Task1: Report (1)

We used the deadlock-free code for this part. We observed that with the increase in number of threads, it was taking more time to finish same amount of meal. One possible reason is the overhead time added to start and finish the extra threads. Another is the shared resource, fork, which the philosophers are sharing and blocking each other.

Number of philosophers	Number of meals	Time (in ms)
5	5	35.94
5	10	65.79
5	50	130.44
10	5	55.86
10	10	59.72
10	50	143.15
100	5	275.65
100	10	233.09
100	50	255.61

Task 1: Report (2)

We removed the deadlock-prevention part for this code. And as expected, when using only 5 threads, we suffered deadlock two times. The deadlock happened possibly because Thread.yield() made OS to kick the thread out of CPU after getting left fork. The program didn't suffer deadlock every time. That's because Thread.yield() doesn't guarantee that the OS will kick out the current thread.

Number of philosophers	Number of meals	Time (in ms)
5	5	Deadlock
5	10	49.35
5	50	Deadlock
10	5	54.45
10	10	77.19
10	50	135.93
100	5	290.18
100	10	281.21
100	50	312.83

Task 1: Report (3)

Part A: [w/shared fork]

Number of philosophers	Time taken for 4 meals (in ms)
1	N/A [Not enough forks.]
4	75.30
16	184.88
64	397.80

Part B: [without/shared fork]

Number of philosophers	Time taken for 4 meals (in ms)
1	37.71
4	59.21
16	150.99
64	330.55

My observation is that:

w/shared fork they took extra time as this is the case of parallelism with shared/limited resources because the philosopher must wait for another philosopher to be done before starting their meal.

without/shared fork it took philosophers less time than w/shared fork because the philosopher didn’t have to wait for any other philosopher. This was an example of maximum parallelism. All the philosophers were eating their meal independently.

Task 2: Report (1)

The reading agents and coordinating agents' task is different from the philosopher task because philosopher task has two classes (Main.java and Philosopher.java) and the reading-coordinating task has three classes (Main.java, ReadingThread.java, and CoordinatingThread.java).

Also, in reading-coordinating task, the threads are doing acquire and release operation on two different semaphores such that N readers and 1 coordinators pattern is maintained.

Also, reading-coordinating tasks didn't demand the concept of barrier either. However, there is an internal barrier (rTurn and wTurn semaphores in my code) which helps threads maintain the N reader and 1 coordinator pattern. But it is not a straightforward implementation of barrier like for etiquette in the philosopher's task.

Also, the reader and the coordinating agent threads do acquire and release operations on two different semaphores (rTurn and wTurn semaphores in my code) whereas in philosopher problem, there is not any concept which requires such type of implementation.

Task 2: Report (2)

The reading-coordinating agents program works on N readers and 1 writer pattern. When we used a large value for ‘N’, there was very limited writer’s access. N ensures that N number of readers must read before 1 writer can write. There were basically just the readers in the buffer most of the time before they all were done reading. This led to writers’ starvation.

Task 4: Report (1)

We followed a top-down approach by designing synchronization using semaphores first, then implementing logic.

For philosopher's problem, we implemented etiquette with the help of barrier. The concept was basically all the philosophers were kept in the sleeping queue until all the philosophers are done doing it. And the last philosopher woke all of them up.

Another challenging part in this task was to prevent deadlock, which we did by changing the pattern of grabbing fork of the last philosopher, all philosophers grabbed their left chopstick first where the last philosopher grabbed its right chopstick first.

Also, we used the semaphore (mutex), when any thread needed to work with number of meals eaten. This was to prevent extra meals being eaten as the threads operate simultaneously.

For readers-coordinators problem, the only challenging part was to maintain the N readers and 1 coordinator pattern. To implement this, we simply made Nth reader thread to wake up 1 coordinator thread, and that 1 coordinator thread on leaving wakes up 1 reader thread. The reader threads kept waking up until Nth thread, and then the same process continues.

This would have been easy if we were not expected to deal with edge cases. The edge cases, like with 0 readers and 0 writers, or what about remaining readers or writers after N readers and 1 writer pattern. We fixed it by using semaphores, conditions checks, and multiple static variables inside the readingAgent class and coordinating Agent class (in context of my code).

For command line problem, it was not bad at all. We created a main class. Depending upon the arguments, the main class evokes the required function. I put "main function" content from each problem (philosopher and reader-coordinator) in their respective function. I copied rest of the classes (philosopher class, readingAgent class, and coordinatingAgent class) as they were. It was the easiest problem.

Task 4: Report (2):

DSA I used for each task includes:

Dining Philosophers

- Array of semaphores (chopsticks) for resource control
- Binary semaphore (mealMutex) for shared meal counter
- Binary semaphore (mutex) for barrier synchronization
- Barrier-like algorithm using semHold
- Deadlock avoidance using asymmetric chopstick acquisition
- Circular indexing ((i + 1) % P)

Readers–Coordinators

- Counting/binary semaphores (sharedFile, rTurn, wTurn)
- Binary semaphore (mutex) for shared counters
- N-readers / 1-writer scheduling algorithm
- Batch processing of readers (limit = N)
- Conditional signaling between readers and writers

Command Line

- Manual argument parsing using args[]
- Input validation and control flow selection