International Institute of Information Technology, Hyderabad

INTERNATIONAL INSTITUTE OF INFORMATION TECHNOLOGY

HYDERABAD

Course Instructor:
Mr. Maneesh Shrivastava

Advanced Operating Systems

Project Name: Problems on Process Coordination

Group Members:

1: Kushal Shah (2021202027)

2: Padam Prakash (2021201071)

3: Ziyad Naseem (2021201021)

4: Soumodipta Bose (2021201086)

Problem 1: Badminton Academy Problem

Introduction:

This problem focuses on thread coordination with help of condition statements and mutex. Each person is a thread in this problem statement and these threads are interdependent on each other under specific conditions.

Problem Description:

A group of 3 people want to play badminton at their courts. The group should consist of 2 players and a referee to manage the match. Whenever a new player/referee enters the academy, he/she waits until he/she gets a chance to meet the organizer. Whenever the organizer is free, he waits until a group of 2 players and a referee can be formed among the unassigned people he met and then the newly formed group enters the court. Then the players warm up for the game and in the meantime the referee adjusts the equipment necessary to manage the game. Once all the three are ready, the game is started by the referee. Organizer waits till the game is started and once it is started he is free and again starts meeting new players/referees entering the academy. Multiple groups can enter the academy and play badminton.

Constraints:

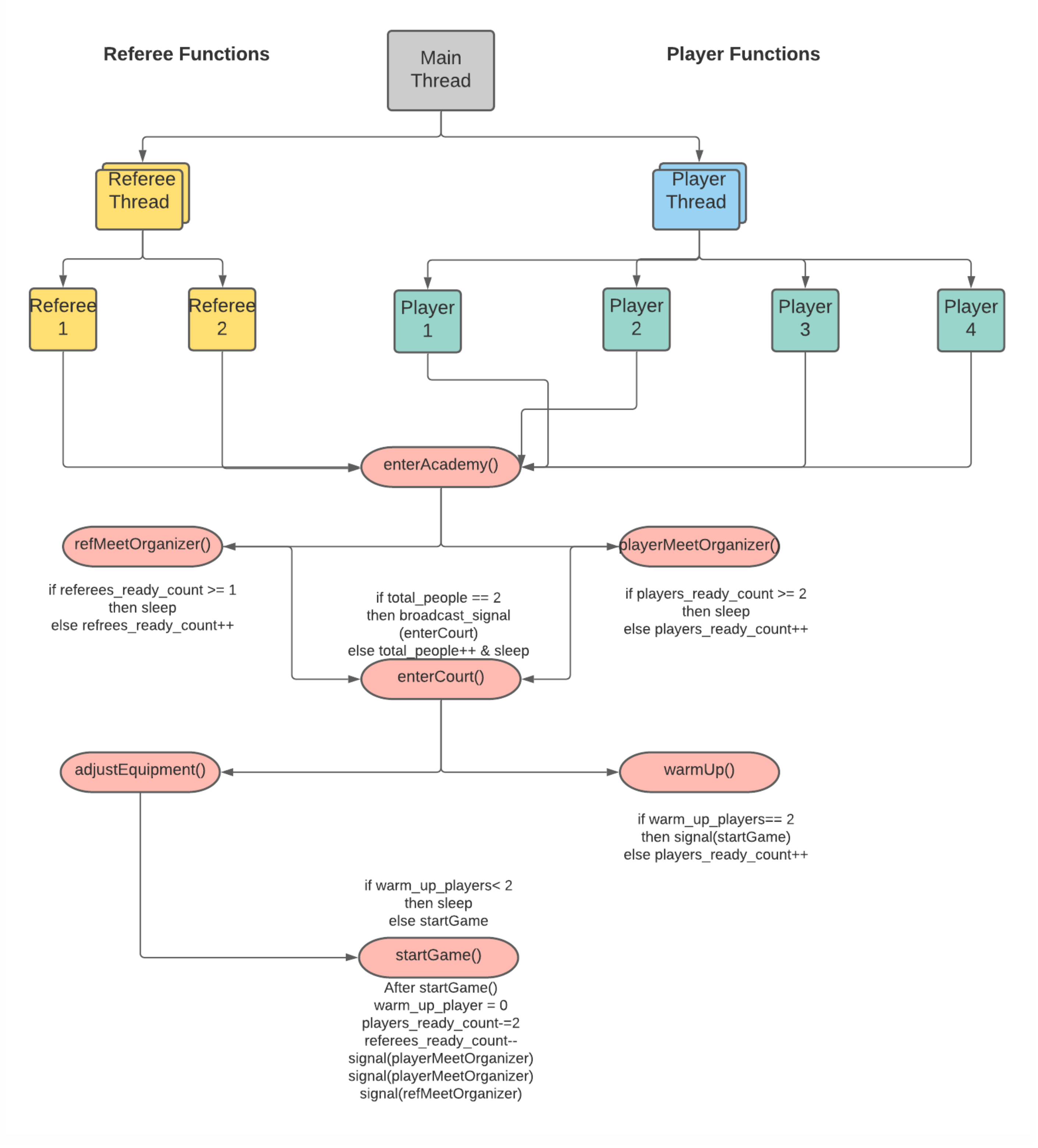
- Using pthread Library to implement threads
- Utilize Mutex and Conditional Variables instead of Semaphores
- Each person will be represented using Threads: Player and Referee
- At any given time only one group can meet the organizer until their match starts.
- For any particular group, all players and referees cannot enter the court until all of them have met the organizer.
- For any particular group, the referee cannot start the match until all players have warmed up.
- Should not suffer from deadlocks and no busy-waiting solution.
- Each new person arrives at the academy with a random, modulo 3 second delay.

