19CSE313 Principles of Programming Languages



Elevator System – Sleeping Barbour Problem

|  |  |
| --- | --- |
| Team Members | Roll No. |
| Athithya S | CB.EN.U4CSE22405 |
| Vishaarad S | CB.EN.U4CSE22451 |
| Yuvaraj Ayyanar A | CB.EN.U4CSE22455 |

**Elevator System**

**Introduction:**

* The Sleeping Barber Problem is a classical synchronization problem in computer science, representing resource allocation in concurrent systems. It models a scenario where a barber (or a limited number of barbers) serves customers who arrive at unpredictable times. If no customers are present, the barber sleeps until a new customer arrives. If all chairs are occupied, incoming customers must leave. This problem is typically used to illustrate process synchronization using semaphores, mutexes, and condition variables.
* In an elevator system, a similar synchronization challenge occurs. The elevator acts as the barber, while passengers are analogous to customers. The elevator operates based on demand, sleeping (idle) when no passengers request a ride and waking up when a request is received. If the elevator is full, additional passengers must wait for the next available space.

**Working States:**

1. **Idle State:** If no passengers request the elevator, it remains idle (similar to a sleeping barber waiting for customers).
2. **Passenger Requests:** When a passenger presses a button, the elevator wakes up and processes the request.
3. **Efficient Scheduling:** The elevator serves requests efficiently, deciding the optimal order in which to serve floors, much like a barber managing waiting customers.
4. **Concurrency Control:** In a multi-elevator system, synchronization ensures that requests are distributed efficiently among available elevators, preventing race conditions.

**Applications :**

1. **Energy Efficiency:** Energy efficiency is a crucial factor in designing an elevator system. It ensures that the elevator operates **optimally**, minimizing power consumption without compromising service quality.
2. **Queue Management:** Queue management is essential to **efficiently handle multiple passenger requests** while avoiding starvation or delays.
3. **Emergency Handling:** Handling emergency situations is critical in ensuring **passenger safety**.

**Challenges:**

**1. Avoiding Deadlock for Passengers (Inside & Waiting)**

* Passengers inside the elevator and those waiting should not cause a deadlock where the elevator stops responding.

**Concurrency Handling:**

1. Multiple passengers requesting floors at the same time

* Each passenger runs in a separate thread and requests a floor independently.

2. Elevator processing requests while new requests are being added

* The elevator moves while passengers are still requesting floors.

**Solution in Code:**

* Synchronization ensures that multiple threads (passengers) do not create conflicts while requesting floors.
* The wait() and notify() mechanisms allow the elevator to pause when idle and resume when a new request arrives.
* The notifyAll() in processFloor() ensures all waiting threads (passengers) are notified when the elevator reaches a floor.

while (!emergencyTriggered && upQueue.isEmpty() && downQueue.isEmpty()) {

try {

moving = false;

System.out.println("No requests. Elevator is sleeping.");

