National University of Computer and Emerging Sciences

Operating Systems Lab

Lab Manual 11

Muhammad Hassan Raza
Fall 2024

Department of Software Engineering
FAST-NU, Lahore, Pakistan

1. Circular Printing with Deadlock Prevention

You are tasked with designing a program where three threads work together to print a repeating sequence of characters: "ABCABCABC...".
Each thread is responsible for printing one specific character:
- Thread 1 prints 'A',
- Thread 2 prints 'B',
- Thread 3 prints 'C'.

The threads must adhere to the following synchronization rules:
1. Each thread can only print its character when it holds a mutex specifically associated with its character.
   - Thread 1 requires MutexA.
   - Thread 2 requires MutexB.
   - Thread 3 requires MutexC.
2. Before printing, a thread must also acquire the mutex of the *next character in the sequence*:
   - Thread 1 (printing 'A') must acquire MutexB after MutexA.
   - Thread 2 (printing 'B') must acquire MutexC after MutexB.
   - Thread 3 (printing 'C') must acquire MutexA after MutexC.
3. After printing, the thread releases its mutex and the mutex of the next character in the sequence.

Your program must:
- Print exactly 30 characters in the sequence "ABCABCABC...".
- Prevent deadlocks, ensuring that the threads do not block indefinitely due to mutex dependencies.
- Use only mutexes for synchronization. Do not use condition variables or semaphores.

Example Output:

```
ABCABCABCABCABCABCABCABCABCABC
```

Hints:
- Carefully design how threads acquire and release the required mutexes to prevent cyclic dependencies.
- Think about using one mutex to signify the "permission" to start the sequence and coordinate thread interactions.

Q2. Variable-Rate Resource Allocation

You are managing a system with 5 shared resources, such as printers, that multiple threads (representing processes) need to use.
Each thread requests between 1 and 3 resources at random. If sufficient resources are not available, the thread must wait until its request can be fulfilled.

The system must adhere to the following rules:
1. The system starts with all 5 resources available. Each thread that starts execution will randomly request 1, 2, or 3 resources.
2. If the requested number of resources is available, the thread acquires them, performs its "task" (e.g., printing), and then releases the resources after 1 second.
3. If the requested number of resources is not available, the thread must wait until enough resources are released by other threads.
4. The system must prevent deadlocks, ensuring no thread waits indefinitely for resources.
5. The allocation must be fair, avoiding starvation. Threads that have been waiting longer should have priority over newly spawned threads.

Your program must:
- Use semaphores to manage the available resources.
- Simulate 10 threads that execute concurrently.
- Print the following information for each thread:
  - Thread ID.
  - Resources requested.
  - Resources acquired.
  - Resources released.

Example Output:

```
Thread 1 requested 3 resources.
Thread 1 acquired 3 resources.
Thread 1 released 3 resources.
Thread 2 requested 2 resources.
Thread 2 acquired 2 resources.
Thread 2 released 2 resources.
```

Hints:
- Use a counting semaphore to track the available resources.
- Combine the semaphore with a queue or priority mechanism to ensure fairness.
- Ensure the system handles overlapping requests for resources efficiently without overcomplicating the logic.