- Players call functions **enterAcademy(), playerMeetOrganizer(), enterCourt(), warmUp()** and referees call functions **enterAcademy(), refMeetOrganizer(), enterCourt(), adjustEquipment(), startGame()**

Solution Approach:

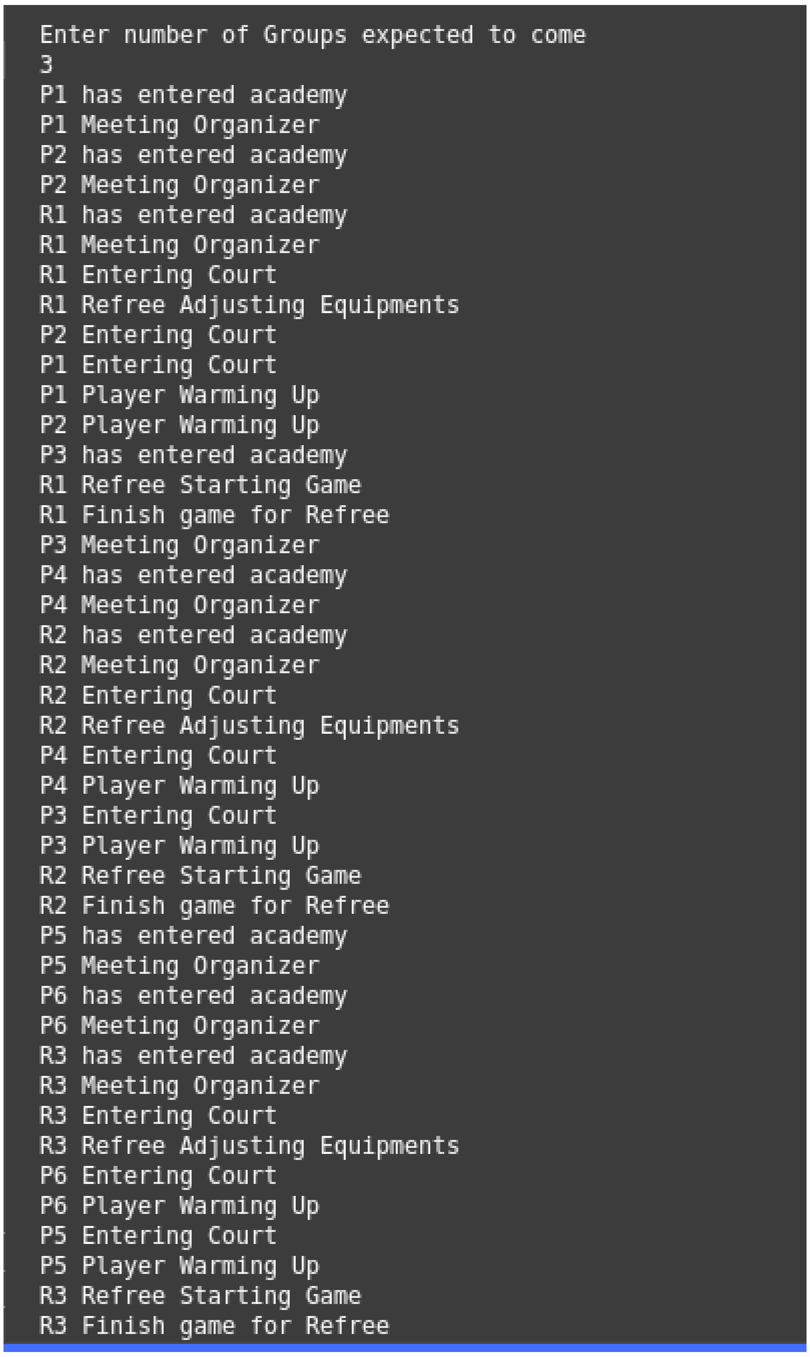
- Created threads for players and referees as they come with random, modulo 3 second delay.
- Both player thread and referee thread can call **enterAcademy()** when they are created irrespective of any conditions.
- When a player thread calls **playerMeetOrganizer()**, a global **players_ready_count** variable is incremented. Similarly when a referee thread calls **refMeetOrganizer()**, a global **refereess_ready_count** variable is incremented.
- If **players_ready_count >= 2** then new player threads calling **playerMeetOrganizer()** will sleep until they are signaled. Similarly if **refereess_ready_count >= 1** then new referee threads calling **refMeetOrganizer()** will sleep until they are signaled.
- Player and referee threads both call **enterCourt()** and until 3 threads call this function, previous threads sleep. After 3 threads call the function, the previous 2 threads wake up and they all enter the court.
- After entering court, the player thread calls **warmUp()** and the referee thread calls **adjustEquipment()**.
- Until both player threads have finished **warmUp()** the referee thread will not start the game. Referee calls **startGame()** and waits for player threads to finish warming up.
- The second player thread after finishing warm up signals the referee thread to start the match.
- After starting the match, the referee thread does **players_ready_count -= 2** and **referees_ready_count--** also it signals the **playerMeetOrganizer()** and **refMeetOrganizer()** to allow new player threads and referee threads to meet the organizer respectively.
- Thus a new group forms and the cycle repeats again until all groups have started their match.

System Design:



Referee Functions
Main Thread
Player Functions
Referee Thread
Player Thread
Referee 1
Referee 2
Player 1
Player 2
Player 3
Player 4
enterAcademy()
refMeetOrganizer()
playerMeetOrganizer()
if referees_ready_count >= 1 then sleep else refrees_ready_count++
if total_people == 2 then broadcast_signal (enterCourt) else total_people++ & sleep
if players_ready_count >= 2 then sleep else players_ready_count++
enterCourt()
adjustEquipment()
warmUp()
if warm_up_players== 2 then signal(startGame) else players_ready_count++
if warm_up_players< 2 then sleep else startGame
startGame()
After startGame() warm_up_player = 0 players_ready_count-=2 referees_ready_count-- signal(playerMeetOrganizer) signal(playerMeetOrganizer) signal(refMeetOrganizer)

Logs:



```
Enter number of Groups expected to come
3
P1 has entered academy
P1 Meeting Organizer
P2 has entered academy
P2 Meeting Organizer
R1 has entered academy
R1 Meeting Organizer
R1 Entering Court
R1 Refree Adjusting Equipments
P2 Entering Court
P1 Entering Court
P1 Player Warming Up
P2 Player Warming Up
P3 has entered academy
R1 Refree Starting Game
R1 Finish game for Refree
P3 Meeting Organizer
P4 has entered academy
P4 Meeting Organizer
R2 has entered academy
R2 Meeting Organizer
R2 Entering Court
R2 Refree Adjusting Equipments
P4 Entering Court
P4 Player Warming Up
P3 Entering Court
P3 Player Warming Up
R2 Refree Starting Game
R2 Finish game for Refree
P5 has entered academy
P5 Meeting Organizer
P6 has entered academy
P6 Meeting Organizer
R3 has entered academy
R3 Meeting Organizer
R3 Entering Court
R3 Refree Adjusting Equipments
P6 Entering Court
P6 Player Warming Up
P5 Entering Court
P5 Player Warming Up
R3 Refree Starting Game
R3 Finish game for Refree
```

Results:

- In the above example there are 3 groups. 2 Players and 1 Referee form one group
  P = Player & R = referee.
  P1, P2 & R1 = Group 1. P3, P4 & R2 = Group 2. P5, P6 & R3 = Group 3.
- We can observe P1, P2 and R1 form one group. Until a match starts between this group, no new players and referees can meet the organizer although they are free to enter the academy.
- P1, P2 and R1 enter the court all together when all are ready.
- R1 does not start the match until P1 and P2 have warmed up.

