Operating Systems - 2
Spring 2019

Instructor - Dr. Sathya Peri
Jatin Chauhan - CS17BTECH11019

Design of Program:

The program has been designed to solve *dining philosophers* problem using conditional variables in c++. The program has been designed to ensure multiple ***non-adjacent*** philosophers can eat and no race conditions arise, while allowing multiple threads to read . The libraries used in the implementation are -

<pthread.h> - for creating threads, condition variables and mutex locks
<random> - To create random number generating engines.
<unistd.h> - To calculate current system time.
<fstream> - Creating file streams.

Various sections of code are explained below:

1) ***Entry Section:*** The segment writing CS request time to output file.
2) ***Eat Section:*** The segment requesting chopsticks, waiting the current philosopher if both the sticks are not available and acquiring them.
3) ***Remainder Section:*** The segment of code writing CS exit time and performing remaining operations.

The variable - **av_time** stores the average waiting time and the variable **worst_time** stores the worst case waiting time for any thread. The worst_case time gives us the empirical idea of how much a thread has to starve.

Implementation of Program:

The main function takes the necessary inputs from the input stream. Creates - **n conditional variables,** where n is the number of philosophers. The worker threads are created and their attributes are initialised.

The function **func_philosopher** is the runner function for the threads. The important variables are described below:

num_philosphers: Number of threads
av_time: to store average waiting time
Worst_time: to store worst case waiting time of a thread
average_think_section: value of mean for distribution for simulating remainder section.
average_eat_section: value of mean for distribution for simulating eat/critical section.
eat_engine, think_engine: random number generating engines which gives the means for generating a time value for eat section and think section simulations.
Chopsticks: boolean array showing the availability of chopsticks.
pthread_cond_t *cond_var: is the array of condition variables which hold the conditional variables. It is initialised as -
cond_var = new pthread_cond_t[num_philosophers];
, in the main function.
pthread_mutex_t mutex: to protect the conditional variables.

The runner function for each thread starts by writing the current time to the log file, which is basically the time of “CS entry request by the thread”. Then it tries to acquire the *mutex,* inside which it checks of both the **left and right** chopsticks are available or not. If yes, then it acquires them by setting their values to false and release the lock. It then enters its critical section and invokes the sleep (showing it is doing some time intensive operation).

Note: Other threads who are not adjacent to current thread, can still read the values of their respective chopsticks, but since one of the chopsticks will missing for both, they won't be able to eat. Also, if any other philosopher not adjacent to the ones already eating tries to acquire the chopsticks, he can, because both the chopsticks are free at the time.

After doing the sleep operation, the *mutex* is acquired again and the values of *av_time and worst_time* are updated. Also, the current thread will set the values of its chopsticks to *true* and signal both its neighboring threads, implying they can try and eat now. After this, it releases the lock and enters its think section, where it sleeps again to simulate think process.

In the above implementation, the adjacent threads can eat in a mutually exclusive manner, while allowing multiple threads to read (or to try and acquire their chopsticks). Non-adjacent threads can eat by acquiring their chopsticks any time if they are free.

Graphs:

The graph contains two bar-plots corresponding to worst case waiting time and average waiting time for the threads for various values of n - 1, 5, 10, 15, 20. The value of h is fixed to 10, whereas the the values of average parameters for the distribution are set to 1 and 2 respectively.

The values are -

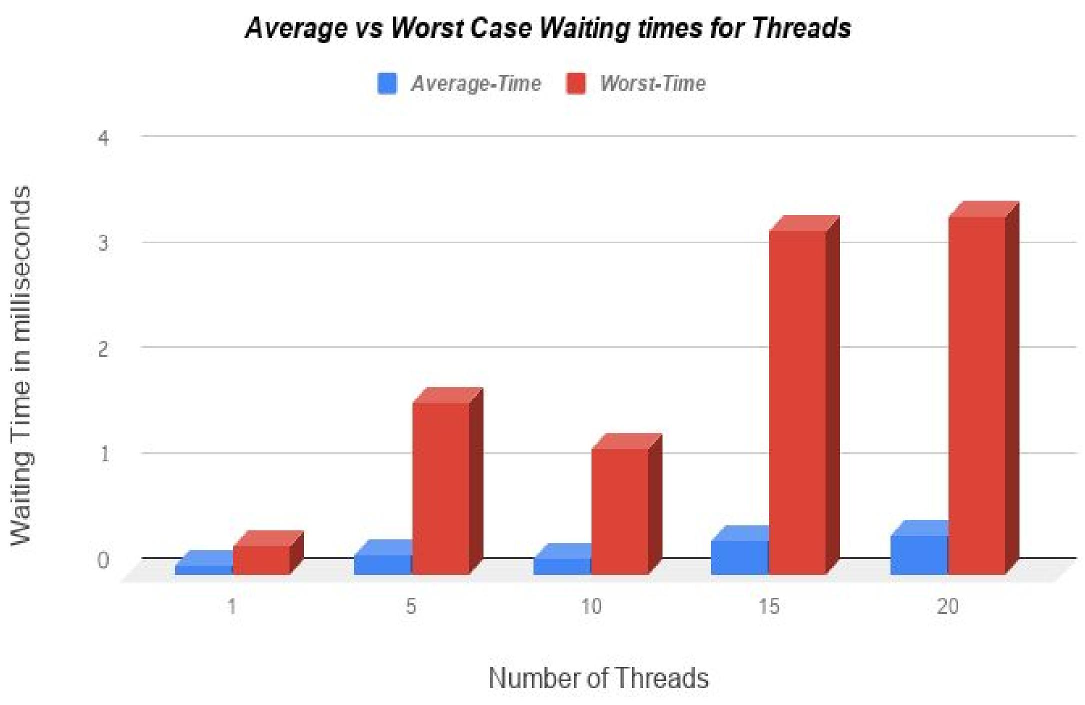
Note: The times for only 1 thread just represents some overhead of acquiring the *mutex lock and other operation on condition variable.*

The actual comparison starts from 5 and above.

Num Threads	Average Time	Worst Case Time
1	0.07	0.26
5	0.17	1.62
10	0.14	1.19
15	0.31	3.25

20	0.36	3.41

The corresponding graph is -



We can clearly see that worst case waiting time is atleast 7fold than that of waiting time for all values. The anomaly between number of threads 5 and 10, might be because of other processes running on CPU’s. We can clearly see that some threads tend to starve (from the waiting time). It was also empirically observed from the log file, thus strengthening the claim that threads can potentially starve. The values tend to show increasing behaviour, which follows, since more the number of threads, more the competition for acquiring the locks and eating.