Assignment 2 due Friday Jan 30, in lecture
Lab 3 is due one week later, Friday Feb 6, in lecture

If you have to miss the lecture, then hand the assignment/lab in early: bring it to the School of Computing Office, 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 and Lab 3

In the course reader:
- Pages 17-19  Review this information on “Processes versus Interrupt Handlers”
- Pages 20-29  Language constructs for creating processes; unpredictable outcomes; critical sections; semaphores; precedence graphs; readers and writers problem
- Pages 33-37  A starvation-free solution to the readers/writers problem (using semaphores)
- Pages 76-79  Lab 3 description

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.

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?

   \[
   \text{process 1: } J = K+3; \\
   \text{process 2: } K = J+J;
   \]

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

   \[
   \begin{align*}
   \text{process 1: } & A1 \text{ MOV R1, K} \\
   \text{process 2: } & B1 \text{ MOV R3, J} \\
   \text{process 1: } & A2 \text{ ADD R1, 3} \\
   \text{process 2: } & B2 \text{ ADD R3, J} \\
   \text{process 1: } & A3 \text{ MOV J, R1} \\
   \text{process 2: } & B3 \text{ MOV K, R3}
   \end{align*}
   \]

   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. A2 does not access shared memory, so it does not interact with other processes and therefore you do not have to consider it in the ordering of instructions. 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 / parallel end”. See pages 25-28 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
  <code for Si>
  Acquire(Si_done)
  Release(Si_done)
  <code for Sj>
```

Lab 3

Lab 3 is described on pages 76-79 of the course reader. This lab will probably take a couple of hours. First get a good understanding of the given code: semaphores are used to implement a solution to the Readers/Writers problem where readers have priority. Then start planning how to change the code so that writers have priority. As it says in step 3 of the lab instructions, it is very important to (1) plan what semaphores and integer counters you need, (2) state the conditions under which a thread should wait on a particular semaphore, and (3) state the conditions under which a thread should signal a particular semaphore. For ideas on how to do this type of synchronization, study the starvation free solution given on pages 33-37 of the course reader. **Plan carefully before you code.**