Problems faced:

- Many times players and referees were able to meet the organizer even though the previous group had not started the game. Several debug cycles were needed to add proper conditions so that the organizer would be locked until the previous group starts the game and then the organizer can meet new people.

Learnings:

- Designing a solution to multi threaded problems.
- Multi-Thread creation and intercommunication between Threads.
- Working of Conditional Variables and Mutexes and how to apply them to avoid race conditions among threads and also avoid deadlocks.

Problem 2:The Queue at the Polling Booth

Introduction:

This particular problem is a design problem that focuses mainly on designing a system and coming up with a solution to solve an issue that plagues Voting Booths and increases the efficiency of the voting process.

Problem Description:

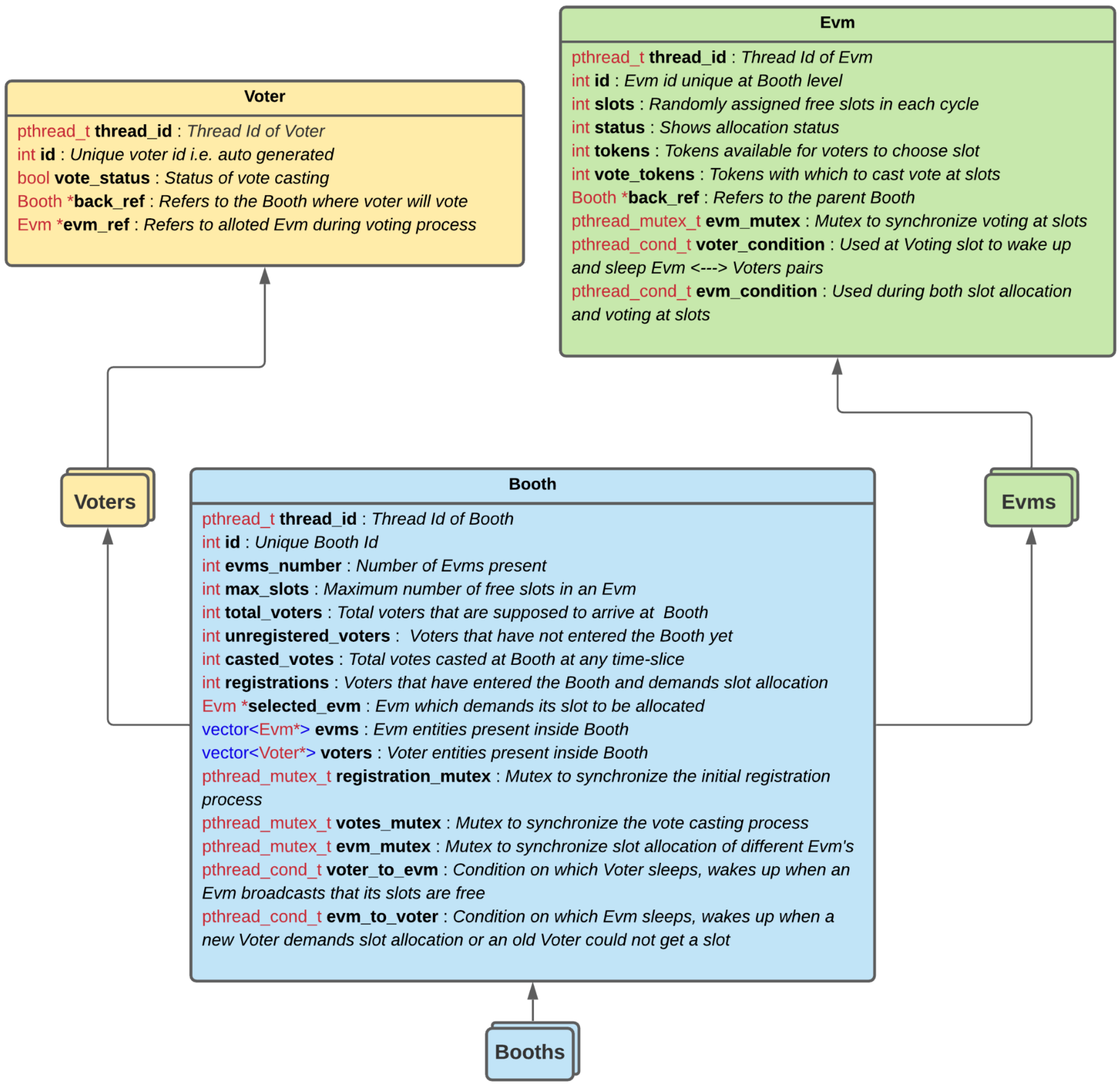
People are fed up with waiting at polling booths on the day of the election. So the government has decided to improve its efficiency by automating the waiting process. From now on, voters will be robots and there will be more than one EVM at each booth. Each of these EVMs can now accept votes from more than one person at a time by having a different number of slots for voting. However one person can only vote once. Each robot and each EVM is controlled by a thread. You have been hired to write synchronization functions that will guarantee orderly use of EVMs. The functions must be built in a way to satisfy the given constraints.

Constraints:

- Using pthread Library to implement threads
- Utilize Mutex and Conditional Variables instead of Semaphores
- Each entity will be represented using Threads: Booth,Evm,Voter
- At a time only one Evm is chosen for its slots to be allocated and voting and slot allocation can run parallely.
- Should not suffer from deadlocks and no busy-waiting solution.
- In each Evm cycle the free slots are decided to be a random value(1<=n<=10)
- Voters will not come in a queue instead multiple voters can come simultaneously.
- Evm calls **polling_ready_evm()** and voter calls **voter_wait_for_evm()** for slot allocation; neither should return until they are allocated voters and slots to cast vote respectively.
- All voters must return from **voter_wait_for_evm()** before they can enter **voter_in_slot()**
- The system should also handle underflow conditions that is when no voters remain for voting, no thread should be in wait state and all resources should be released.

