Problem 1 (6 Points) [From Stallings]

Jurassic Park consists of a dinosaur museum, and a park for safari riding. There are \( N \) single-passenger cars and \( M \) visitors. Each visitor wanders around the museum for a while, and then lines up to take a ride in a safari car. When a car is available, it enters the loading dock where cars are lined up for loading in FIFO order. When the car reaches the front of the loading line, it is available for loading the next passenger in line. It waits for the visitor to signal he/she is ready to start the ride; and then travels around the park for a random amount of time before returning to the museum. If the \( N \) cars are all being used, a visitor who wants to ride must wait; if a car is ready to load but there are no waiting visitors, then the car must wait. After the ride in the park, the car signals the visitor when it is safe to exit the car, and the visitor leaves the park. The algorithm skeleton below simulates the above scenario.

```c
int nFullCars = 0; // number of full cars
Process Visitor (i) {
    ... walk around museum ...
    ... ride around park ...
}
Process Car (j) {
    do forever {
        ... ride around park ...
    }
}
```

a) Complete the above algorithm using semaphores to synchronize the \( M \) passenger processes and the \( N \) car processes in the manner described above. Do not use busy waiting.

b) Indicate the purpose of each semaphore and shared variable by giving a single phrase or sentence that captures the essence of the purpose (e.g., "allow atomic update of \( n \)").

Problem 2 (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>Process 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>Process B</td>
<td>2 0 1 1 0</td>
<td>2 2 3 1 0</td>
<td></td>
</tr>
<tr>
<td>Process C</td>
<td>1 1 0 1 0</td>
<td>2 1 6 1 0</td>
<td></td>
</tr>
<tr>
<td>Process 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.

Problem 3 (0 Points)

Tanenbaum, Problem 37 (Chapter 2).
Problem 4 (4 Points)

We wish to derive the equations of motion for a workload that consists of \( N + 1 \) jobs consisting of \( N \) type A jobs and one type B job being serviced by the simple scheduling scheme described below. Job type A has a CPU demand of \( a \) seconds, and job type B has a demand of \( b \) seconds. Furthermore, \( b \) is a large integer multiple of \( a \), and type A jobs arrive at fixed time points \( X, 2X, 3X, \) etc. where \( X \) is a positive integer multiple of \( a; \) i.e., \( X = ka \) where \( k \) is a positive integer.

In this system, type A jobs have a higher priority than the one type B job and will preempt (with 0 overhead) any type B job from the CPU.

a) Draw the space-time diagram (time runs to the right) for the case when \( a = 1, b = 10, k = 2, \) and \( N = 4 \).

b) Derive an expression for the turnaround time and the queueing time of the type B job and each type A job when \( b > a(N(k-1) + 1) \).

c) Extra Credit (2 Points): Derive an expression for the turnaround time and the queueing time of the type B job and each type A job for the general case of arbitrary \( N \) and \( k \) although subject to the constraints specified earlier; i.e., \( k \) is a positive integer, and \( b \) is a large integer multiple of \( a \).

Problem 5 (6 Points)

The goal of this problem is to get acquainted with the ucontext calls makecontext, getcontext, and swapcontext by implementing an example of co-routine style flow control where control is passed from thread \( i \) to thread \( (i+1 \mod 4) \) several times.

Modify the ucontext-basic.c program so that it creates 4 instances of the childFiber thread that executes the algorithm shown below. The details are the following:

- The id of \( k \)th childFiber is \( k \).
- The algorithm for each childFiber is:

  Display "BEGIN" followed by my id;
  Do 2 times {
    Sleep for 1 second;
    Yield to my neighbor thread;
    Display "RESUME" followed by my id;
  }
  Sleep for 1 second;
  If I am not thread 3 Then
    Yield to my neighbor thread;
  Else
    Yield to the main thread;
  End
  Display "END" followed by my id;

In the above algorithm, thread \((k+1) \mod 4\) is the next neighbor of thread \( k \); i.e., the threads form a ring control structure. The id is the instance number; i.e., 0, 1, 2, or 3.
Submit the following:

a) Your source listing
b) The output of your program
c) A short explanation of why your output shows that your program is functioning properly.