1)

```
signal(semaphore[i]);
for(j = 0; j < N; j++) {
  wait(semaphore[j]);
  signal(semaphore[j]);
}
[[Rendezvous Code Here]]]
```

a. Prove that the above code will work. What should be the initialization of semaphore[i]?

The above code will work because it ensure that all processes arrive at the rendezvous point in the order they arrive to the first line of code. As a semaphore reaches the first line of code, it increments its value by one. Then, when the for loop executes, the process i is not blocked by executing wait(). As other processes execute the above code, the processes that have already made it through the first line still avoid being blocked by executing wait().

semaphore[i] should be initialized to 0. This way, only processes that have begun to execute the above piece of code can avoid being blocked.

When presented with the above code, another student suggested that a better solution is the one below.

```
signal(semaphore[i]);
for(j = 0; j < N; j++) {
  wait(semaphore[j]);
  signal(semaphore[j]);
}
[[Rendezvous Code Here]]]
wait(semaphore[i]);
```

b. Could a process ever block as a result of executing the last instruction in the above code? If yes, give an example, if not, then explain why not.

No, a process could never block because after reaching the rendezvous code section, each semaphore has a value of 1.

c. Why do you think the last instruction in the above code is included?

This was included because the above code increases the value of the semaphores by 1 when entering the rendezvous section. So this final instruction was included to return the semaphores to their original value.

2) a. Explain how the same washing machine can be assigned to two different customers.

Since there is nothing protecting the code within the for loop (which returns a free machine) it is possible that requests from the two stations could bypass the statement *if (available[i])* at the same exact time. For example, say machine 0 is free. If both stations request a new machine at exactly the same time, and both simultaneously check if machine 0 is free, it is possible that. before either station switches its value to false, both stations will see that it is available and return that machine number.

b. Modify the pseudo code above to eliminate the problem.

We can easily fix this problem by adding another semaphore that ensures only one station searches for available machines at a time.

Let this semaphore be called mutex, and be initialized to 1.

```
int allocate() /* Returns index of available machine */
{

  wait(mutex);
  wait(nfree); /* Wait until a machine is available */
  for (int i=0; i < M; i++)
    if (available[i]) {
      available[i] = FALSE;
      return i;
    }
  signal(mutex);
}
```

3) Explain how you could use the above template to design a protocol for a set of processes to access the CS according to some arbitrary metric -- e.g., using EDF based on a deadline imposed on the process, or by accounting for the number of times a process was allowed to enter the CS, and giving priority to the process that has used the CS the least so far in its lifetime, etc.

This template design could easily used with any scheduling protocol implemented. When the thread responsible for scheduling runs, we can simply replace the process selection method with any other method. In this example, the next process to be scheduled is the one with the lowest id number that is requesting access. Implementing a different method of scheduling is simply a matter of keeping track of the variables you need (e.g: the current deadline for each thread, or number of times a thread accesses the CS) and comparing the values for each of the thread (e.g. EDF would pick the process with the closest deadline, or you may choose the one that access the CS most/least).