Dining Philosophers eat, think and talk

Task 1. The Philosopher Class

This class describes actions that a philosopher can do. The actions are not synchronized, i.e. they can be accomplished in parallel by several threads (philosophers). The following modifications were implemented:

- eat() method:
    - Printing methods were added to indicate that a philosopher started and finished eating. Method getTID() is used to determine which philosopher is eating.
- think() method:
    - Printing methods were added to indicate that a philosopher started and finished thinking. Method getTID() is used to determine which philosopher is thinking.
    - Similar to the eat() method, a sleep method is added with a random number generator to determine how long the thinking process will go on.
- talk() method:
    - again, printing methods were added to indicate that a philosopher started and finished talking. Method getTID() is used to determine which philosopher is talking.
- run() method:
    - Talking decision is based on a random number generator with a 50% probability.
    - Before and after talking, the philosopher requests and ends talk with the respective methods from the Monitor class (requestTalk(piTID) and endTalk()).
    - requestTalk() method definition was changed to send the philosopher's ID into the method, so that we can determine that this specific philosopher (the caller) is not eating.

Task 2. The Monitor

The monitor class is the key class for this assignment. We used the implicit Java monitor with synchronized methods for atomicity and wait()/notifyAll() methods for blocking/waking up threads. This is a detailed description augmented with comments in the code:

- The following *data members* were added to class Monitor:
    - `private boolean [] chopstick_arr:` the array to track chopsticks. True value means that a chopstick is on the table and available, false means that it is in use.
    - `private boolean isTalking:` flag to track if a philosopher is talking.
    - `private boolean [] eating:` flag array to track if a particular philosopher is eating, so that he/she does not eat and talk at the same time.
    - `private Queue <Integer> hungryQ:` queue to track hungry philosophers to avoid starvation for food.
    - `private Queue <Integer> talkingQ:` queue to track philosophers intending to talk, to avoid starvation for talk.
    - `private int prev`: variable to track the philosophers previously added to the hungry queue, helps to improve concurrency due to the fact that neighbors cannot

eat at the same time as they use one same chopstick (see below for implementation details).

- *Initialization*:
    - The Monitor constructor is used for initialization to establish how many chopsticks we need and to initialize other variables.
    - The *chopstick_arr* array is initialized with true: all chopsticks are available.
    - The *eating* array is initialized with false: no one is eating.
    - The *prev* variable is initialized with 0 – meaning there is no previous philosopher added to the queue as their numbering starts with 1.
- *Handling chopsticks*:
    - There are two methods to handle chopsticks, pickUp() and putDown(), both taking the philosopher ID as piTID variable.
    - A philosopher can pick up his/her chopsticks if they are both available. Philosopher 1 uses chopsticks 0 and 1, philosopher 2: 1 and 2, etc., and *philosopher N uses chopsticks N-1 and 0*.
    - So, we block the thread in pickUp() if the chopstick to the left or right is unavailable, using modulo operation (piTID % chopstick_arr.length) for the right chopstick to make our table circular.
    - To avoid starvation, a hungry philosopher is added to the queue hungryQ. If a philosopher trying to pick up chopsticks is not the next in queue, then he/she has to wait.
    - To improve concurrency with the queue, we need to remember that neighbors cannot eat at the same time. So, if the current philosopher is a neighbor of the previous philosopher ((piTID ± 1) % chopstick_arr.length), we make him/her wait for 10 ms, so that a non-neighbor has a chance to be added to the queue before the neighbor.
    - After all waits are over, the philosopher is removed from the hungry queue, chopsticks are picked up (assigned false), *eating [piTID]* is assigned true and all other threads are notified of the opportunity to move on. The eating process starts.
    - Once eating is completed, *putDown* operation is performed by setting the *eating [piTID]* to false, putting down chopsticks (assigning true to the respective elements of *chopstick_arr*) and notifying all other threads.
    - It is important to note that all *wait()* methods except for the timed one are performed within while loops, so that even if a process wakes up but the conditions for its progress are not fulfilled, it is blocked again.
- *Talking*:
    - If talking is requested, the candidate is added to the queue.
    - Talking is possible if three conditions are met, otherwise the process is blocked:
        - The talker is the next one in queue (avoidance of starvation)
        - No one else is talking (a flag *isTalking* is used for this purpose)
        - The talker is not eating (checked with *eating[piTID]* array).
    - Right before talking begins, the talker is removed from the queue, *isTalking* flag is set to true and all other processes are notified.
    - Upon completion of talking, *isTalking* is set to false and all processes are notified.

