Section A: Questions (10 points each)

1. The operating system manages processes with a Process Control Block (PCB). List three examples of information contained in the PCB that must be updated when the operating system makes a process switch. Give a one- or two-sentence description of how this item is changed.

   Process State: The operating system updates the state of the process according to the state transition diagram. So if the process is currently running, but then has to wait for I/O, the state in the PCB gets set to Blocked.

   Control and Status Registers: This is the context of the process and must be saved for a process being taken off the processor, or restored for a process that is now going to run.

   Data Structuring: The process is going to be put into a different queue (i.e. Ready or Blocked). These fields are what the operating system uses to link the PCB into a queue, so they need to be modified.

2. What is the difference between a thread and a process? On a system that implements only user-level threads, why is it not possible for a multi-threaded application to take advantage of multiple processors?

   A thread is a unit of execution, whereas a process is a unit of resource ownership. A thread has its own execution state, context, and stack, whereas a process maintains control of resources such as files and I/O devices. A process may have one or more threads.

   On a system that implements only user-level threads, the operating system only supports scheduling of processes. With multiple processors, the OS will be able to run several processes at the same time, but not several threads from the same process.

3. How does a hardware test-and-set instruction guarantee mutual exclusion? Explain one of its disadvantages.

   The test-and-set instruction guarantees mutual exclusion by providing an atomic instruction to set a lock on a variable. The instruction tests the value of the variable. If the value is 0, the instruction sets it to 1 and returns true. Otherwise, the value stays the same and the instruction returns false. Because the instruction cannot be interrupted, only one process can receive true and enter the critical section.

   The disadvantages of test-and-set include: (a) busy waiting is used, which wastes processor cycles and (b) starvation is possible because processes waiting for the variable to clear are not ordered.

Section B: Problems (10 points each)

1. Give an example of how two processes, using a shared variable $x$, could be interleaved to produce an error if they are NOT using synchronization. Show the content of the program executed by the processes, the result they should get, the ordering of process instructions, and the result actually received.
Suppose two processes are executing the following line from a program:

\[ x := x + 1; \]

If \( x \) contains the value 0 initially, then when the two processes both execute this piece of code, \( x \) should contain the value 2. However, without any synchronization, it could be that process A loads \( x \) into a register, then process B does the same, so that they both have 0 in their register. Next A increments and stores, writing 1 into the variable \( x \). Then B increments and stores, also writing 1 into the variable \( x \). So the actual result is 1.

2. The following is the current state for a system. Run the banker’s algorithm to determine whether the system is in a safe or unsafe state. Show all your work and give the sequence of any processes that are able to run.

<table>
<thead>
<tr>
<th>Available</th>
<th>R1</th>
<th>R2</th>
<th>R3</th>
<th>R4</th>
<th>R5</th>
<th>R6</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

Calculate the Need =

<table>
<thead>
<tr>
<th>P1</th>
<th>1, 1, 0, 0, 1, 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>P2</td>
<td>0, 0, 0, 0, 0, 1</td>
</tr>
<tr>
<td>P3</td>
<td>0, 1, 0, 2, 0, 0</td>
</tr>
<tr>
<td>P4</td>
<td>0, 0, 0, 1, 1, 0</td>
</tr>
<tr>
<td>P5</td>
<td>2, 3, 4, 0, 1, 0</td>
</tr>
<tr>
<td>P6</td>
<td>1, 0, 0, 2, 1, 0</td>
</tr>
</tbody>
</table>

Available = (1, 0, 0, 1, 1, 0).

Run P4. Available = (1, 1, 1, 1, 1, 0).

Run P1. Available = (3, 2, 5, 1, 1, 1).

Run P2. Available = (4, 2, 5, 1, 1, 1).

We are in an unsafe state! P4, P1, and P2 can run, but then we are deadlocked because P3, P5, and P6 cannot run.

Section C: Problems (20 points each)

1. This is a modification of a classic synchronization problem known as the bakery problem. A bakery has room for 20 customers. As customers enter, they take a number, then wait to be served. A baker serves 100 customers a day, in order from 1 to 100.

When a customer enters the shop, he takes a number and then waits for the baker to call his number. When it is the customer’s turn, he selects an item and pays the baker, waits for the baker to give him the item and his change, then leaves.

The baker sleeps when there are no customers. While there are customers in the shop, he chooses the next number (in order from 1 to 100) and tells that customer it is his turn. When he is finished serving that customer he serves the next customer in order or (if there are no more customers in the shop) goes to sleep.
Write code to synchronize the customers and the baker, using counting semaphores. You should include (1) variable declarations and initializations, (2) code for the customer process, and (3) code for the baker process.

```pascal
program bakery;

var shop: semaphore(:=20); (* control the capacity of shop *)
   number[100]: semaphore(:=0); (* the number a customer waits on, chosen when entering the shop *)
   customer_ready: semaphore(:=0); (* wake baker up if sleeping *)
   item: semaphore(:=0); (* select the item *)
   amount: semaphore(:=0); (* wait for total of purchase *)
   payment_ready: semaphore(:=0); (* tell barber payment is ready *)
   item_and_change: semaphore(:=0); (* tell baker customer is done *)
   mutex: semaphore(:=1); (* protect access to the numbers as they are chosen *)
   count: integer; (* global customer number *)

procedure customer;
var mynumber: integer; (* the number I choose *)

begin
   wait(shop);
   enter shop;
   wait(mutex);
   mynumber := count; (* choose a number *)
   count := count + 1;
   signal(mutex);
   signal(customer_ready); (* wake up barber *)
   wait(number[mynumber]); (* wait to be called *)
   select_item();
   signal(item); (* tell barber the desired item *)
   wait(amount); (* wait to be told the amount *)
   make_purchase();
   signal(payment_ready); (* tell the payment is ready *)
   wait(item_and_change);
   exit_shop();
   signal(shop);
end;

procedure baker;
var serving: integer;

begin
   for (serving := 1 to 100) begin
      wait(customer_ready);
      signal(number[serving]);
      wait(item);
      figure_amount();
      signal(amount);
      wait(payment_ready);
      figure_change();
      signal(item_and_change);
   end;
end;

begin (* main program *)
   count := 1;
   parbegin
      customer;...100 times;...customer;
      baker;
   parend
end.
```