Problem 1 (0 Points)

Consider the algorithm shown below and assume that we have created \( N \) processes numbered from 0 to \( N - 1 \) running on \( N \) processors. Furthermore, \( N \) is an odd integer.

Shared Variables:
- Semaphore \( X[N] = 0; \) // array of \( N \) semaphores initialized to 0
- Semaphore \( Y = 1; \)
- int \( a = 0; \)
- Process \( i: \) // 0 <= \( i <= N-1 \)
  - int \( b; \)
  - if ((\( i \) Mod 2) == 0) { // beginning of body
      ... Compute \( b \) ...
      Wait (\( Y \));
      \( a = a + b; \)
      Signal (\( Y \));
    } else {
      Wait(\( X[i-1] \));
      Wait(\( X[\text{Min}(i+2, N-1)] \));
      ... Compute \( b \) ...
      Wait (\( Y \));
      \( a = a + b; \)
      Signal (\( Y \));
    }
  Signal (\( X[i] \)); // end of body

a) Consider the case of \( N = 7 \) and where each process is running uninterrupted (except for synchronization) on its own processor. In what order will each process compute their local values of \( b \)? That is, give the sequence of process numbers. Explain.

b) What is the purpose of each of the semaphores \( X[i] \), and \( Y \)?

c) Suppose that we put the body of the above process code in a do forever ... loop. Show how you can insure that each process does not start loop iteration \( i + 1 \) until all processes have completed iteration \( i \).
Problem 2 (4 Points)

Barrier synchronization between \( N \) processes works as follows:

- A counter is initialized to \( N \), the number of processes participating in the barrier.
- The first \( N - 1 \) processes arriving to a barrier should wait.
- The \( N \)th process arriving to a barrier should unblock the \( N - 1 \) waiting processes so that all \( N \) processes can continue with the rest of the program.

The last page of this assignment defines the assembler instruction set of a machine which includes the \textit{atomic swap} instruction. The atomic swap instruction denoted by \( R1 \leftarrow \leftarrow R \) exchanges the contents of register \( R1 \) with the word at the memory location named \( R \) without allowing any intervening interrupts or deferred traps.

a) Suppose that your machine has enough processors so that each process can execute on its own CPU. Give a barrier synchronization algorithm that uses the \textit{atomic swap} hardware instruction and busy waiting. Assume that there are no synchronization functions except the \textit{atomic swap} instruction. Note that the algorithm needs to perform a barrier synchronization between \( N \) processes only once. Also, assume that there are no other synchronization primitives available.

b) Explain how your algorithm works.

Problem 3 (4 Points)

We consider using semaphores to implement the two functions \texttt{Accept(int x)} and \texttt{Post(int x)} whose semantics are defined as follows:

- **void Accept(int \*x)**: Upon return, \( x \) has the value by a process that has called \texttt{Post(y)}. Furthermore, if there are multiple processes calling \texttt{Accept} and \texttt{Post}, they are matched and serviced in FIFO order; i.e., the first \texttt{Accept} is paired with the first \texttt{Post}; the second \texttt{Accept} is paired with the second \texttt{Post}; etc.

- **void Post(int x)**: Upon return, the value of \( x \) has been transferred to the \textit{matching} \texttt{Accept} call.

\texttt{Accept} and \texttt{Post} form a \textit{rendezvous} between two processes in which the process calling \texttt{Post} passes an integer message to the process with matching \texttt{Accept}.

a) Give a simple implementation for \texttt{Accept} and \texttt{Post} that uses semaphores (not busy waiting).

b) Explain how your implementation in Part a satisfies the FIFO pairing description.
**Problem 4 (2 Points)**

The bank teller example with 1 teller given in class has the following features:

- There are $M = 1$ tellers and $N$ customers.
- The bank lobby has a capacity of $K = 20$ customers.
- One teller services customers one at a time from a common FIFO queue.
- The teller tells customers when he/she is ready before a customer can come to the teller window.
- Customers arriving to a full lobby return after a random delay.

