Operating Systems–2: CS3523 2019 Programming Assignment 5: Implement Dining Philosopher's using Conditional Variables Report

Sai Harsha Kottapalli
CS17BTECH11036

April 9, 2019

1 Aim

We have to solve the Dining Philosopher's problem using conditional variables.
We then compare the average and worst-case time taken for each thread to access the critical section (shared resources) for different number of threads.

2 Design & implementation of the Program:

We use conditional variables and a mutex to implement the Dining philosophers using monitor approach.

2.1 Design

1. Class **Dphil** acts as the the monitor in the c++ program.

    - we define THINKING, HUNGRY, EATING states accessible by each philosopher.

- A list of conditional variables, one for each philosopher.
- A mutex **monitor** for implementing monitor's mutual exclusion property(wrapped for every function inside the class).
- We initialise with all philosophers in THINKING mode, mutex monitor to be available.
- **test()** to check if a particular philosopher can eat at a given instance.
- Algorithm:

```
void test(int i) {
        if ((state[(i + (n_p - 1)) % n_p] != EATING) &&
                (state[i] == HUNGRY) &&
                (state[(i + 1) % n_p] != EATING)) {
                state[i] = EATING;
                pthread_cond_signal(&self[i]);
        }
         return;
}
```

- **pickup()** - calls test() to check if this philosopher can eat, else calls wait().
  Here, pthread_cond_wait() unlocks the mutex associated with it, before suspending this thread.
  When unlocked, the mutex is then locked.

- Algorithm:

```
void putdown(int i) {
        state[i] = HUNGRY;
        test(i);
        if (state[i] != EATING)
                pthread_cond_wait(&self[i], &monitor);
        return;
}
```

- **putdown()** - philosopher is done eating, puts down the chopstick(allows neighbours to pick this chopstick) and starts thinking.
- Algorithm:

```
void putdown(int i) {
        state[i] = THINKING;
        test((i + (n_p - 1)) % n_p);
        test((i + 1) % n_p);
        return;
}
```

- We have to call functions in the following order:

```
philosophers.pickup(id);
\\ EAT - CS
philosophers.putdown(id);
```

2.2 Critical Section :

This part of the code should be accessed by atmost two non-consecutive philosophers respectively at a time.

2.3 Entry Section :

This part of the code is accessible for all the philosophers. The job of this code is to make sure atmost two non-consecutive philosophers are given access(allowed to take the corresponding chopsticks) to the critical section at a particular time.

2.4 Exit Section :

This part of the code is where the philosopher has executed the critical section(Ate) of the program, and will putdown his chopstick so that other philosopher can now enter the critical section(if any) while this philosopher completes the remainder section parallelly(Thinking).

2.5 Average Waiting Time :

This is defined as the amount of time each access to critical section takes after requesting for the same.

2.6 Worst Waiting Time :

This is defined as the highest amount of time taken by a philospher respectively to critical section takes after requesting for the same.

2.7 getInput()

This is a helper function which helps in keeping the code modular.
It reads the value of the following from the input file, specified in ReadMe.txt

- **Number of philosophers** - n
- **Number of readers** - h

- **lam1 and lam2**
  Represents the mean of value of two exponential distribution curves which is to be constructed.
  Given in milli-seconds.

2.8 dining()

The dining() function simulates a philosophers trying to get chopsticks to eat.
Only one phi at a time, can be inside the CS to avoid synchronization issues.

2.9 currTime()

Takes input time_t structure to generate the time in HH:MM:SS syntax and return this as a string.

2.10 Other important variables used

- **exponential_distribution**
  Produces random non-negative floating-point values x, distributed according to probability density function defined for exponential distribution about the constant rate given as a parameter.

- **default_random_engine**
  This is a random number engine class that generates pseudo-random numbers.

- **thread *th for writers and readers**
  th_w and th_r are the pointers for storing the array of nw and nr threads respectively. This will be useful for later calling **join()** to properly exit the threads spawned previously.

- **Average and worst waiting times for writers and readers**
  Used to calculate the average time as well as worst waiting time, a writer/reader has to wait for getting access to the CS.

