Problem 1 (10 points): Does the Peterson’s protocol for two-process mutual exclusion work correctly when process scheduling is strictly non-preemptive? Would it work correctly on a system that uses priority-based scheduling and the competing processes may have different priorities? Explain your answers for both these questions.

Problem 2 (10 points): (Problem 55, Chapter 2) Consider the procedure put_forks in Figure 2-47. Suppose that the variable state[i] was set to THINKING after the two calls to test, rather than before. How would this change affect the solution?

Problem 3 (20 points): Barrier synchronization problem: A parallel program contains N processes, which execute in synchronized phases (steps). After completing execution of phase i, each process waits for all the other processes to complete their i’th phase. For this purpose, each process executes a function called BarrierSynch(i), after executing the code for phase i. When the last process reaches the barrier and executes this synchronization function, it unblocks all the other (N-1) waiting processes to resume their execution for phase (i+1). This form of execution of steps and barrier synchronization repeats at each phase.

Write synchronization code using counting semaphores to implement the barrier described above. You should pay attention to race conditions where a process resumed from a barrier may quickly finish its execution of the next phase and start executing the barrier code for the next (i+1) phase while some other processes are still executing the barrier synchronization code for phase i.

Problem 4 (16 points): Consider a shared bathroom facility used by both men and women, but with the following policy. When a woman is in the bathroom, other women may enter but no men may enter, and vice versa. A sign on the bathroom door indicates which of three possible states it is currently in:
- Empty
- Women present
- Men present

To coordinate men and women in properly entering the bathroom, we need to write the following four procedures. The coordination is to be implemented using semaphores and any other required data structures.
Problem 5 (20 points): Write a monitor to solve the Reader-Writer problem with the following two requirements in addition to the usual ones regarding reader-writer and writer-reader mutual exclusion. The monitor would provide the following four interface procedures: `start-reading`, `finish-reading`, `start-writing`, `finish-writing`. A reader or writer would first execute the corresponding “start” procedure, perform its read or write operation on the shared data which is external to the monitor, and then execute the corresponding “finish” procedure of the monitor.

**Writer Process**

```
Monitor.start-writing();
Perform writes to the shared data;
Monitor.finish-writing();
```

**Reader Process**

```
Monitor.start-reading();
Perform read operations on the shared data;
Monitor.finish-reading();
```

In case both readers and writers are competing then the following policies will ensure fairness by giving alternating access to the readers and writers.

1. When a writer completes execution of `finish_writing()`, and if both writers and readers are waiting, allow all the waiting readers to proceed with reading but start blocking any new readers.
2. When any new readers arrive (i.e. executes `start_reading()` ) while some readers are reading and one or more writers are waiting, then the new readers should wait till a waiting writer is allowed to proceed with writing. Otherwise new readers can be allowed to proceed with reading if there are no waiting writers.
3. After the last reader finishes executing `finish_reading()`, and if both writers and readers are waiting, then one of the waiting writers should be allowed to proceed with writing.

Problem 6 (16 points): Consider a system with two producer processes, say P1 and P2, and two consumer processes, say C1 and C2. The producers and consumers communicate items through two buffers, Buffer1 of size N and Buffer2 of size M. All these are non-terminating processes, i.e. they endlessly produce or consume items.

Producer P1 deposits items in Buffer1, and producer P2 deposits items in Buffer2. The consumer processes extract items from either of the two buffers, whichever is non-empty.

Using counting semaphores, write code for the synchronization of the producer and consumer processes, addressing the following requirements:

- Your solution should not contain any busy waiting.
- Make sure that a consumer process does not get blocked, waiting to get an item from an empty buffer while the second buffer is non-empty.
- A producer process should not get blocked while another producer is depositing an item in its buffer.

Problem 7 (8 points): Read *Implementation of Processes and Threads in Linux* (Section 10.3.3, Modern Operating Systems). For each of the following statements, indicate if it is true or false.

a. In Linux, it is possible for a process to create a sibling process.
b. In Unix, it is possible for a process to create a sibling process.
c. In Linux it is possible for a process to create a new process and share file descriptors with that process. They share the same descriptors, not a copy.
d. In Unix, the fork call by a process creates a copy of its file descriptors for the child process. They do not share the file descriptors.
e. In Linux, it is possible for a process to create another process and share with it the current working directory.
f. In Unix, it is possible for a process to create another process and share with it the current working directory.
g. In Unix, a child process created by fork inherits signal handling settings of its parent.
h. In Unix, a child process created by fork inherits parent’s pending signals.