Problem 1 (2 Points)
Consider the following parallel program:

```c
int X = 0; // Global (shared)
int want[10] = {0, ... , 0}; // Global (shared)
void sum() {
    for (int n=0; n < 10; n++)
        if (!want[n]) { X += n; want[n] = 1; }
}
void main() { X=0; parbegin sum(); sum(); parend; print(X); }
```

The construct `parbegin S1; S2; ... parend` with statements S1, S2, ... means that the statements can execute in parallel subject to any synchronization.

Determine the smallest value of the shared variable X that will be printed and explain how you arrived at this answer. Assume: 1) processes can execute at any relative speed; 2) a value can be incremented/decremented after it has been loaded into a register; and 3) the values of X and `want[n]` are loaded only once for each execution of the loop.

Problem 2 (0 Points)
This problem considers Peterson's 2-process algorithm given in class.

a) Explain how the algorithm prevents one process from monopolizing the critical section; i.e., prevent starvation?

b) Explain how the algorithm guarantees freedom from deadlock?

Problem 3 (6 Points)
The following is a software-only mutual exclusion algorithm for two processes (id = 0 and 1):

```c
1   boolean blocked[2] = { 0, 0 };
2   int who = 0;
3   void P (int id) {
4       while (TRUE) {
5           blocked[id] = TRUE;
6           while (who != id) { while (blocked[1-id]) who = id; }
7           // ... some critical section is here ...
8           blocked[id] = FALSE;           // exit section
9       }
10   }
11   void main () { parbegin P(0); P(1); parend; }
```
a) If the processes execute at the same rate as much as possible, in what order will the processes enter the critical section.

b) Summarize how the algorithm attempts to guarantee mutual exclusion in the general case.

c) The algorithm contains some flaws. Give an example where the algorithm exhibits starvation.

d) Give an example where the algorithm exhibits livelock and explain why your example exhibits livelock.

e) In what sense is the algorithm speed-sensitive?

Problem 4 (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.

\[
\text{Shared Variables:}
\]

<table>
<thead>
<tr>
<th>Semaphore</th>
<th>( X[N] = { 0, \ldots, 0 } );</th>
</tr>
</thead>
<tbody>
<tr>
<td>Semaphore</td>
<td>( Y = 1 );</td>
</tr>
<tr>
<td>int</td>
<td>( a = 0 );</td>
</tr>
<tr>
<td>Process ( i: )</td>
<td>// ( 0 \leq i \leq N-1 )</td>
</tr>
<tr>
<td>int ( b; )</td>
<td>( )</td>
</tr>
<tr>
<td>if ( ((i \text{ Mod } 2) == 0) )</td>
<td>// beginning of body</td>
</tr>
<tr>
<td>( )</td>
<td>( )</td>
</tr>
<tr>
<td>Compute ( b );</td>
<td>( )</td>
</tr>
<tr>
<td>Wait ( (Y) );</td>
<td>( )</td>
</tr>
<tr>
<td>( a = a + b; )</td>
<td>( )</td>
</tr>
<tr>
<td>Signal ( (Y) );</td>
<td>( )</td>
</tr>
<tr>
<td>} else</td>
<td>( )</td>
</tr>
<tr>
<td>Wait( (X[i-1]) );</td>
<td>( )</td>
</tr>
<tr>
<td>Wait( (X[\text{Min}(i+2,N-1)]) );</td>
<td>( )</td>
</tr>
<tr>
<td>... Compute ( b );</td>
<td>( )</td>
</tr>
<tr>
<td>Wait ( (Y) );</td>
<td>( )</td>
</tr>
<tr>
<td>( a = a + b; )</td>
<td>( )</td>
</tr>
<tr>
<td>Signal ( (Y) );</td>
<td>( )</td>
</tr>
<tr>
<td>}</td>
<td>( )</td>
</tr>
<tr>
<td>Signal ( (X[i]) );</td>
<td>// end of body</td>
</tr>
</tbody>
</table>

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 \texttt{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 5 (6 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 *atomic swap* instruction. The atomic swap instruction denoted by \( R1 \leftrightarrow X \) exchanges the contents of register \( R1 \) with the word at the memory location named \( X \) without allowing any intervening interrupts or deferred traps.

In all cases, assume that there are no other synchronization primitives available, and a software mutual exclusion algorithm is disallowed.

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 *atomic swap* hardware instruction and busy waiting. Assume that there are no synchronization functions except the *atomic swap* instruction. Note that the algorithm needs to perform a barrier synchronization between \( N \) processes only once.

b) Explain how your algorithm works.

c) Give a solution in which there would be no more than 10 calls to the *atomic swap* instruction for the case of \( N=3 \) processes. Explain how your solution meets this criteria. **HINT:** Use an auxiliary variable to indicate that the value of the *atomic swap* counter is being updated.

d) **Extra Credit – 2 Points:** A solution in which the \( N \) processes can safely execute a sequence of barrier synchronizations must address difficult issues. Discuss why this is a difficult problem.

e) **Extra Credit – 2 Points:** Give a solution in which the \( N \) processes can safely execute a sequence of barrier synchronizations. You can use a high-level language to describe your solution.
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  
<statement> ::= <register> <-- <integer> | # load immediate  
<statement> ::= <register> --> <variable> | # store  
<statement> ::= <register> <-- <variable> | # atomic swap  
goto <label> | # branch  
<conditional> | # conditional  
<operation> | # arithmetic operation  
<operation> ::= <uop> <register>  
<uop> ::= incr | decr  # unary operators  
<bop> ::= + | - | * | /  # binary operators  
<register> ::= R0 | R1 | R2 | R3  
<label> ::= C-variable name  # name of memory location  
<integer> ::= L0 | L1 | ... | L9  
<integer> ::= an integer  
<conditional> ::= ifeq0 <register> goto <label> | # if reg equal 0  
<conditional> ::= ifne0 <register> goto <label> | # if reg not equal 0  
<conditional> ::= ifge0 <register> goto <label> | # if reg >= 0  
<conditional> ::= ifle0 <register> goto <label> | # if reg <= 0  
<conditional> ::= ifgt0 <register> goto <label> | # if reg > 0  
<conditional> ::= iflt0 <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

4