CISC 324, Winter 2018, Assignment 2

This assignment is due in lecture Tuesday Jan 30

Lab 3 is due one week later, in lecture Tuesday Feb 6

If you can't make it to lecture, then hand the assignment/lab in early at the School of Computing Office in Goodwin 557. After hours, you can slide the assignment under the door of Goodwin 557; label it as "for Professor Blostein".

Readings for Assignment 2

In the course reader:

Pages 19-27 Programming language constructs for creating processes; unpredictable outcomes; critical sections; semaphores; cobegin/coend and precedence graphs.

Pages 27-33 Readers and writers problem: an in-depth example of how to use semaphores. Read this in preparation for lab 3. Assignment questions about Readers and Writers are with assignment 3.

In the textbook:

Sections 3.1 to 3.4 Processes: scheduling, operations, communication

Section 5.1 (6.1 in 8th edition) Introductory information for process synchronization

Section 5.6 (6.5 in 8th edition) Semaphores. Skip the busy-waiting definition of *semaphore* at the start of Section 5.6. Instead, focus on the semaphore definition in section 5.6.2 that uses a list of processes (a list of PCBs). These two definitions are consistent with each other: they define the same behaviour for the semaphore operations *wait* and *signal*. However, the definition in section 5.6.2 has the advantage that busy-waiting is avoided: a waiting process becomes blocked (the operating system puts the PCB on a semaphore queue) rather than wasting CPU time constantly checking whether the wait is over.

Assignment 2 Questions

(1) A process is said to be "a program in execution". Briefly describe how the operating system maintains the connection between a process and its corresponding program. Illustrate how it is possible for several processes to be executing the same program (for example: several users simultaneously running the same text editor).

(2) When a process isn't executing, the PCB (process control block) stores the information needed for later restart of the process. Which of the following information should be stored in the PCB, and which does not need to be stored there? Briefly justify your answers.

a. The values in the general-purpose CPU registers.

b. A copy of the program (the machine code) that the process is executing.

c. A copy of the disk areas (the files) that the process has access to.

Note: *Hyperthreading* reduces the need to constantly copy values between registers and PCBs. In a hyperthreading architecture, the CPU chip provides each thread with its own physical copy of the registers, as described in course reader page 76. (Hyperthreading will not be tested on exams.)

(3) J and K are shared variables stored in main memory, and initialized to 1. What are the possible outcomes (final pairs of values for J and K) if the following code is executed concurrently?

```
process 1          process 2
J = K+3;           K = J+J;
```

To answer this question, you have to think in terms of machine instructions because a context switch can occur after any machine instruction. The compiler translates the above code to machine instructions such as the following.

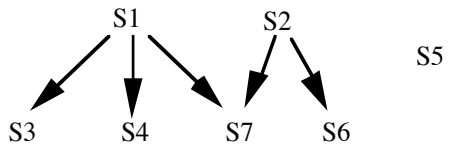
```
  process 1            process 2
A1  MOV R1, K        B1  MOV R3, J
A2  ADD R1, 3        B2  ADD R3, J
A3  MOV J, R1        B3  MOV K, R3
```

Hint: To find all possible outcomes, look at all possible orderings of the instructions that access shared memory. In this problem you have to consider all possible orderings of A1, A3, B1, B2, B3. Since A2 does not access shared memory, it does not interact with other processes and you can lump it together with A1 or A3 in the instruction ordering. Remember that A1 must execute before A3; also B1, B2, B3 must execute in that order. In summary: you need to find all ways to interleave the list A1 A3 with the list B1 B2 B3. You can treat A2 as bundled with instruction A1, or bundled with instruction A3.

(4) Draw a precedence graph that illustrates the parallelism present in the following code. Start by drawing nine graph nodes, labeled S0 to S8. Then draw an edge to connect node Si to node Sj if computation Si must complete before computation Sj can begin. (Reminder: "cobegin/coend" stands for "concurrent begin / concurrent end". Sometimes this is called "parbegin/parend", short for "parallel begin / parellel end". See pages 24-27 of the course reader for examples.)

```
begin
     S0
     cobegin
          S1
          S2
          begin
               S3
               S4
          end
          begin
               cobegin
                    S5
                    S6
               coend
               S7
          end
     coend
     S8
end
```

(5) (a) Write cobegin/coend code for the following precedence graph. Make your program express as much parallelism as possible within the limitations of cobegin/coend, while being sure to enforce all the constraints that are in the precedence graph. (The best solutions I know of introduce two or three extra precedence edges. Try to find one of these solutions.)



(b) Express the precedence graph from part (a) using semaphores. Show the initial value of each semaphore. Hint: As is discussed in lecture, one way to do this is to define semaphores with names such as S1_done, S2_done. If Si has to complete before Sj starts you could say:

process i	process j
<code for Si>	Acquire(Si_done)
Release(Si_done)	<code for Sj>

(6) Processes P and Q are executing concurrently, using a shared semaphore named *sem* that is initialized to 3.

```
var sem: semaphore := 3;
Process P                        Process Q
loop                             loop
    acquire(sem);                    printf("Q speaking");
    printf("P speaking")             release(sem)
endloop                          endloop
```

Pnum and Qnum are the number of times processes P and Q have printed their messages. Choose the correct answer, and write a brief justification.

(a) Pnum = Qnum-3 (b) Pnum = Qnum+3 (c) Pnum ≤ Qnum-3 (d) Pnum ≤ Qnum+3

(e) Pnum ≥ Qnum-3 (f) Pnum ≥ Qnum+3 (g) Pnum does not depend on Qnum (h) none of the above

(7) The following processes are executing concurrently. This code does not occur in a loop: each process Pi executes its worki_code exactly once.

Process P1	Process P2	Process P3	Process P4	Process P5
<pre-code>	<pre-code>	<pre-code>	<pre-code>	<pre-code>
work1_code	work2_code	work3_code	work4_code	work5_code
<post-code>	<post-code>	<post-code>	<post-code>	<post-code>

(a) Use semaphores to enforce the following constraint: Process P1 can start executing work1_code only when two or more of the other four processes have reached their <post-code>. State the initial value of any semaphore(s) you use. Add the necessary acquire and release calls to the <pre-code> and <post-code> sections of the processes.

(b) Use semaphores to enforce the following constraint: Process P1 can start executing work1_code only when **at least one** of the following is true:

- P2 has finished work2_code and P3 has finished work3_code

or - P4 has finished work4_code and P5 has finished work5_code

Your solution should not force P2 or P3 or P4 or P5 to wait in the <post-code>. Each of these processes needs to be able to execute their <post-code> without having to wait for other processes to finish. Your code can use shared variables (such as counters) in addition to semaphores, or you can create additional processes to help with the synchronization.

A note about lab 3

Lab 3 is described on pages 93-97 of the course reader. This lab is challenging because it is your first time writing synchronization code. As a first step, study and execute the given code where readers have priority (course reader page 29). Make sure that you can tell from looking at the output that readers are indeed getting priority. Then plan how to create code that gives priority to writers. As it says in step 3 of the lab instructions, first decide what semaphores and integer counters you need, and then figure out how to test the counters to decide whether a thread needs to acquire some semaphore or release some semaphore. For ideas on how to design and implement this type of synchronization, study the starvation free solution given on pages 31-33 of the course reader (section 3.3 Blostein Code: Semaphores other than mutex are initialized to 0).