wait(); // Elevator waits for new requests

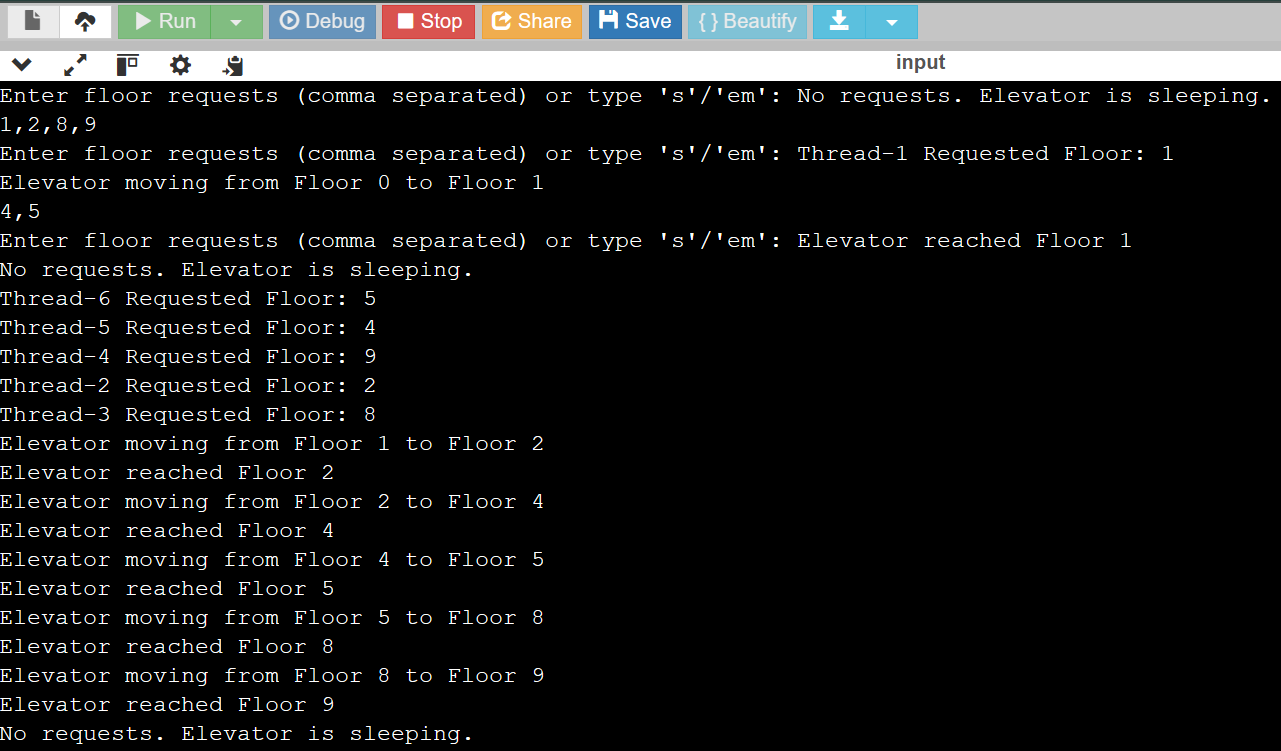
} catch (InterruptedException e) {

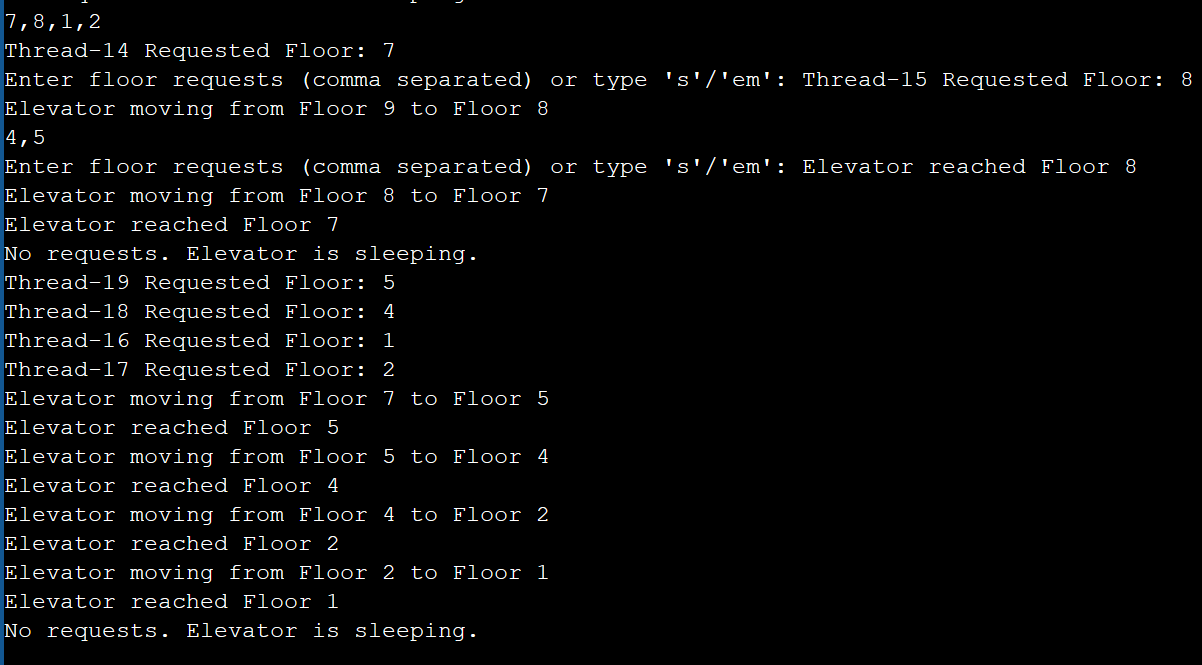
Thread.currentThread().interrupt();

}

}

* If no requests exist, the elevator **sleeps** to save processing power.
* When a new request comes (request(int floor)), notify() wakes up the elevator and continues its operations.





