1. Why do condition variables with priorities (using a conditional-wait operation) only work properly with Hoare-style monitors?

2. Consider the following resource allocation scenario. There exists a pool of \( n \) identical units of a resource type. Users request any number \((\leq n)\) of those units. You may assume that user processes are sufficiently civilized not to make a further request before its most recent allocation request has been satisfied and those units have been returned to the system. However, a user may choose to return its holdings piecemeal. You may not assume a limit on the number of processes contending for access to the resource pool.

Design and code a monitor which manages the above resource pool. The monitor should provide two externally visible operations:

- \textbf{Request(num)} is invoked by a user to acquire \( \text{num} \) resource units. You need not inform the user of which units are granted to him/her.
- \textbf{Release(num)} is invoked by a user to return \( \text{num} \) units of his/her current allocation to the system. You may assume that a perverse user never attempts to Release what has not been previously Request-ed and allocated.

Your solution should ensure that a user requesting a small number of resources does not wait unnecessarily behind a user requesting a larger number when only the smaller number is currently available.

3. Give a sequence of events that shows that the Dining Philosophers monitor that we discussed in class (and in your textbook) is not starvation-free.
4. A particular river crossing is shared by both cannibals and missionaries. A single boat is used to cross the river, but it only seats three people, and must always carry a full load. In order to guarantee the safety of the missionaries, you cannot put one missionary and two cannibals in the same boat (because the cannibals would gang up and eat the missionaries), but all other combinations are acceptable.

Each cannibal is a thread that executes the following code:

```java
void Cannibal()
{
    while (true)
    {
        M.CannibalArrives(); // register to cross the river

        // come back for another river crossing
    }
}
```

Each missionary executes similar code:

```java
void Missionary()
{
    while (true)
    {
        M.MissionaryArrives(); // register to cross the river

        // come back for another river crossing
    }
}
```

Both cannibal and missionary threads will run in an infinite loop. When a cannibal arrives at the river bank, it calls the monitor function `CannibalArrives()` with some arguments. A missionary thread, on the other hand, calls the monitor function `MissionaryArrives()`. The caller will be blocked in the monitor until the monitor can find a safe boat load containing the caller. When a safe boat load is possible, your monitor releases three threads and rows the boat. After this, each of these three threads returns to their respective thread function, does something and magically floats back across the river for another crossing.

It is very important that no one be able to jump off the boat while it is crossing, and that no one be able to jump on a boat before getting permission to do so. Also, make sure deadlock is not possible.