1. Consider the following incorrect $n$-process mutual exclusion algorithm.

```pseudocode
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
```

Explain why this algorithm is incorrect and modify it so that it will work correctly.

This algorithm is known as the “bakery algorithm.” The basic idea behind this algorithm is that processes who wish to enter the critical section get a “ticket” from a ticket machine similar to the ones found in bakeries that dispense a monotonically increasing sequence of tickets. Processes are then “serviced” (allowed to enter the critical section) in the order in which they received tickets. In theory this order is the same as the order in which they started to execute the critical section entry protocol (i.e., FCFS). Because the scheduling is FCFS, the protocol is expedient and has bounded waiting. Moreover, so long as the ticket dispenser gives each process a unique ticket, the protocol is correct (ensures mutual exclusion).

The problem, however, is that the dispenser may give two processes the same ticket. If this happens, two processes may simultaneously fail the test in the main synchronization loop and enter the critical section simultaneously. Thus the code in this loop needs to be changed to handle this case. The following change to the synchronization logic fixes the problem:

```pseudocode
while (num[k] ≠ 0 and (num[k] < num[i] or (num[k] = num[i] and k < i) do
  NO_OP
```

That is, if two processes have the same ticket number, then we break the “tie” in favor of the lower numbered process. (The higher numbered process will wait until the lower numbered process exits the critical section and resets its number.)
5. 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
  begin
    down(mutex)
    if (numWaiting < numChairs) then
      numWaiting += 1
      up(newStudent) /* Clear throat */
    else
      up(mutex) /* Leave without advice */
    endif
    up(TAisFree) /* Sync with TA */
  end student_i
```

(Note that although you are writing one student process, assume multiple instances of the process are active simultaneously.)

This solution uses binary semaphores for students to wait on and for the TA to play tetris on.
7. Consider the following implementation of a general (counting) semaphore. This implementation assumes the existence of binary semaphore operations \( \text{up}_b \) and \( \text{down}_b \) implemented with a test-and-set instruction.

Procedure \( \text{DOWN}(S : \text{semaphore}) \):
\[
\begin{align*}
\text{Down}_b(\text{mutex}) \\
S &:= S - 1 \\
\text{If } (S < 0) \text{ then} \\
\text{up}_b(\text{mutex}) \\
\text{down}_b(\text{delay}) \\
\text{endif} \\
\text{up}_b(\text{mutex}) \\
\end{align*}
\]

End \( \text{DOWN} \)

Procedure \( \text{UP}(S : \text{semaphore}) \):
\[
\begin{align*}
\text{down}_b(\text{mutex}) \\
S &:= S + 1 \\
\text{if } (S \leq 0) \text{ then} \\
\text{up}_b(\text{delay}) \\
\text{else} \\
\text{up}_b(\text{mutex}) \\
\text{endif} \\
\end{align*}
\]

End \( \text{UP} \)

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 \( \text{UP} \) and \( \text{DOWN} \) 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.)

a) First-Come-First-Served

b) Shortest-Job-First (you may assume either a preemptive or non-preemptive version)

c) Priority (you may assign whatever priorities to the processes you wish)

d) Round-Robin

To begin, note that this semaphore implementation differs from those considered in class in that this semaphore can take on negative values and there is no additional variable to keep track of the number of processes waiting for the value of the semaphore to be greater than 0. In this solution the number of waiting processes is encoded in the semaphore value. When the semaphore is less than 0, then the absolute value of the semaphore is the number of waiters.

Since the entire body of each procedure is a critical section and since mutual exclusion is achieved via busy-waiting, preemptive, priority driven scheduling policies will lead to livelock as described in (1) above. However, because these procedures also include condition synchronization (achieved via the “delay” binary semaphore), non-preemptive scheduling policies will not work either. This is because with a non-preemptive scheduling policy, once a process busy-waits for condition synchronization, no other process will ever execute and in particular, no process can ever make the awaited condition true. Thus non-preemptive scheduling policies will lead to livelock as well.

This leaves only the scheduling policies that are preemptive but assign priorities in a dynamic manner (i.e., policies wherein the execution priority of a process changes over time). The best example of such a process is RR.

The message from this exercise is that unless there are multiple physical processors involved, busy waiting is a poor choice of synchronization primitive!
Show how monitors can be used to implement semaphores. Write a monitor with entry routines \texttt{UP} and \texttt{DOWN} that implements a general semaphore.

```pascal
generalSemaphore : monitor
var
  semaphoreFree : condition
  semaphore     : integer  := 0

monitor invariant
  (semaphore = 0)

synchronization condition
  (semaphore > 0)

procedure down()
begin
  if (semaphore = 0) then
    wait( semaphoreFree)
  end

  assert( semaphore > 0)
  semaphore := semaphore - 1
end

procedure up()
begin
  semaphore := semaphore + 1
  assert( semaphore > 0)
  signal( semaphoreFree)
end;
end generalSemaphore
```

Alternatively, one could formulate a more general monitor in which the actual semaphore is passed as a parameter to routines \texttt{P} and \texttt{V}.