**2.Emergency Situations Should Override Normal Operations**

* During an emergency, the elevator should **immediately stop** all operations and clear pending requests. Once resolved, normal operations should resume.

**Concurrency Handling:**

1. Emergency Handling Overrides Normal Processing

* If an emergency occurs, it interrupts the normal operations and stops immediately.

**Solution in Code:**

* The emergencyStop() method clears all floor requests, ensuring the elevator does not move to any requested floors.
* The handleEmergency() method ensures the elevator pauses, waits, and resumes after the emergency is resolved.

public synchronized void emergencyStop() {

emergencyTriggered = true;

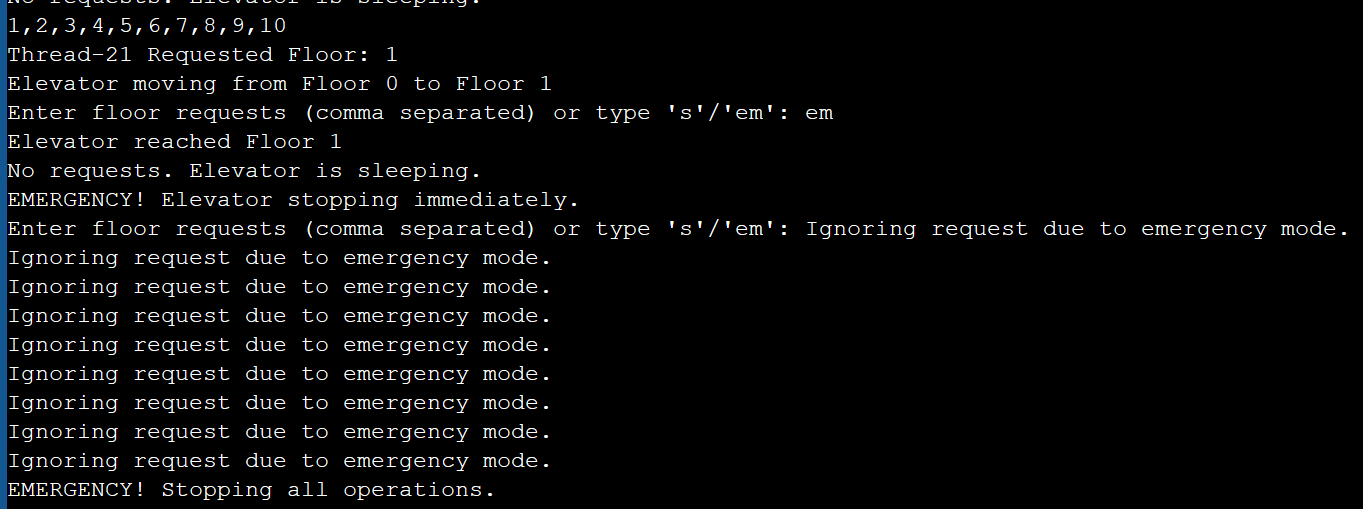
upQueue.clear(); // Clear all pending requests

downQueue.clear();

System.out.println("EMERGENCY! Elevator stopping immediately.");

notify(); // Wake up the elevator if it's waiting

}



* The emergencyTriggered flag prevents **any new request** from being processed.
* upQueue.clear(); downQueue.clear(); ensures **all pending requests are ignored** during an emergency.
* The notify() ensures the elevator wakes up from waiting mode and stops.

private synchronized void handleEmergency() {

System.out.println("EMERGENCY! Stopping all operations.");

try {

Thread.sleep(5000); // Simulates emergency handling delay

} catch (InterruptedException e) {

Thread.currentThread().interrupt();