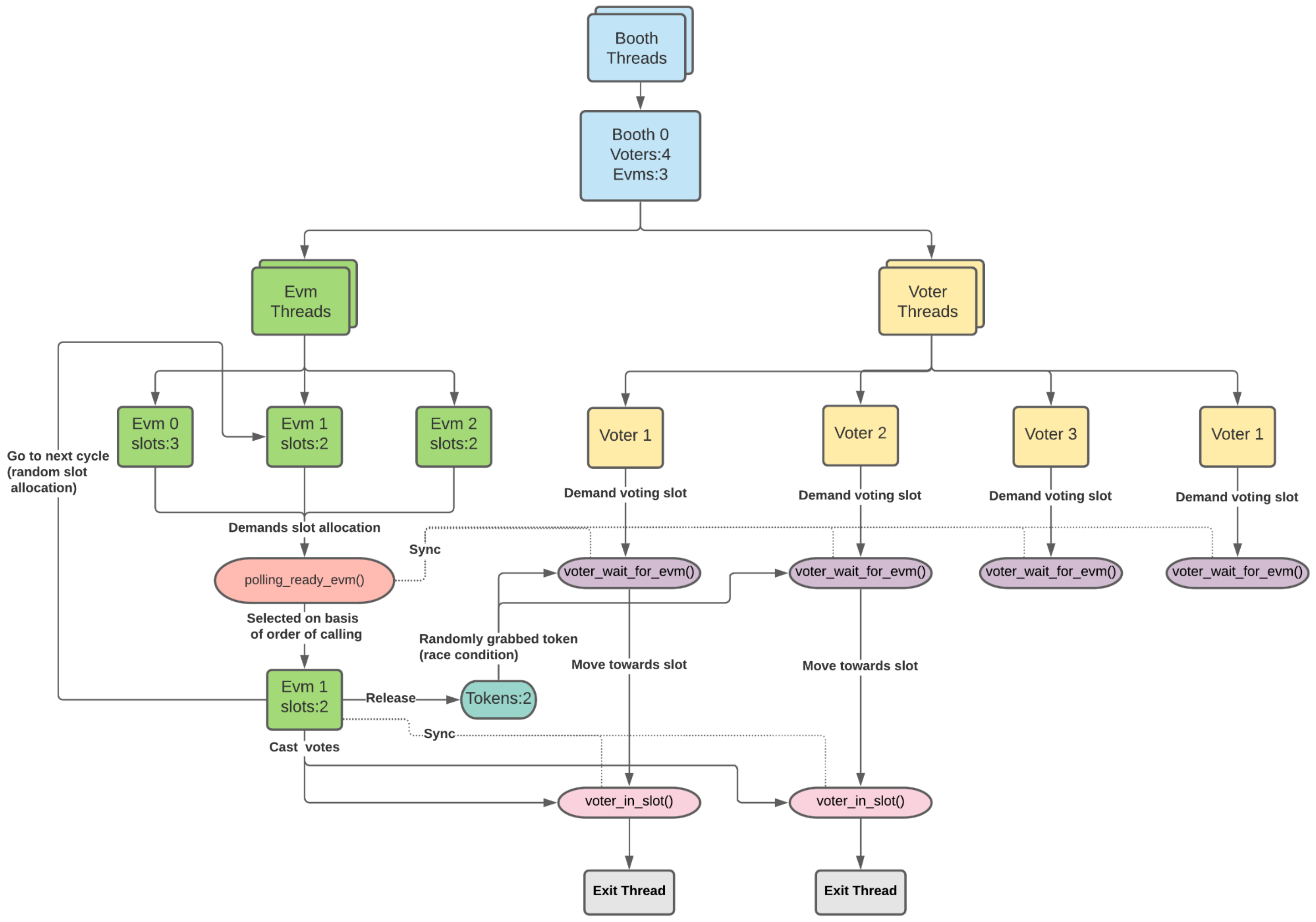
Entity Design:

Our solution revolves around entities created using structures. Each Booth has multiple evms and multiple voters. Each of these are realised using threads. Following lies the parameters that have been used in each entity.



Voter
pthread_t **thread_id** : *Thread Id of Voter*
int **id** : *Unique voter id i.e. auto generated*
bool **vote_status** : *Status of vote casting*
Booth ***back_ref** : *Refers to the Booth where voter will vote*
Evm ***evm_ref** : *Refers to alloted Evm during voting process*
Evm
pthread_t **thread_id** : *Thread Id of Evm*
int **id** : *Evm id unique at Booth level*
int **slots** : *Randomly assigned free slots in each cycle*
int **status** : *Shows allocation status*
int **tokens** : *Tokens available for voters to choose slot*
int **vote_tokens** : *Tokens with which to cast vote at slots*
Booth ***back_ref** : *Refers to the parent Booth*
pthread_mutex_t **evm_mutex** : *Mutex to synchronize voting at slots*
pthread_cond_t **voter_condition** : *Used at Voting slot to wake up and sleep Evm <---> Voters pairs*
pthread_cond_t **evm_condition** : *Used during both slot allocation and voting at slots*
Voters
Evms
Booth
pthread_t **thread_id** : *Thread Id of Booth*
int **id** : *Unique Booth Id*
int **evms_number** : *Number of Evms present*
int **max_slots** : *Maximum number of free slots in an Evm*
int **total_voters** : *Total voters that are supposed to arrive at Booth*
int **unregistered_voters** : *Voters that have not entered the Booth yet*
int **casted_votes** : *Total votes casted at Booth at any time-slice*
int **registrations** : *Voters that have entered the Booth and demands slot allocation*
Evm ***selected_evm** : *Evm which demands its slot to be allocated*
vector<Evm*> **evms** : *Evm entities present inside Booth*
vector<Voter*> **voters** : *Voter entities present inside Booth*
pthread_mutex_t **registration_mutex** : *Mutex to synchronize the initial registration process*
pthread_mutex_t **votes_mutex** : *Mutex to synchronize the vote casting process*
pthread_mutex_t **evm_mutex** : *Mutex to synchronize slot allocation of different Evm's*
pthread_cond_t **voter_to_evm** : *Condition on which Voter sleeps, wakes up when an Evm broadcasts that its slots are free*
pthread_cond_t **evm_to_voter** : *Condition on which Evm sleeps, wakes up when a new Voter demands slot allocation or an old Voter could not get a slot*
Booths

System Design:



The above is a narrow representation of a single Booth in the system and how all the various thread's and functions fit together.

