBs. So, S = -4 -2 = -6. It then executes Signal(S) five times to increase S to -1.
Wait(S) decreases S by 1 and blocks a process.
Signal(S) increases S by 1 and wakeups a process.

SC2005 Operating Systems

TUTORIAL FIVE

Process Synchronization- Part II

c) The largest value S can have is 0. When S value is 0, there are no more PCBs in the blocked process list as every pair of Wait(S) and Signal(S) has already completed. Wait(S) has to be executed before Signal(S). ✗

The largest value S can have is 2 as at most 2 processes can hold S simultaneously. This is as the blocked list of S is non empty when 2 processes have successfully executed Wait(S)

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.

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.**

Share 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 in waiting state;
}
else lock = false;
```

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

```
[ 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 ]
}
```

1: Wait(A) ✓ — acquire apple and release semaphore A

2: Wait(O) ✓ ~~Wait(O)~~ Signal(A)

3: Signal(O) ✓ ~~Signal(O)~~, ~~Signal(A)~~

4: ✗ Signal(O), Wait(A) — need to release semaphore O again and access semaphore A

5: Signal(A) ✓

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;
```

```
boolean TestAndSet(boolean *semaphore.value){
    boolean rv = *semaphore.value;
    *semaphore.value = 0;
     enqueue(semaphore.L);
    return rv;
```
✗

```
while(1){
   while(TestAndSet(&semaphore.value));
     critical section
    semaphore.value = 1;
    dequeue(semaphore.L);
}
```
✗