}

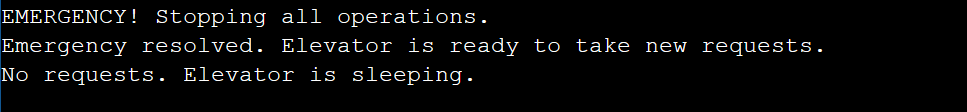
emergencyTriggered = false;

System.out.println("Emergency resolved. Elevator is ready to take new requests.");

notifyAll(); // Allow passengers to request floors again

}

* The elevator **waits for 5 seconds**, simulating emergency handling.
* After resolution, notifyAll() wakes up all threads (passengers) waiting to send requests.



**3.The Current Shut down leads to stop all request of the floor**

if (input.equalsIgnoreCase("s")) {

System.out.println("Elevator System is sleeping");

elevator.stopElevator();

break;

}

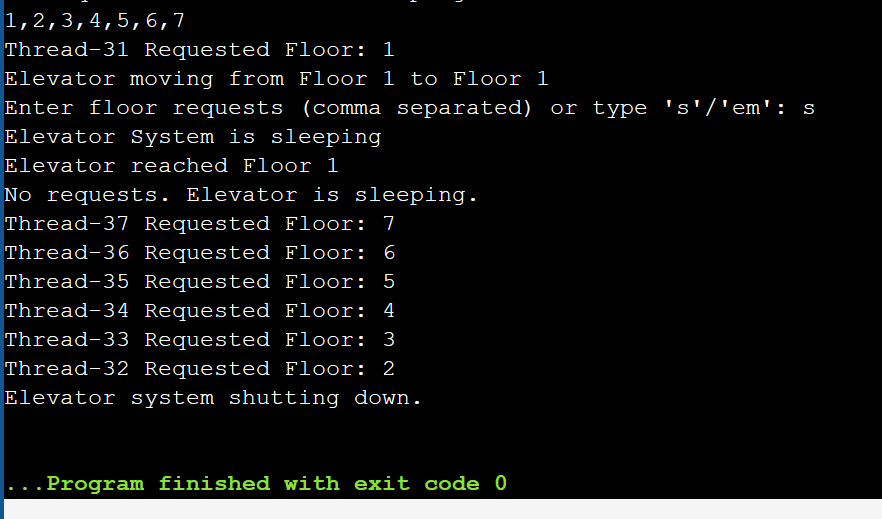
else if (input.equalsIgnoreCase("em")) {

elevator.emergencyStop();

continue;

}

If the enter value is s – means Stop, the elevator goes to sleep mode then shutting down stage



**Summary of Challenges:**

1. **Deadlock Prevention:** Handled using synchronized, wait(), notify(), and priority queues.
2. **Emergency Handling:** Handled by emergencyStop(), clearing requests, and stopping operations.
3. **Concurrency:** Achieved using multiple passenger threads, an independent elevator thread, and synchronized methods.

**CODE :**

import java.util.\*;

class Elevator {

private int currentFloor = 0;

private boolean moving = false;

private boolean emergencyTriggered = false;

private boolean running = true;

private final PriorityQueue<Integer> upQueue = new PriorityQueue<>();

private final PriorityQueue<Integer> downQueue = new PriorityQueue<>(Collections.reverseOrder());

public synchronized void request(int floor) {

if (emergencyTriggered) {

System.out.println("Ignoring request due to emergency mode.");

return;

}

if (floor > currentFloor) {

upQueue.offer(floor);

} else {

downQueue.offer(floor);

}

System.out.println(Thread.currentThread().getName() + " Requested Floor: " + floor);

if (!moving) {

moving = true;

notify();

}

}

private synchronized void move() {

while (running) {

while (!emergencyTriggered && upQueue.isEmpty() && downQueue.isEmpty() && running) {

try {

moving = false;

System.out.println("No requests. Elevator is sleeping.");

wait();

} catch (InterruptedException e) {

Thread.currentThread().interrupt();

}

}

if (!running) {

System.out.println("Elevator system shutting down.");

break;

}

if (emergencyTriggered) {

handleEmergency();

continue;