3 Conditional Variables

A condition(conditional variable) is a synchronization device which can be used to suspend threads until some condition is satisfied.
There is also a mutex associated with this conditional variable to avoid the race condition where a thread prepares to wait on a condition variable and another thread signals the condition just before the first thread actually waits

on it.

4 Output

- Log file for dining philosophers.
  This stores the events which occur throughout the simulation.

  Example :
  ```
  1st eat request by Thread 1 at 15:37:51
  1st eat request by Thread 4 at 15:37:51
  1st CS Entry by Philosopher Thread 4 at 15:37:51
  1st eat request by Thread 3 at 15:37:51
  1st eat request by Thread 2 at 15:37:51
  1st CS Exit by Philosopher Thread 4 at 15:37:51
  ```

- Times.txt
  This file stores the average times as well as worst waiting time for philosophers.
  The values are in milli-seconds.

  Example:
  ```
  Average waiting time for philosophers(ms): 0.804050
  Worst time taken for philosopher to eat(ms): 6.597000
  ```

5 Comparision

5.1 Graphs

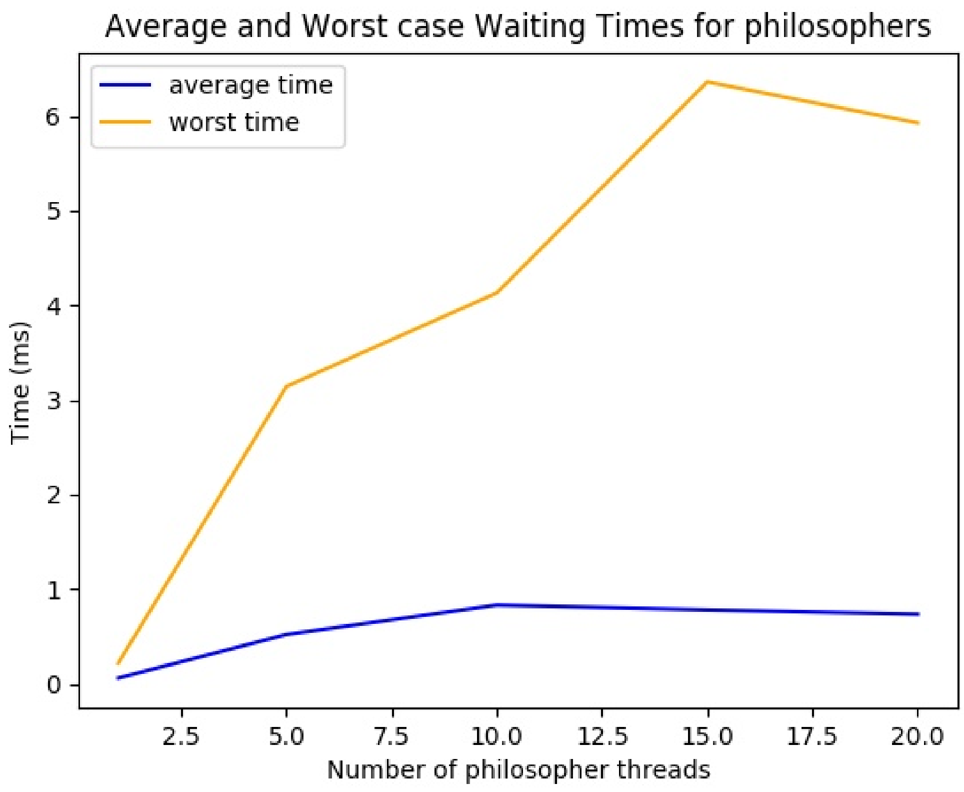

Figure 1: Average time taken and worst time taken to enter the CS by the philosophers

5.2 Notes

- Parameters used for test cases:
  n = 1, 5, 10, 15, 20
  h = 10
  Λ_1 = 1 milli-seconds and Λ_2 = 2 milli-seconds

- Each of the plotted points is the average of 5 testcases.

- We can infer that, as number of philosopher threads increases, the average value converges to some value.

- As for worst case waiting time, though irregular, in general the worst case waiting time increases, as number of threads increases.

- This is because, the given implementation has a drawback, that is starvation, hence the worst case waiting time will increase as number of philosophers increase.