Solution Approach:

- Initialize the different booths with starting parameters like how many evms and how many voters and maximum slots per evm.
- Each Booth thread creates Evm threads and Voter threads.
- Evm's get random free slots in each cycle and all of them try to call **polling_ready_evm()** but whoever manages to call first wins and blocks other evms from getting their slots allocated. So there is a race condition here. We will use a mutex lock here.
- Voter threads are also started parallelly and they call **voter_wait_for_evm()** where they first register themselves using register counter on the Booth

(Registered voter are those who have called the above function and non registered are those haven't)

- **polling_ready_evm()**: Each Evm calls this function, but at a time only one evm can get its slots allocated. If there aren't a sufficient number of registered voters then evm waits on a condition and mutex and waits for a voter to wake it up. Once a voter wakes it up it rechecks the number of registered voters. If a sufficient number of voters arrive then it releases the slot tokens and decrements the registered voters count and wakes up all the voters who were sleeping/waiting on another condition variable and mutex. It then waits until all the tokens are taken. Once slot tokens are allocated evm wakes up and exits the function, and some other blocked evm can now try to allocate themself.
- **voter_wait_for_evm():** Voter after entering the function wakes up an evm to check the number of registered voters and itself waits on a condition variable and mutex. The woken evm may broadcast all voters to wake up if it feels sufficient voters have arrived. When awoken all voters try to snatch the tokens for the slots. So there is a race condition here. If it doesnt find a token then the voter waits again and tries to wake up another evm if present, else the voter takes the token and returns from this function and moves towards the voting slot.The selected evm is woken up and informed that all tokens have been taken.
- All voters wait outside the evm slot until Evm releases the lock to ensure that all voters have returned from **voter_wait_for_evm().**
- Voter reaches **voter_in_slot()** where it again synchronizes with the evm using the evm’s internal conditional variables and mutex.Here evm checks if all voters have cast its vote and waits. On the other hand each voter casts its vote and wakes up evm to check total votes matches the slot tokens that were allotted. There is a two-way wait/wakeup setup here also to ensure in case a voter somehow delays to come to the slot the evm slot function the voting cycle doesn't start without it.
- After all votes are casted voter threads are destroyed and evm moves into the next cycle.
- This process continues until all the non registered voters register themselves and all the registered voters cast their vote.
- The same goes on with other Booths and since each booth is isolated, it will not directly affect other booths.

<u>Results:</u>

Waits in Threads:

- During the Evm slot allocation process there are two conditional variables that are used to synchronize between **voter_wait_for_evm()** and **polling_ready_evm()** ,one is the **evm_to_voter** and the other is **voter_to_evm.** Here each time Evm checks for the number of registered voters and then waits. It is woken each time a new voter arrives and wakes it or an old voter who couldn’t get a slot wakes it up. Another wait is on the voters side where it waits for an evm to call it. The purpose is so that when a sufficient number of registered voters have arrived, all the voters are woken using a broadcast so that each has a chance to get the slot tokens.

- Another wait is required inside **polling_ready_evm()** by evm so that when all tokens are taken by voters it can be informed that the slot tokens have been taken and it can leave the function.
- The next set of waits is required in the Evm thread itself where it synchronizes with the function at the voter end called **voter_in_slot()**. Here evm waits for all the voters to arrive at the voter_in_slot() function. Each voter enters the voter_in_slot() , wakes up evm thread to inform it to check if all voters have arrived at their designated slots and sleeps. Evm after checking that all the voters have arrived broadcasts and wakes up all the voters to cast their votes and keeps on sleeping,waking and checking if all votes have been casted. On voter end it casts vote wakes the Evm and moves out of the function and the voter thread gets destroyed. When all votes have been cast evm moves to next cycle.

Race Conditions:

- During Evm selection whenever multiple evm's try to get allocated at the same time only one wins and others are not allowed until that evm is allocated.
- When slot tokens are released by an EVM , a signal is broadcasted to wake up all the sleeping voters. Here each voter tries to grab the tokens and utilizes a mutex lock here. So only a limited number of voters can get tokens and everyone else has to wait again until some other EVM tries to allocate slots.

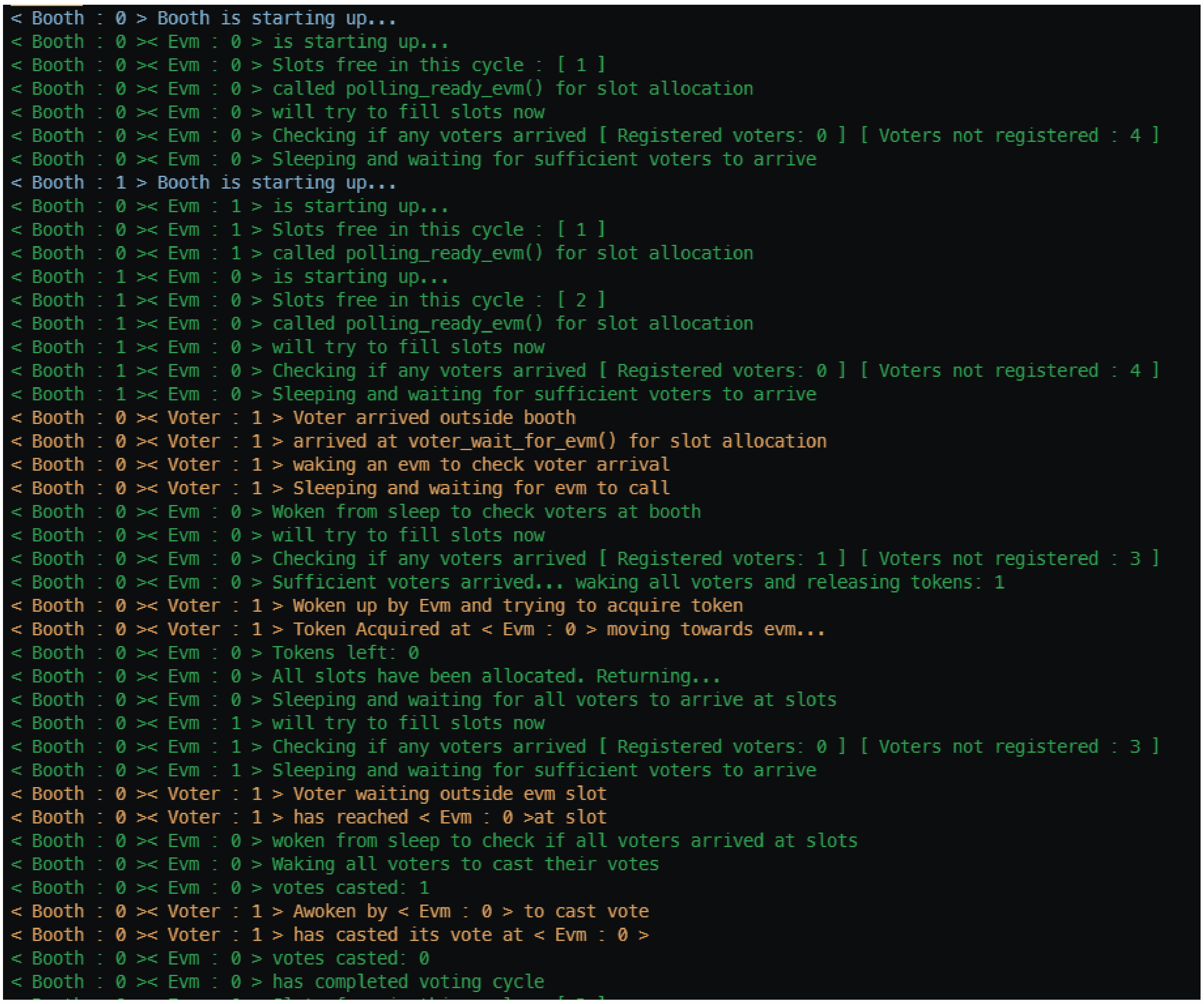
Starvation:

