Operating Systems, Spring 2020, Exercise 2

1. Running track. Running track is 400m long. Ann and her friends come there to run 4000m. Ann is social and wants to wait for her friends in some ways depending on her mood. Sometimes her friendsmay be social also. Solve the resulting synchronization problems with semaphores and shared memory. Write the pseudocode for each runner. They will likely have different code each. Define your semaphores and other (global, local) data structures, with their initial values. Protect critical sections, if any. Make your solution as simple as possible, but still correct in all possible scenarios. You can start with the following pseudocode (that does not include synchronization solution yet):

```
                sem something = <init value>

   Ann                 Bill
   for (i=1 to 10)     for (i=1 to 10)
      <run lap>           <run lap>
      P(something)        ...
```

   a. Ann is running with Bill. After each lap, Ann waits for Bill to catch up with her (on the same lap, so that both have run equal number of laps). If Bill happens to be running ahead, he will keep on running and let Ann try to catch him up. Bill likes that. It happens so rarely that Ann does not mind. So Bill just runs for his 10 laps. However, on a good day Bill could pass Ann many times and finish his run when Ann still has many laps to go. Ann does not like that.

   b. There are three runners: Ann, Bill and Charlie. Ann and Bill wait after every lap until Charlie has caught up with her or him (equal number of laps). If Charlie is ahead of Ann or Bill, Ann and Bill do not wait. Ann and Bill do not wait for each other. Charlie does not wait for anybody.

   c. There are five runners: Ann, Bill, Charlie, Dorothy, and Eve. Ann waits after every lap until either (i) both Bill and Charlie have caught up with her (equal number of laps), or (ii) both Dorothy and Eve have caught up with her (equal number of laps). If both Bill and Charlie are ahead of Ann, Ann does not wait. If both Dorothy and Eve are ahead of Ann, Ann does not wait. The others do not wait for anybody.

2. **One-lane bridge with semaphores.** There runs a river between two villages A and B and the villages are connected by a narrow one-lane bridge, where cars are allowed to drive only in one direction in a time. Thre can be many cars on the bridge at a time, but of course in one direction only. When cars are passing the brigde from A to B (westbound), then the cars willing to cross the bridge from B to A (eastbound) have to wait. The car processes call `enter_bridge_direction` before using the bridge, and `exit_bridge_direction`, when they leave the bridge.

```
Process car [i = 1 to N] {
   .....
   enter_bridge_west()
      drive over the bridge westbound
   exit_bridge_west()
   ......
   enter_bridge_east()
      drive over the bridge eastbound
   exit_bridge_east()
   ......
   }
```

   Give the code for synchronization procedures (enter_bridge, exit_bridge). The solution must be based on semaphores and wait and signal operations. The solution does not need not to be *fair*, which means that the waiting times on the other end may be very long. When the waiting ends, the cars must be allowed to proceed in FCFS order. Remember to define and initialize all your semaphores and counters.

3. Redo the **One-lane bridge** problem (problem 2) with monitor.
   You need to write a monitor Bridge that solves the synchronization problem. The monitor Bridge should have four methods for synchronization: enter_west, exit_west, enter_east, exit_east. Remember to define all your condition variables and counters. Use Hoare's signalling semantics (signal and wait).

```
Process car [i = 1 to N] {
   .....
   Bridge.enter_west()
      drive over the bridge westbound
   Bridge.exit_west()
   ......
   Bridge.enter_east()
      drive over the bridge eastbound
   Bridge.exit_east()
   ......
   }
```

4. (Problems 6.7 [Sta18][Sta15], modified)
   A spooling system (see Fig. 6.17) consists of an input process I, a user process P, and output process O= connected by

two buffers. The processes change data in blocks of equal size. These blocks are buffered on a disk using a floating boundary between input and output buffers, depending on the speed of the processes. The communication primitives used ensure that the following resource contraint is satisfied:

```
i+o ≤ max
```

where

```
max = maximum number of blocks on disk
i = number of input blocks on disk
o = number of output blocks on disk
```

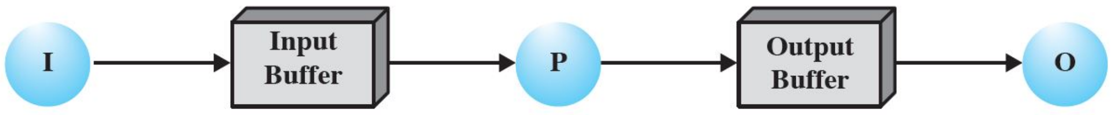

Figure 6.17 A Spooling System

The following is known about the processes:

1. As long as the environment supplies data, process I will eventually input it to the disk (provided disk space becomes available).
2. As long as input is available on the disk, process P will eventually consume it and output a finite amount of data on the disk for each block input (provided disk space becomes available).

Show that this system can become deadlocked.
Show a resource allocation graph for deadlock situation. Where is the loop?
How would you avoid this problem?

5. Overlapping critical sections. Operating system kernel for a multicore system has time critical code (routines K1, K2, ..., K7) with three short critical sections (A, B, C), that may overlap. Sometimes K1 needs B, and then later on maybe also A (while still in B). Sometimes K2 wants C and then maybe also A (while still in C). K3 always wants B and then maybe also C (while still in B). K4 wants C and then later on possibly also B (while still in C). Still other kernel routines (K5-K7) need A, B , or C alone every now and then.
   How do you solve this problem with three overlapping critical sections without deadlock?
   Which method do you use to protect critical sections?

6. Recovering from (danger of) deadlocks
   a. A 9-threaded application has one main program thread and 8 worker threads. Overall problem is divided into 10000 subtasks, and any of the 8 worker threads can compute any subtask. Subtasks are almost independent of each other. Main program gives subtasks to worker threads, and keeps track on which subtasks have been completed. Whenever any worker thread completes a subtask, the main program gives it another one to complete. Subtasks take minutes to run, and overall program takes hours to run on a 8-core system. Because of the selected programming paradigm, there is a slight chance that any two threads running different subtasks can deadlock. Changing the programming paradigm is not possible. How can this (deadlock) problem be avoided in the main program so that we can quarantee the application (and all subtasks) is completed in a reasonable time?
   b. In what type of environment would Banker's algorithm be used? How would it be used in that case? What changes do you need to make to the application if Banker's algorithm is used?