So, let us briefly describe solutions for each sub-item of Task 2:

1. *Atomic pick-up of chopsticks*:

All methods in the monitor are synchronized, thus atomic. When picking up chopsticks, we determine their availability by checking if the left and right chopsticks are available, using modulo operation. If they are unavailable, the process is blocked (the philosopher has to wait).

2. *Handling starvation:*

Starvation is handled by queues. For talking, a thread is added to the queue upon its request to talk and removed from the queue immediately before talking.

For eating, it is a bit more complicated. If we implemented the queue without a staggering mechanism, the threads would be added in the following order: 1, 2, 3, 4, 5, repeat (the way the run() works). This would mean that though philosophers One and Three can eat at the same time, they will not, as after One, Two is next in line, and they cannot eat simultaneously.

The solution is to add a staggering mechanism, so that the queue would go 1, 3, 2, 4, etc. For this purpose, if the current thread happens to be the neighbor of the previous thread, it is sent to wait() for 10 ms, thus an opportunity is given to a non-neighbor of the previous thread to add itself to the queue before the neighbor. This provides better concurrency with starvation still avoided, though the bounded wait for a specific "delayed" process may be longer (but not by many turns as it is added to the queue anyway and so it is guaranteed to run).

3. *Talking*

Talking is possible only when the requesting philosopher is the next one in line to talk, is not eating, and there is no one else talking at the same time.

Task 3. Variable Number of Philosophers

Changes were made to the DiningPhilosophers.java class to accommodate the requirement to accept the number of philosophers as a parameter.

In Java, parameters as submitted as an array of strings received by the *main* method traditionally called *args[]* (unlike in C, no *argv* value that specifies the number of parameters is required as *args* is an array, not a pointer). The names are arbitrary, though.

The iPhilosophers variable is initiated with the constant called *DEFAULT_NUMBER_OF_PHILOSOPHERS*.

Then the following logics are used:

- If the number of arguments is 0, we use the default parameter. If it is more than 0:
- We test the args[] for length. If it is over 1, an exception is thrown.
- If the number of arguments is 1 (the correct number), we try to parse this value for integer. If we succeed and this number equals or exceeds 1, then we check if it equals 1. As the problem of Dining Philosophers requires more than 1 philosophers present, the received value is changed from 1 to 2 and assigned to the iPhilosophers variable.
- It is important to note that irrespective of error in the parameter, one exception is thrown, with the message specified in the assignment. The thrown exception if of

NumberFormatException() class – this exception is thrown by the Integer.parseInt(String) method, which helps to avoid code repetition.

Task 4. Starvation

In this assignment, starvation of the dining philosophers is possible when they request eating or talking. In both cases, a queue is used to resolve starvation and establish the upper bound on waiting.

1. Preventing Eating Starvation:

Whenever a philosopher requests to pick up chopsticks, his/her number is added to the queue "*hungryQ*". If the philosopher is the neighbor of the previous philosopher (tracked with variable *priv*), we suspend him/her with *wait(10)* for 10 msec. This way we try to **stagger** the queue as following: 1, 3, 2, 4 … instead of 1, 2, 3, 4, as in the latter case concurrency is adversely affected: 2 cannot eat at the same time as 1, and 3 who can eat in parallel with 1 has to wait for his queue after 2.

If a philosopher is not next in queue, his/her thread is blocked, until they are next in queue.

Thus, the philosophers do not wait longer than the number of the other philosophers in queue before them, establishing the upper bound on waiting.

A philosopher is removed from the queue immediately before he/she starts eating.

2. Preventing Talking Starvation:

The same queue-based mechanism is used with the queue called *talkingQ*, but without the staggering mechanism as it is not required: anyone can speak after anyone. The upper bound for waiting is the number of the other philosophers registered in the queue when the current philosopher is added to the queue. For example, if the queue goes: 1, 2, 3 and we add 4, then Four has to wait for three turns.