- During Evm selection one particular evm is selected ,the others are blocked. However during unblocking the next evm is selected in the order of arrival, so no evm gets starved from here onwards.
- When the Evm releases slot tokens there might be a case that a particular voter may not be able to grab tokens for many Evm cycles, so it might get starved because in each cycle all voters are woken up to make the process fair.

Deadlocks:

- The implementation does not suffer from deadlocks as everything is properly synchronized using mutexes and conditional variables.

Sample Instance:



```
< Booth : 0 > Booth is starting up...
< Booth : 0 >< Evm : 0 > is starting up...
< Booth : 0 >< Evm : 0 > Slots free in this cycle : [ 1 ]
< Booth : 0 >< Evm : 0 > called polling_ready_evm() for slot allocation
< Booth : 0 >< Evm : 0 > will try to fill slots now
< Booth : 0 >< Evm : 0 > Checking if any voters arrived [ Registered voters: 0 ] [ Voters not registered : 4 ]
< Booth : 0 >< Evm : 0 > Sleeping and waiting for sufficient voters to arrive
< Booth : 1 > Booth is starting up...
< Booth : 0 >< Evm : 1 > is starting up...
< Booth : 0 >< Evm : 1 > Slots free in this cycle : [ 1 ]
< Booth : 0 >< Evm : 1 > called polling_ready_evm() for slot allocation
< Booth : 1 >< Evm : 0 > is starting up...
< Booth : 1 >< Evm : 0 > Slots free in this cycle : [ 2 ]
< Booth : 1 >< Evm : 0 > called polling_ready_evm() for slot allocation
< Booth : 1 >< Evm : 0 > will try to fill slots now
< Booth : 1 >< Evm : 0 > Checking if any voters arrived [ Registered voters: 0 ] [ Voters not registered : 4 ]
< Booth : 1 >< Evm : 0 > Sleeping and waiting for sufficient voters to arrive
< Booth : 0 >< Voter : 1 > Voter arrived outside booth
< Booth : 0 >< Voter : 1 > arrived at voter_wait_for_evm() for slot allocation
< Booth : 0 >< Voter : 1 > waking an evm to check voter arrival
< Booth : 0 >< Voter : 1 > Sleeping and waiting for evm to call
< Booth : 0 >< Evm : 0 > Woken from sleep to check voters at booth
< Booth : 0 >< Evm : 0 > will try to fill slots now
< Booth : 0 >< Evm : 0 > Checking if any voters arrived [ Registered voters: 1 ] [ Voters not registered : 3 ]
< Booth : 0 >< Evm : 0 > Sufficient voters arrived... waking all voters and releasing tokens: 1
< Booth : 0 >< Voter : 1 > Woken up by Evm and trying to acquire token
< Booth : 0 >< Voter : 1 > Token Acquired at < Evm : 0 > moving towards evm...
< Booth : 0 >< Evm : 0 > Tokens left: 0
< Booth : 0 >< Evm : 0 > All slots have been allocated. Returning...
< Booth : 0 >< Evm : 0 > Sleeping and waiting for all voters to arrive at slots
< Booth : 0 >< Evm : 1 > will try to fill slots now
< Booth : 0 >< Evm : 1 > Checking if any voters arrived [ Registered voters: 0 ] [ Voters not registered : 3 ]
< Booth : 0 >< Evm : 1 > Sleeping and waiting for sufficient voters to arrive
< Booth : 0 >< Voter : 1 > Voter waiting outside evm slot
< Booth : 0 >< Voter : 1 > has reached < Evm : 0 >at slot
< Booth : 0 >< Evm : 0 > woken from sleep to check if all voters arrived at slots
< Booth : 0 >< Evm : 0 > Waking all voters to cast their votes
< Booth : 0 >< Evm : 0 > votes casted: 1
< Booth : 0 >< Voter : 1 > Awoken by < Evm : 0 > to cast vote
< Booth : 0 >< Voter : 1 > has casted its vote at < Evm : 0 >
< Booth : 0 >< Evm : 0 > votes casted: 0
< Booth : 0 >< Evm : 0 > has completed voting cycle
```

Problems faced:

- During various test cycles the major issue was deadlock. With conditional variables and mutex there comes the problems of not handling locks and unlocks properly which might lead to this state and not configuring the wait & signal pair properly. It took various testing cycles to perfect the design.

Learning:

- Designing solutions to system problems.
- Multi-Thread creation and intercommunication between Threads.
- Working of Conditional Variables and Mutexes and how to apply them to avoid busy waiting and deadlocks.(using wait and signal/broadcast)
- Resolving deadlocks.

Problem 3 Concurrent Merge Sort

Problem Description:

Sort ‘n’ numbers using merge sort by

a) recursively making two child processes. The parent of the two children then merges the result and returns back to the parent and so on.
b) using threads in place of processes for Concurrent Merge Sort.

The problem mainly focuses on the execution differences arising due to multiprocessing and multithreading.

Introduction:

We have compared the performances of normal merge sort with the merge sort implemented using multiprocessing and multithreading respectively.