Here is the algorithm given in class:

```
Global Variables:
Semaphore tRdy = 0, cRdy = 0, tDone = 0;
int n;            // # in lobby
Semaphore nLock = 1;    // protect n

Process customer (int i) {
    do forever {
        ... Random Delay ...
    } until [[ n < 20 &&
        n = n+1; ]]
        Signal(tRdy);
    Wait(tRdy);
    Signal(cRdy);
    ... Get service ...
    Wait(tDone);
    [[ n = n-1; ]]    // [[ ]] means Wait(nLock); ...
    ... Leave bank ...
}

Process teller (int i) {
    do forever {
        Signal(tRdy);
        Wait(cRdy);
        ... Serve customer ...
        Signal(tDone);
    }
}
```

The bank has decided to add more tellers so that there will be a total of $M$ tellers to serve the single line of customers.

a) Give an algorithm that simulates the general bank teller problem with an arbitrary number ($M$) of tellers and arbitrary number of ($N$) customers assuming that you have the two primitives `Accept` and `Post` at your disposal. Mark with a ’+’ in the left margin those lines that are different from the original algorithm.

b) Explain why your algorithm correctly simulates the synchronization required in the bank teller problem.
Problem 5 (4 Points)
We consider the use of semaphores for the entry and exit sections of a critical section that have the following properties:

- **EnterCondCS** (enter critical section)
  - This is executed before entering the critical section.
  - At most one process can enter the critical section.
  - A process can not enter the critical section unless the value of a boolean expression B is TRUE.
  - The value of the boolean expression B can only change inside the critical section; i.e., the critical section can change the value of B.

- **ExitCondCS** (exit critical section)
  - This is executed after leaving the critical section.
  - All blocked processes should retest B and the first one finding B to be TRUE should be allowed to enter the critical section.

a) Give semaphore implementations of **EnterCondCS** and **ExitCondCS**.

b) Explain the purpose of each semaphore and shared variable.

Problem 6 (4 Points) [From Tanenbaum]

A system has four processes and five allocatable resources. The current allocation and maximum needs are as follows:

<table>
<thead>
<tr>
<th>Process</th>
<th>Allocated</th>
<th>Maximum</th>
<th>Available</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1 0 2 1 1</td>
<td>1 1 4 2 2</td>
<td>x 0 y 1 1</td>
</tr>
<tr>
<td>B</td>
<td>2 0 1 1 0</td>
<td>2 2 3 1 0</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>1 1 0 1 0</td>
<td>2 1 4 1 0</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>1 1 1 1 0</td>
<td>1 1 3 2 1</td>
<td></td>
</tr>
</tbody>
</table>

What are the smallest values of $x$ and $y$ for which this is a safe state? Explain.
A Simple Assembler Language (ASAL)

The following specifies the syntax and informal semantics of the assembler language for a RISC-like machine. BNF (Backus-Naur Form) is used to describe the syntax where the metasymbols are ::= (is defined as), <> (class), and | (or). All other symbols are terminal symbols. Note that everything to the right of the # symbol should be treated as a comment.

<program> ::= <statement> ; | # a single statement
<statement> ::= <register> <-- <variable> | # load
<register> --> <integer> | # load immediate
<register> --> <variable> | # store
<variable> --> <register> | # atomic swap
goto <label> | # branch
<conditional> | # conditional
<operation> | # arithmetic operation

<register> ::= R0 | R1 | R2 | R3
<variable> ::= C-variable name # name of memory location
<label> ::= L0 | L1 | ... | L9
<integer> ::= an integer
<conditional> ::= ifeq <register> goto <label> | # if reg equal 0
ifne <register> goto <label> | # if reg not equal 0
ifgt <register> goto <label> | # if reg > 0
iflt <register> goto <label> | # if reg < 0

In this language, a variable refers to the contents of a memory location, and a register name refers to the contents of a register. In a multiprocessor, each CPU has its own set of registers. For example, if there are four processors, each processor has R0, R1, R2 and R3 registers. The following is a program that continuously examines a memory location until it contains a 0 and then loads the value 1 into x:

...  
R1 <-- 1; # load the constant 1
L0:  R0 <-- x;  # load R0 with contents of x
     ifne0 R0 goto L0;  # branch to L0 if R0 not equal to 0
     R1 --> x;  # put a 1 into x