1. Problem:

    a. Assume you have a system with a static priority CPU scheduler. Assume the scheduler supports preemption. Describe what the program below prints if the priorities are set such that T2 has a high priority, T1 has the middle priority, and T0 has the low priority. Assume the system starts with only T0 executing. Assume the semaphore `mutex` is initialized to 1.

`void T0() {`
`printf("T0-S\n");`
`StartThread(T1);`
`printf("T0-E\n");`
`}`
	`void T1() {`
	`printf("T1-S\n");`
	`wait(mutex);`
	`printf("T1-1\n");`
	`StartThread(T2);`
	`printf("T1-2\n");`
	`signal(mutex);`
	`printf("T1-E\n");`
	`}`
		`void T2() {`
		`printf("T2-S\n");`
		`wait(mutex);`
		`printf(T2-1\n");`
		`signal(mutex);`
		`printf("T2-E\n");`
		`}`

    b. What changes, if any, would happen to the order the printfs if priority donation were added to the CPU scheduler? Priority donation is what in lottery scheduling paper was implementing via ticket transfers: if a high priority thread is waiting on a mutex held by a lower priority thread, it donates its priority for the next scheduling decision.

2. Assume you are building a special operating system that requires executing the expression:

$$z = F3(F1(x),F2(F3(y)));$$

where x, y, and z are integers and F1, F2, F3 are functions.

The functions F1() and F3() must execute on thread T1 while the function F2 needs to execute on thread T2.

    a. Your job is to write the code to force the expression to be evaluated regardless of the order they are run by the CPU scheduler. Note you will have to add code to all three functions below.

```
int z, x, y; //shared variables

//Declare shared variables and semaphore with initial values here.
```

void ComputeZ() {
 	
StartThread(T1);
 	
StartThread(T2);
 	
return;
}
	void T1() {
	
	}
		void T2() {
		
		}

b. Does your solution handle the following cases? If so, explain why. If not, explain the problem and how you could fix it.

   i. Would your solution handle the case when ComputeZ() is called twice in a row like:

```
x=...;  y=...; //Set x and y arguments
ComputeZ();
printf("z = %d\n", z);
x=...;  y=...; //Set x and y arguments
ComputeZ();
printf("z = %d\n", z);
```

   ii. Would your solution handle the case when ComputeZ() is called twice from

two different threads? For example:

```
void Z1() {
x=...; y=...; //Set x and y arguments
ComputeZ();
printf("z = %d\n", z); }

void Z2() {
x=...; y=...; //Set x and y arguments
ComputeZ();
printf("z = %d\n", z); }

StartThread(Z1);
StartThread)(Z2);
```