The performance is compared with respect to time, number of input elements and conclusion is drawn on the speed for multiple processes and multiple threads.

Programming Platform Used:

Programming Language: C++ Language

Machine Architecture details:

● Architecture:	x86_64
● CPU(s):	2
● Thread(s) per core:	2
● Model name:	AMD Ryzen 5 3600 6-Core Processor

Solution Approach:

→ Multiprocessing:

To achieve concurrent sorting in multiprocessing, we created a shared memory space using ‘shmat’ and ‘shmget’. This memory is filled with ‘n’ random integers where the value of ‘n’ is provided by the user. The memory is accessed by the child processes created by fork calls. Two child processes are created recursively, one for the left half, and the other one for the right half. Each segment is split into left and right child which is sorted concurrently. When the number of elements is less than 5, selection sort is performed. The parent waits for the child processes to complete and then merges the result of the two children and returns back to the parent and so on. The shared memory is released using ‘shmdt’ and ‘shmctl’.

Also, a normal merge sort is performed with the same set of numbers to compare the execution timings.

→ Multithreading:

For multithreading, we have used the pthread library to create multiple threads.

We have created two different ways for executing multithreading.One way is to create two threads for every subarray in recursive merge sort and another way is by dividing the array into some user given number of threads.

For the first method, a thread is created which calls the ‘merge_sort’ function. For every left and right sub array, a new thread is created which recursively calls merge sort function and then pthread_join is called, from the thread that created it, to wait for the thread to terminate. Then their sorted results were merged by calling the ‘merging’ function. Execution time was recorded before and after Sorting.

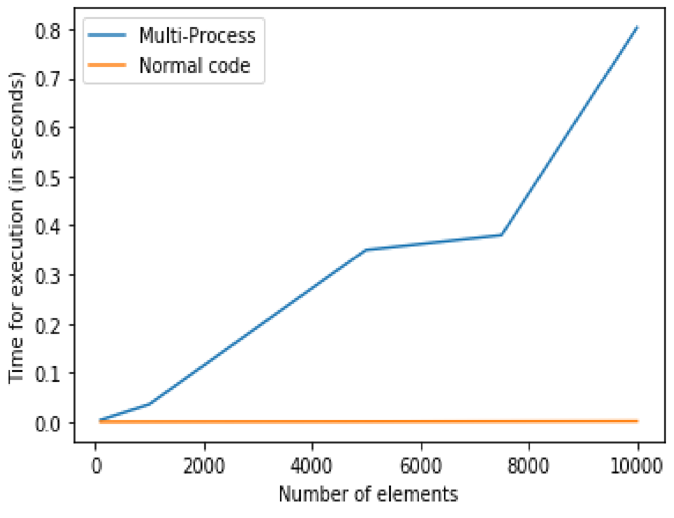
In the second method, first the given number of arrays are created. The array elements are divided into the given number of threads.So each thread performs sorting on its own subarray using ‘merge_sort’ function. Then pthread_join is called, to wait for all the threads to terminate. After that,every sorted subarray of each thread is merged. Execution Time was recorded before and after Sorting.

Results:

We tested out our codes on the following data and the results are tabulated as follows:

→ Multiprocess:

No. of Elements	Time for conventional merge sort (seconds)	Time for merge sort with multiprocessing (seconds)
100	0.000009	0.003760
1000	0.000110	0.035690
5000	0.000607	0.349950
7500	0.000980	0.380361
10000	0.001422	0.803144

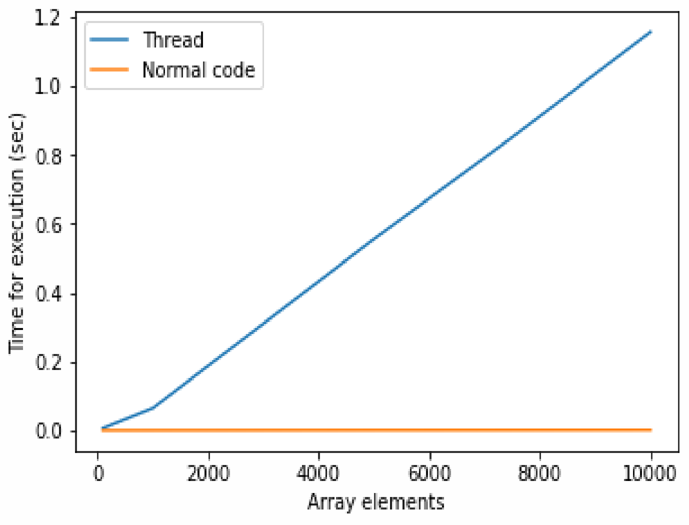


(i) Multiprocess merge sort vs normal merge sort

→ Multithreading:

i) Creating threads every time for sorting each subarray

No. of Elements	Time for conventional merge sort (seconds)	Time for merge sort with multithreading (seconds)
100	0.000007	0.006728
1000	0.00011	0.064532
5000	0.000663	0.554968
7500	0.000832	0.850002
10000	0.001161	1.1559

