Dining in Hell: A Monitor-Based Solution (Single FIFO Queue)

Group 35

February 10, 2026

1 Problem Statement

In the Dining in Hell variant, $N = 5$ people sit around a pot of stew. Each person has a spoon that can reach the pot but cannot feed themselves, so they must feed one another. We model each person as a thread. The requirements are:

- Threads must coordinate so that **no two or more people feed the same person simultaneously**.
- The algorithm must be **starvation-free**: no person should be postponed forever.
- The program must print a trace of interesting events (e.g., who feeds whom).

2 High-Level Idea

We implement synchronization using a single **monitor** (a Java object with `synchronized` methods). The monitor maintains a single FIFO queue that represents the current order of who should be fed next.

Key rule: The person at the head of the queue is the next one who should be fed. Therefore:

- If a thread discovers that it is at the head, it must wait (since it cannot feed itself).
- Otherwise, it removes the head from the queue (selecting a target) and feeds that target.
- After feeding, it enqueues the target at the tail so the rotation continues.

This design uses removal from the queue to represent that a person is currently being fed (so they cannot be chosen as a target by another feeder until they are added back).

3 Monitor (Concurrent Object) Design

The shared object `Table` contains:

- A FIFO queue `waitingQueue` holding IDs of people in the order they should be fed.
- Monitor methods (`synchronized`) that enforce atomicity and coordination using `wait()` / `notifyAll()`.

3.1 Operations

`Feed(feeder)` → `target` Called by a feeder to obtain someone to feed. If the feeder is currently at the head of the queue, the feeder waits. Otherwise, it removes the head (the next person to be fed) and returns it as `target`.

`Feeding(feeder, target)` Simulates the actual feeding action (prints a trace and sleeps for a random duration). This method is *not* synchronized so that feeding does not block other threads from accessing the monitor.

`Done(target)` Called after feeding completes. It re-inserts the fed person (`target`) to the tail of the queue and wakes waiting threads.

4 Why `notifyAll()` After `poll()` Matters

A subtle but important point is that after removing the queue head, the identity of the head changes. If the previous head thread was blocked in `wait()` (because it was at the head), it must be woken so it can re-check the condition and potentially proceed. Therefore, `Feed()` performs `notifyAll()` immediately after `poll()`.

5 Correctness Arguments

5.1 Safety: No two feed the same person simultaneously

A person can be selected as a target only by removing them from the shared FIFO queue in `Feed()`. Because `Feed()` is synchronized, only one thread can remove the head at a time. Once removed, the target is not in the queue until `Done(target)` runs, so no other feeder can select the same target concurrently.

5.2 Starvation-Freedom

Each time a person is fed, they are appended to the end of the FIFO queue. Thus, the head of the queue advances in a cyclic fashion, guaranteeing that every person reaches the head infinitely often and will be selected as a target repeatedly. Additionally, a thread only waits when it is at the head (i.e., it is the next to be fed), and other threads can always remove the head when it is not themselves, ensuring progress.

6 Implementation Mapping (Submitted Code)

The solution is implemented across:

- `Table.java`: the monitor and queue logic.
- `Person.java`: thread behavior: `Feed` → `Feeding` → `Done`.
- `Main.java`: creates the shared table and starts the threads.

7 Example Trace Output

The program prints events such as:

```
Thread 2 feeds thread 0
Thread 4 feeds thread 1
Thread 1 feeds thread 3
```

This provides an observable trace showing that feeding occurs between distinct threads and rotates through targets.

8 Conclusion

We solved Dining in Hell using a single monitor-protected FIFO queue. The queue provides both mutual exclusion on target selection (safety) and round-robin style fairness (starvation-freedom). The implementation keeps blocking (`wait`) strictly inside synchronized monitor methods, while the time-consuming feeding simulation (`sleep`) is performed outside the monitor to maintain concurrency.