Assignment 4 Report

Shaunak Badani
20171004

1. Concurrent quick sort

Processes :

- A global array has been declared for all processes.
- The quicksort function is invoked.
- This function quickly performs an insertion sort if the number of elements is less than 5, and returns. Else :
- The array is partitioned around a particular pivot.
- The pivot in my code has been chosen as the first element of the array.
- The pivot is put in its right position, all the elements smaller than that are placed left of it, unordered, and all the elements bigger than that are placed right to it, unordered.
- Now, two child processes are created.
- One of the children takes care of the left half of the array, call it the left child.
- The right child sorts the right half of the array.
- This recursion goes on until the two ends of the function parameters do not converge.

Threads :

- A single thread is created, whose *start_routine* parameter, i.e. the function that the thread calls upon creation is *threaded_quicksort*.
- The partition function is called first. The pivot element is placed in its right position.
- After the partition function is over, two different threads are created. One to sort the left half, one to sort the right half.
- Those two threads are then created, which goes on recursively until l<r.

Performance analysis :

- For small N [N <= 5], normal and concurrent quicksort are almost the same speed, while threaded quicksort takes a little longer to complete.
- For N >= 7, normal quicksort is the fastest, followed by threaded quicksort, and last but not the least, concurrent quicksort.

2. Robot chef problem :

Following are the global variables declared :

Data type	Variable name	Purpose
Long long int [array of 1000 elements]	robot_biryani_ vessels	The ith element of the array stores the number of biryani vessels prepared by the ith robot.
Long long int	no_of_students	No of students who will come in an orderly fashion to satisfy their hunger pangs with biryani.
Long long int	no_of_tables	No of serving tables available.
Long long int	no_of_chefs	No of robot chefs available to make biryani.
Long long int	time_consume_ biryani	Amount of time every student takes to eat biryani.
Long long int	wait_time_stud ent	The time gap between 2 students entering in line.
Tab_info [struct]	tables	Tab_info is a struct containing two integers : p -> feeding capacity of the serving container on the serving table. no_of_slots -> No of slots available for that particular table. Hence these two integers are recorded for every table in the mess.

The program starts with initializing the pipeline.
The *initializePipeline()* method creates threads for the simulation. *No_of_chefs* threads for the chefs, *No_of_tables* threads for the tables and *No_of_students* threads for the students.

Assumptions :

- The student will not be allotted a table where there is no biryani, as opposed to the clarification mentioned later on Moodle :
- “Student X has gone to table Y”
- “Student X has been served biryani”
- If we had to print these two statements in my simulation, they would be printed one after the other, as the student is not allotted to the table where there is no biryani.

Robot Chef Implementation :

- Every robot chef has its own thread. The thread starts with the execution of the *make_biryani* function. This function does the following :
    - Waits for previous biryani vessels prepared to be emptied into serving tables.
    - Gives some rest time before preparing the next batch of biryani.
    - Makes biryani within 2 - 5 seconds.
    - Prepares 1 - 10 biryani vessels in the time interval mentioned.
    - After that is done, it invokes the *biryani_ready* function.
    - This function runs in the particular thread until the no of students becomes 0, i.e until all students are served.
- The *biryani_ready* function updates the no of vessels prepared in the *robot_biryani_vessels* array, and returns.

Serving Tables Implementation :

- Every serving table has its own thread. The thread starts with the execution of *wait_for_biryani* function. This function does the following :
    - Looks for biryani vessels in the *robot_biryani_vessels* array.
    - If a biryani vessel is found, it allocates a certain feeding capacity (i.e. no of students it can serve) between 25 - 50.
    - Updates the *tables* array with the feeding capacity of that particular table.
    - Calls *ready_to_serve_table* function.
- The *ready_to_serve_table* function selects 1 - feeding_capacity no of slots, and assigns it to the particular table.

Student Implementation :

- Every student has its own thread. It invokes *wait_for_slot* function which does the following :

    - Looks for empty slots with tables which have non empty biryani vessels.
    - If such a table is found, the *student_in_slot* function is called.
- The *student_in_slot* function does the following :
    - Allots portion of biryani to student.
    - Waits for *time_consume_biryani* seconds during which the student eats biryani.
    - After that time, the student leaves that slot to make space for other hungry students standing in the circular Kadamb mess line.

3. Ober Cab services

Following are the global variables declared :

Data type	Variable name	Purpose
Long long int	no_of_riders	No of riders who will come eventually.
Long long int	no_of_drivers	No of drivers in the system.
Long long int	no_of_payment_servers	No of payment servers available.
Long long int	wait_time_rider	Wait time gap between every rider.
d_info [array of 1000 elements]	cab	d_info (short for driver info) is a struct which stores the following information of the cab : ● state of the cab. [wait_state, on_ride_pool_one, on_ride_pool_full, on_ride_premier]
p_slip [array of 10000 elements]	payment_servers	p_slip is a struct which stores the following information of the payment server : ● Whether the server is in use or not.

		• Rider no who has to complete the payment. • The cab no whose driver the rider has to pay,

Rider Implementation :

- Every rider has its own thread. It invokes *bookcab* function which does the following :
    - Looks for a cab of the type that the rider has requested for.
    - If the cab cannot be found for the rider in *maxWaitTime* seconds, then the rider times out, and the thread ends.
    - If a cab is found, it rides in the cab for *rideTime* seconds.
    - After that time has passed, it proceeds to empty the cab, while making space for new passengers to occupy the seat of the cab.
    - The rider proceeds to the payment gateway.
    - Here the rider waits until a payment server is active.
    - Once an active payment server is found, the transaction takes two seconds to complete, and the payment server is made active for other users to complete their transactions.
    - Then the rider exits.

Payment Server Implementation :

- Every payment server has its own thread. It invokes *check_payment* function which does the following :
    - Checks if the payment server of that thread has been activated or not.
    - If yes, it waits for 2 secs for the transaction to complete.
    - As soon as the transaction is done, the payment server goes back to being inactive until activated by a rider that needs to complete his/her payment.