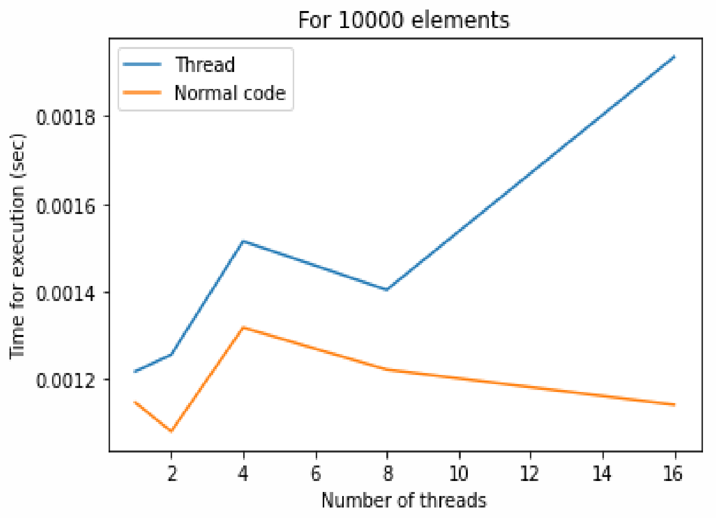


(ii) Creating threads every time for sorting each subarray

ii) Fixed number of threads

a) For 10000 elements

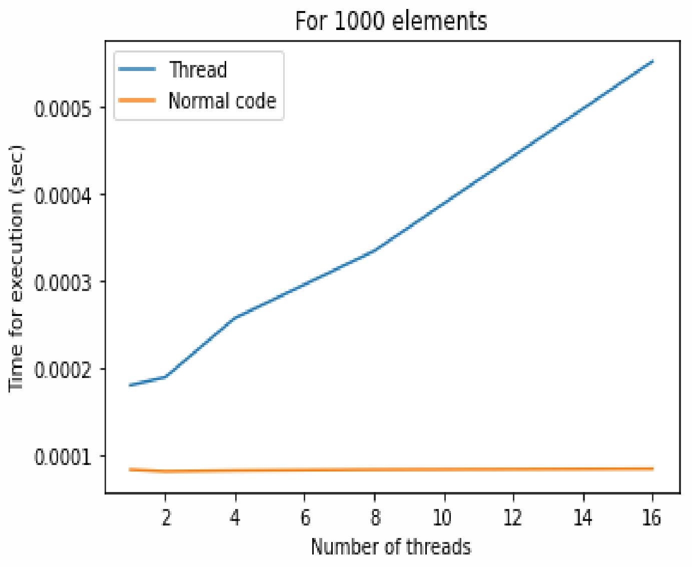
No. of Threads	Time for conventional merge sort (seconds)	Time for merge sort with threads (seconds)
1	0.001147	0.001218
2	0.001081	0.001256
4	0.001318	0.001515
8	0.001222	0.001404
16	0.001142	0.001935



(iii) Creating fixed number of threads for sorting 10000 elements

b) For 1000 elements:

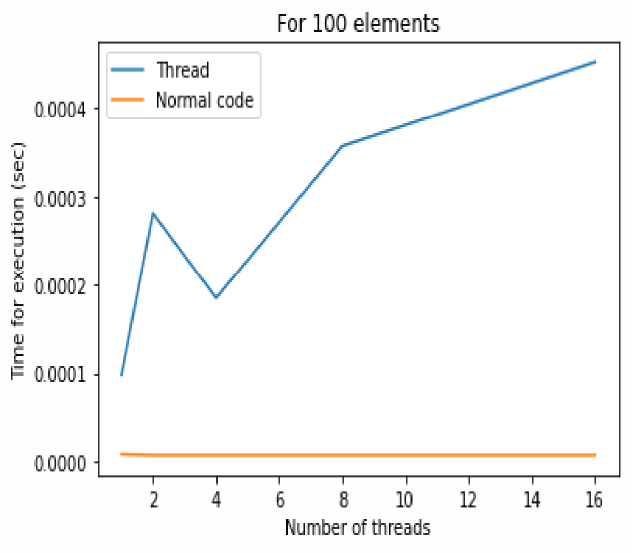
No. of Threads	Time for conventional merge sort (seconds)	Time for merge sort with threads (seconds)
1	0.000083	0.00018
2	0.000081	0.000189
4	0.000082	0.000257
8	0.000083	0.000334
16	0.000084	0.000552



(iv) Creating fixed number of threads for sorting 1000 elements

c) For 100 elements:

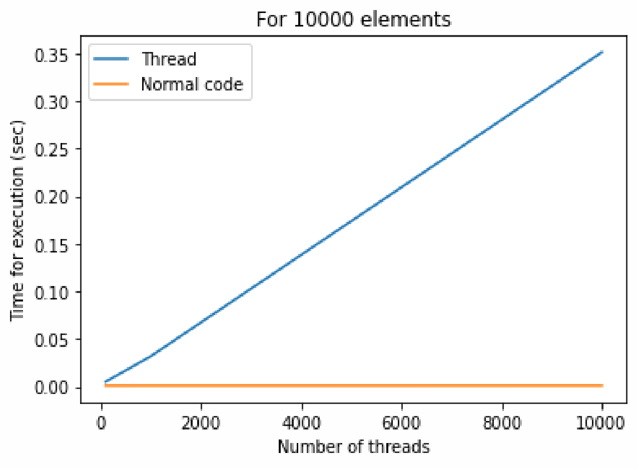
No. of Threads	Time for conventional merge sort (seconds)	Time for merge sort with threads (seconds)
1	0.000008	0.000098
2	0.000007	0.000281
4	0.000007	0.000185
8	0.000007	0.000357
16	0.000007	0.000452



(v) Creating fixed number of threads for sorting 100 elements

iii) For large number of threads :

Number of Threads	Time for conventional merge sort (seconds)	Time for merge sort with threads (seconds)
100	0.005032	0.00112
1000	0.031692	0.001131
10000	0.351029	0.001164



(vi) Creating large no. of threads for sorting 10000 elements

Learning and Conclusion:

The conclusion which can be drawn from the graphs is that for given inputs, the normal merge sort is working faster than multiprocessing merge sort and multithreading merge sort.

For a small number of threads, execution time of multithreading merge sort is comparable to normal merge sort. When a thread is created for each subarray, the execution time of multithreading merge sort is comparable to multiprocess merge sort.

The slow execution of **multiprocess** merge sort is due to the cache misses. A cache miss can occur when the left child is accessing the left subarray and the right child tries to access the right subarray. This happens because initially the left subarray is loaded into the cache of a processor, and when the right subarray is accessed, there is a cache miss and then the right subarray is copied to the cache memory. These cache misses degrade the performance.

For **Multithreading**, we observe that when we are creating threads every time for sorting each subarray, execution time is increasing a lot as compared to normal merge sort (graph (ii)). This is because of the overhead of creating threads and their executions. As a large number of threads are being created they are getting stalled for the chance for their execution which results in increased execution time as compared to normal merge sort.

The overhead from creating all those threads far outweighs the gains we get from parallel work. This is further confirmed by execution of the program with a fixed number of threads (graph (iii),(iv),(v)).

Finally we observed that as we increase the number of threads to a large number for 10000 elements (graph (vi)), execution time is getting close to what we got in graph (ii) for 10000 elements as a large number of threads are being created each of which then only does very little work.

Also we have tested the code on a machine having a small number of cores, so execution time for normal merge sort and threads are close when number of threads are close to number of cores.

So from a certain number of threads, the work is not really parallelized, because the processor does not have enough cores to execute this work at the same time, however the overload caused by multithreaded does penalize the performance.

Problems Faced and Shortcomings:

- For both multiprocessing and multithreading merge sorts, having a system with good specifications could have resulted in more accurate results.