}

Integer nextFloor = getNextFloor();

if (nextFloor != null) {

processFloor(nextFloor);

}

}

}

private Integer getNextFloor() {

if (!upQueue.isEmpty()) {

return upQueue.poll();

} else if (!downQueue.isEmpty()) {

return downQueue.poll();

}

return null;

}

private void processFloor(int floor) {

System.out.println("Elevator moving from Floor " + currentFloor + " to Floor " + floor);

try {

Thread.sleep(3000);

} catch (InterruptedException e) {

Thread.currentThread().interrupt();

System.out.println("Elevator interrupted!");

}

currentFloor = floor;

System.out.println("Elevator reached Floor " + currentFloor);

synchronized (this) {

notifyAll();

}

}

public synchronized void emergencyStop() {

emergencyTriggered = true;

upQueue.clear();

downQueue.clear();

System.out.println("EMERGENCY! Elevator stopping immediately.");

notify();

}

private synchronized void handleEmergency() {

System.out.println("EMERGENCY! Stopping all operations.");

try {

Thread.sleep(5000);

} catch (InterruptedException e) {

Thread.currentThread().interrupt();

}

emergencyTriggered = false;

System.out.println("Emergency resolved. Elevator is ready to take new requests.");

notifyAll();}

public void startElevator() {

new Thread(this::move).start();

}

public synchronized void stopElevator() {

running = false;

notifyAll();

}

}

class Passenger extends Thread {

private final Elevator elevator;

private final int floor;

public Passenger(Elevator elevator, int floor) {

this.elevator = elevator;

this.floor = floor;

}

public void run() {

synchronized (elevator) {

elevator.request(floor);

}

}

}

public class ESD {

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in);

Elevator elevator = new Elevator();

elevator.startElevator();

while (true) {

System.out.print("Enter floor requests (comma separated) or type 's'/'em': ");

String input = scanner.nextLine();

if (input.equalsIgnoreCase("s")) {

System.out.println("Elevator System is sleeping");

elevator.stopElevator();

break;

} else if (input.equalsIgnoreCase("em")) {

elevator.emergencyStop();

continue;

}

String[] tokens = input.split(",");

for (String token : tokens) {

int floor = Integer.parseInt(token.trim());

new Passenger(elevator, floor).start();

