SYNC - Assignment 4

By Arthur ADAM and Alexandre FLION

Please note that every file mentioned in this PDF is provided in the Canvas Submission.

Part 1 - Dining Philosophers

The dining philosophers problem is a synchronisation problem where 5 philosophers wants to eat a huge plate of spaghettis.
Each philosopher must have 2 forks to serve from the plate: the one on his right and the one on his left.

The version of this problem must use conditional variables, not using hold-and-wait deadlock conditions.

A. Variables

The program has 5 variables:

- A number of philosophers
- A Mutex
- A list of indices difining which philosopher is running in which loop
- An array of condition variables, one for each philosopher
- An array of booleans, one for each philosopher

When thread is launched, the thread gets an index defining which philosopher is running.
After getting his index, the thread launches the loop.

B. Loop

When loop is launched, the thread checks if the left and right forks are free. If forks aren't availables, the current philosopher's condition variable waits until the forks are availables.
If forks are availables, the current philosopher takes his left and right fork setting left and right index booleans to 'True'.
After this, the thread gets spaghettis and then frees forks. Then, the current philosopher notify left and right philosophers condition variables to carry on the loop.

Each part of the loop is encapsulated between `mutex.wait()` and `mutex.signal()`.

Part 2 - Dining Savages

The dining savages problem is a synchronisation problem where `x` carnivore savages and `x` vegetarian savages serves from a pot. The loop works at follows:

- When a vegetarian serving is in the pot, a vegetarian savage eats the serving from the pot.
- When a carnivore serving is in the pot, a carnivore savage eats the serving from the pot.
- If the pot is empty / doesn't contains the corresponding type of serving, a savage should trigger the corresponding cook to make a serving.

A. Variables

Our implementation of this problem contains the following variables:

- 3 mutex: one for the carnivores savages, one for the vegetarians savages and one for the MyBag handling
- A variable of type 'MyBag' to handle the servings
- 2 conditional variables linked to the mutexes
- 2 Semaphores in order to trigger the cooks

B. Loops

I. Savages

The savages loops are the pretty much the same. At first, the thread waits the mutex corresponding to its type (carnivore / vegetarian). After this, it triggers the corresponding cook in order to cook the right kind of serving.
When the cook is triggered, we're making a loop checking if the serving list contains the corresponding kind of meal for that savage. If not, a conditional variable waits for a notification. If the pot contains a meal, the savage is going to get that serving and eat it, inside a `wait` / `signal` pattern for the bag's mutex. After eating, the savage signals its mutex in order to carry on.

Each time the savage tries to access the MyBag variable, the statements are encapsulated between the bagMutex variable `wait` and `signal`.

II. Cooks

The cook loops are as the savages one, pretty similar to each other. At first, the cook thread waits to be triggered. When triggered, the cook access the bag and put a new serving corresponding to its kind. After putting the new serving, the cook notifies the savages that a new serving is available.

Part 3 - Readers / Writers

The reader writers problem is a synchronisation problem where a group of threads can read a shared ressources that writers can update.
The implementation should use condition variables and configurable priority to choose which group has priority.
Unfortunately, while implementing, we faced issues with configurable priority. Thus, we didn't implemented it and only implemented the condition variables.

A. Behaviour

Our implementation uses a global mutex to protect the shared ressources. When a reader or a writer tries to launch its loop, it makes the other wait thanks to the `ressourceMutex.wait()`.

I. Reader's Loop

In the reader's loop, after entering the loop, we are checking if any writer is busy writing at the same moment or if any writer is waiting, if so our condition variable makes other reader thread wait.
If no writer is busy anymore, we're implementing the number of readers and our reader reads the ressource. After reading, the number of readers is decremented and the condition variable notifies all threads waiting.
Finnaly, the loop signals the `ressourceMutex` and go back for another loop.

II. Writer's Loop

In the writer's loop, after entering the loop, we're directly updating our `isWriterWaiting` boolean to True. That should indicate that a writer is ready to write and lock the reader's loop. If there is at least one reader or one writer working, the current writer's thread waits until these conditions are false.
If the writer gets out of the waiting loop, we're reseting our boolean value to False and incrementing the number of writers by 1. Directly after this, since the `ressourceMutex` is taken by this thread, we can access it and update it without race condition.
After writing, the number of writers is decremented and the condition variable notifies all threads waiting.
Finnaly, the loop signals the `ressourceMutex` and go back for another loop.