1. (20 points) Consider the following *incorrect* \( n \)-process mutual exclusion algorithm.

```plaintext
const n : integer                    /* The number of processes in the system */
var choosing : array 0..n-1 of boolean /* All entries are initialized to false */
var number   : array 0..n-1 of integer /* All entries are initialized to 0 */

process P(i : integer)               /* i is the identity of the process */
loop
    choosing[i] := true               /* MAX is a function that returns a value */
    number[i] := MAX( number) + 1     /* that is greater than or equal to the */
                                        /* largest value stored in the array. */
    choosing[i] := false
for k := 0 to n-1 by 1 do
    while (choosing[k]) do
        NO_OP
    end while
    while (number[k] ≠ 0 and number[k] < number[i]) do
        NO_OP
    end while
end for
<Critical section>
number[i] := 0;
end loop
end P_i
```

Explain why this algorithm is incorrect and modify it so that it will work correctly. In answering this question you should state what it means for such an algorithm to be correct.
2. (20 points) Consider the following program:

```
P1: {
    shared int x;
    x = 10;
    while (1) {
        x = x - 1;
        x = x + 1;
        if (x != 10)
            printf(``x is %d'',x);
    }
}
```

```
P2: {
    shared int x;
    x = 10;
    while (1) {
        x = x - 1;
        x = x + 1;
        if (x != 10)
            printf(``x is %d'',x);
    }
}
```

Note that the scheduler in a uni-processor system would implement pseudo-parallel execution of these two concurrent processes by interleaving their instructions, without restriction on the order of the interleaving.

a) Show how (i.e., trace the sequence of inter-leavings of statements) such that the statement “x is 10” is printed.

b) Show how the statement “x is 8” is printed. You should remember that the increment/decrements at the source language level are not done atomically, i.e., the assembly language code:

```
LD     R0,X         /* load R0 from memory location x */
INCR   R0           /* increment R0 */
STO    R0,X         /* store the incremented value back in X */
```

implements the single C increment instruction (x=x+1).

3. (10 points) Show where/how to add semaphores to the program in the previous problem to insure that the `printf()` is never executed. Your solution should allow as much concurrency as possible.

4. (10 points) Is a busy-waiting implementation of `UP()` and `DOWN()` less efficient (in terms of processor time) than a blocking implementation in all situations? Explain.

5. (20 points) The CSCE 451/851 Teaching Assistant holds office hours twice a week in his office. His office can hold 2 persons: 1 TA and 1 student. Outside his office are 4 chairs for waiting students. If there are no students waiting to see the TA, the TA plays tetris. If a student arrives at the TA’s office and the TA is playing tetris, the student loudly clears his throat and the TA invites the student in and begins helping her. If a student arrives at the TA’s office and the TA is busy with another student, the student waits in a chair outside the TA’s office until the TA is free. If the arriving student finds all the chairs occupied, then he leaves.

Using semaphores, write two process, `student_i` and `TA`, that synchronize access to the TA’s office during his office hours. These processes will have approximately the following structure

```
Process TA
Loop
    <Entry protocol to synchronize with a student>
    <Advise a student>
    <Exit protocol>
end loop
end TA
```

```
process student_i
    <Entry protocol to synchronize with the TA>
    <Get advice or leave>
    <Exit protocol>
end student_i
```

(Note that although you are writing one student process, assume multiple instances of the process are active simultaneously.)
6. (20 points) The Dining Philosophers Problem [Dijkstra, 1968]. Consider five philosophers seated around a circular table on which there are five plates of pasta and five forks (one fork to the right of each plate of pasta). The philosophers spend their lives thinking and eating. While the philosophers think, they ignore the pasta and do not require a fork. When a philosopher decides to eat, she must obtain two forks, one from the left of the plate and one from the right of the plate. Assume a philosopher can pick up only a single fork at a time, but must obtain both forks before she can eat. After consuming food, the philosopher replaces the forks and resumes thinking. A philosopher to the left or right of a dining philosopher cannot eat while the dining philosopher is eating, since forks are shared.

Using semaphores, write the pseudo-code for process \textit{philosopher}_i that synchronizes access to the forks such that the philosophers alternate between eating and thinking. Make sure the philosophers do not end up in a deadlock state, and that no philosopher starves to death. Your solution should allow more than one philosopher at a time to eat.

7. (15 points) Consider the following implementation of a general (counting) semaphore. This implementation assumes the existence of binary semaphore operations \textit{up}_b and \textit{down}_b implemented with a test-and-set instruction. (Thus, the \textit{down}_b operation busy waits to block.)

\begin{verbatim}
Procedure DOWN( S : semaphore):
    Down_b(mutex)
    S := S - 1
    If (S < 0) then
        up_b(mutex)
        down_b(delay)
    endif
    up_b(mutex)
end DOWN

procedure UP( S : semaphore):
    down_b(mutex)
    S := S + 1
    if (S \leq 0) then
        up_b(delay)
    else
        up_b(mutex)
    endif
end UP
\end{verbatim}

For each of the following scheduling policies, will the above code yield a correct implementation of a semaphore? (That is, assume a set of processes call \textit{UP} and \textit{DOWN} to coordinate their activities and that these processes are scheduled by one of the policies below. For each policy, explain the effect of the policy, if any, on the coordination of the processes.)

\begin{enumerate}
    \item First-Come-First-Served
    \item Shortest-Job-First (you may assume either a preemptive or non-preemptive version)
    \item Priority (you may assign whatever priorities to the processes you wish)
    \item Round-Robin
\end{enumerate}

8. (20 points) Show that message passing and semaphores have equivalent functionality by

\begin{enumerate}
    \item Implementing message-passing using general semaphores and shared memory. (\textit{Hint}: Make use of a shared buffer area to hold mailboxes, each one consisting of an array of message slots.)
    \item Implementing a general semaphore using message-passing. (\textit{Hint}: Introduce a separate synchronization process.)
\end{enumerate}

9. (15 points) Show how monitors can be used to implement semaphores. Write a monitor with entry routines \textit{UP} and \textit{DOWN} that implements a general semaphore.