}

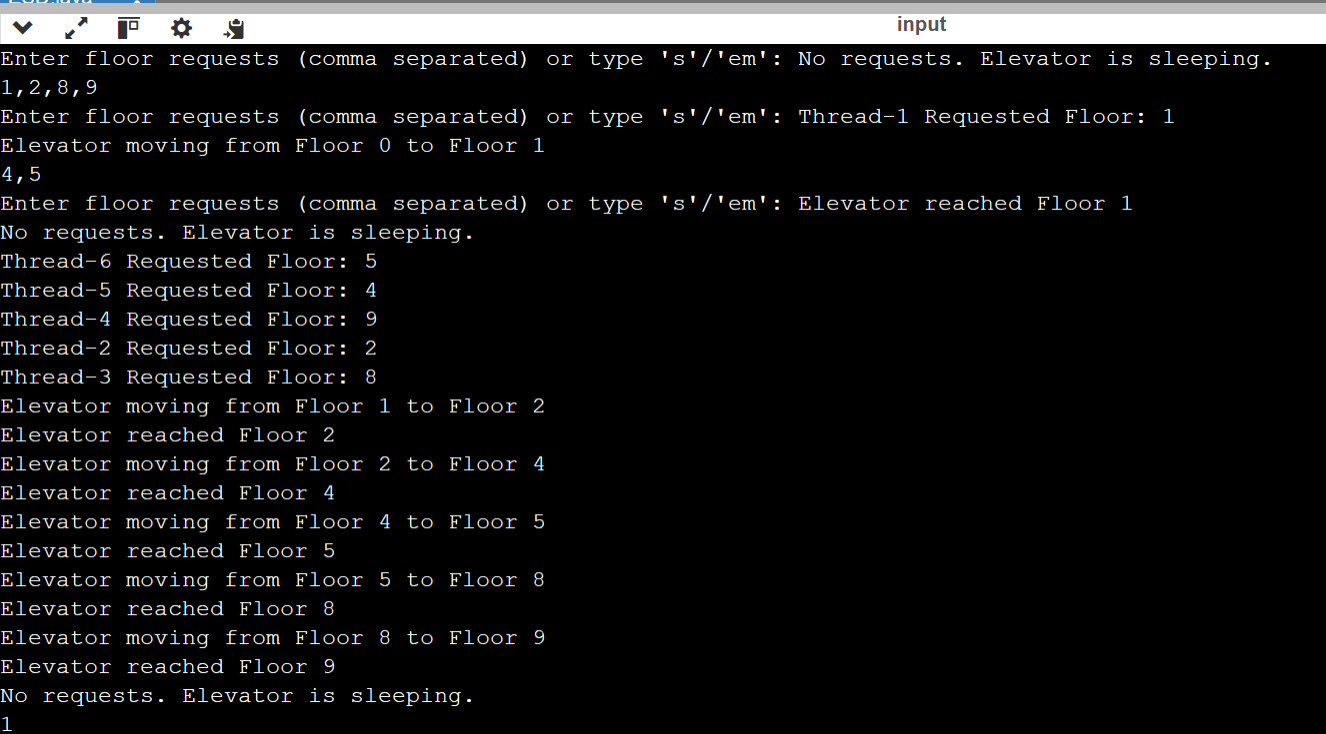
}

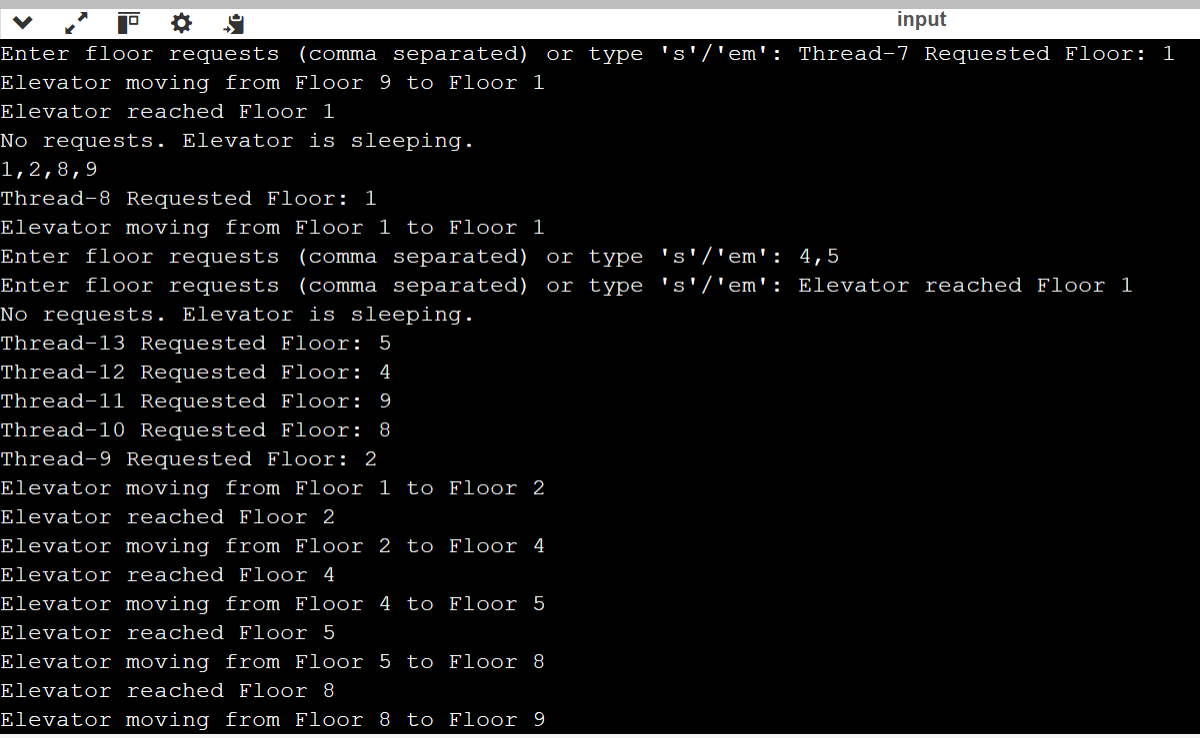
scanner.close();

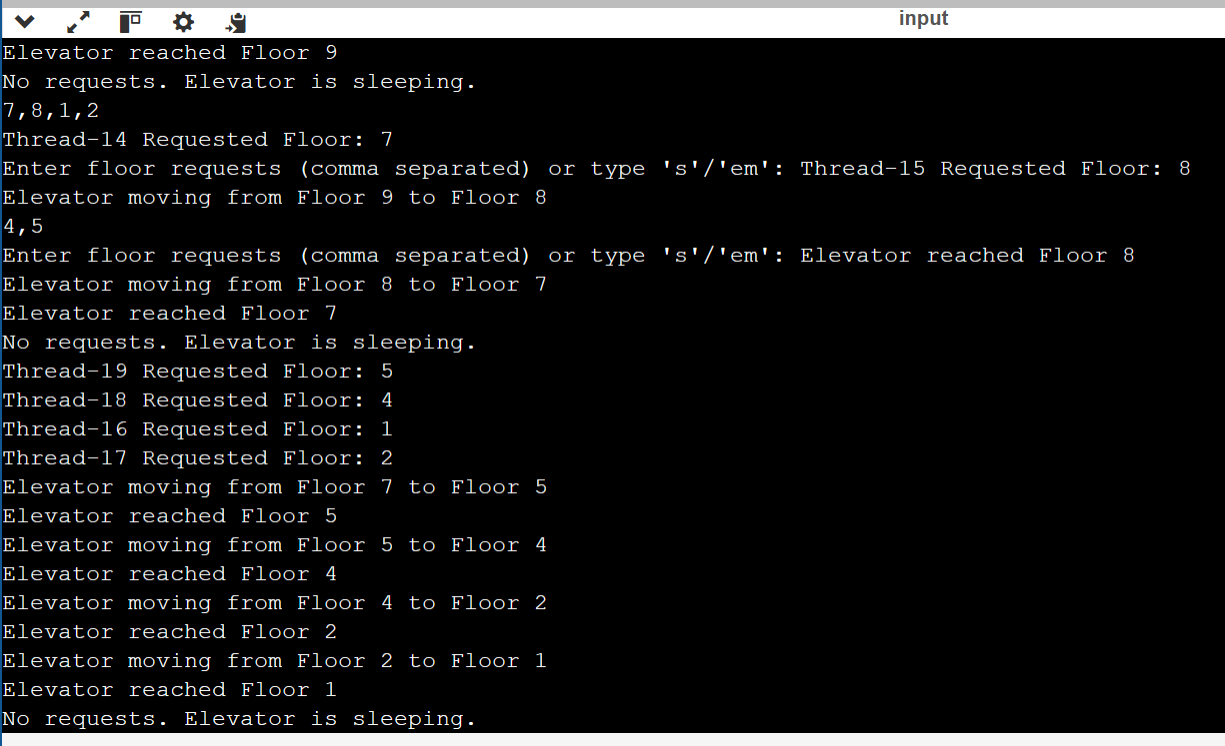
}

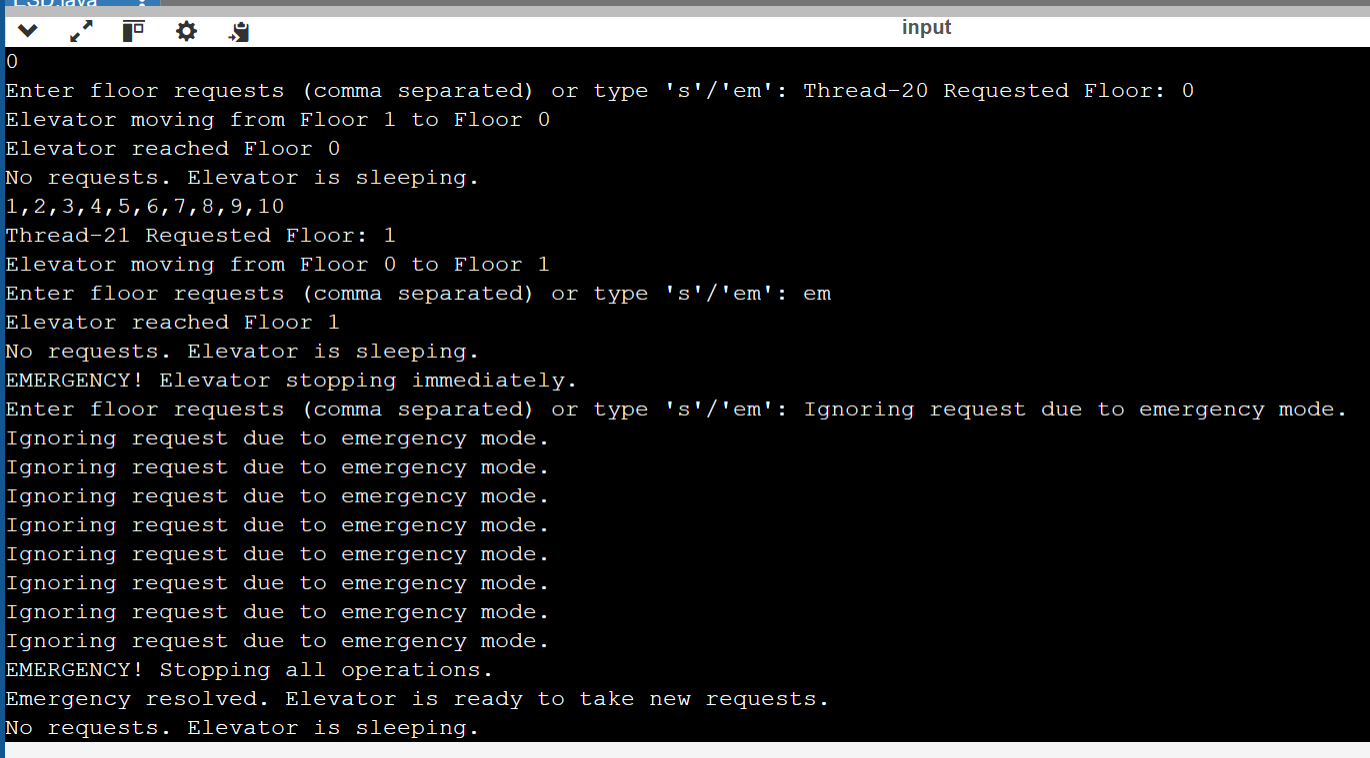
}

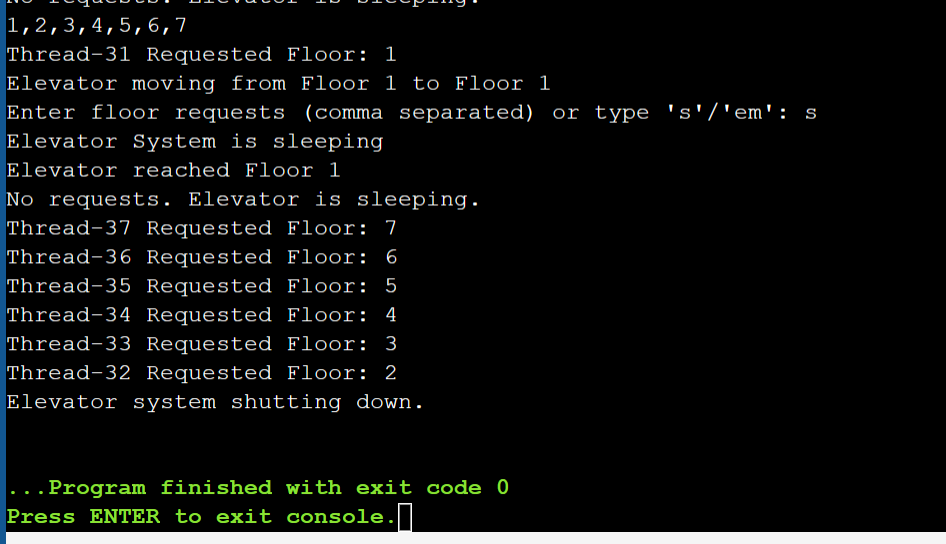
**Screenshot:**



****

****

****

****

****