Problems on Process Co-ordination

The goal of this project is to solve some problems involving process co-ordination, in C using pthreads and semaphores. Grading of this project will be based on two criteria, correctness and presentation of code, and report that you will submit.

For each problem, you have to do the following in your report:

1. Identify the correctness constraints for each problem
2. Specify the conditions where wait should happen and justify your reasons for them.
3. Also specify, when your algorithm can lead to starvation, deadlock or race condition etc, and if not why
4. Implement the solution using semaphores only.

Problem 1 Badminton Academy Problem

Gopichand academy recently started allowing a **group of 3 people** to play badminton at their courts. The group should consist of **2 players** who play against each other 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 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.

- Players invoke the following functions in order: **enterAcademy, meetOrganizer, enterCourt, warmUp**
- Referees invoke the following functions in order: **enterAcademy, meetOrganizer, enterCourt, adjustEquipment, startGame**

Write code that enforces the above with appropriate **print statements** for each of the above actions.

Input: an integer **n** indicating **2*n players** and **n referees** will arrive at the academy in total. Use sleep(1) for warmUp and sleep(0.5) for adjustEquipment actions. Make the new person arrive with a **random, modulo 3** second delay and probability that he is a player/referee depends on the remaining number of players/referees that will be coming and each person is a thread.

Problem 2 The Queue at the Polling Booth

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 vote from **more than one person** at a time by having 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. You must define a structure struct booth, plus several functions described below.

- When an EVM is free and is ready to be used, it invokes the function **polling_ready_evm(struct booth *booth, int count)** where count indicates how many slots are **available to vote at this instant** (by this we mean that the count is not fixed every time we call a specific EVM. There is a possibility that a particular EVM invokes this once with count = 5 and later with count = 7. Use a random function to determine the count value from a fixed range **(1<=count<=10))** The function must not return until the EVM is satisfactorily allocated (all voters are in their slot, and either the EVM is full or all waiting voters have been assigned – note that a voter doesn't wait for a particular EVM but he only waits to vote. So it will return if there is no voter that is waiting at the booth to vote)
- When a voter robot arrives at the booth, it first invokes the function **voter_wait_for_evm(struct booth *booth)** This function must not return until an EVM is available in the booth (i.e. a call to the **polling_ready_evm** is in progress) and there are enough free slots on the EVM for this voter to vote (one slot for one voter) Once this function returns, the voter robot will move the voter to the assigned slot (do not worry about how this works!)
- Once the voter enters the slot, he/she will call the function **voter_in_slot(struct booth *booth)** to let the EVM know that they reached the slot.
- Assume that there are a fixed number of voters per booth (no voters will come after this process starts), a fixed number of EVMs per booth and a fixed number of booths (will be given as inputs to the program)
- Stop this simulation when all the voters are done with voting. Note that an EVM can actually be used more than once for voting So basically when there are still voters left all EVM's should invoke **polling_ready_evm()**.
- You may assume that there is never more than one EVM in the booth available free at once (have a look at the sample output) and that any voter can vote in any EVM. Any number of EVM's can call **polling_ready_evm()**. But only one will be ready to vote at a time.
- Create a file that contains a declaration for **struct booth** and defines the **three** functions described above and the function **booth_init** which will be invoked to initialize the booth object when the system boots. Use appropriate small delays (sleep()) for the simulation.

- **No semaphores** are to be used. You should not use more than one lock in each **struct booth**. Use mutexes and conditional variables to do this question.
- Your code must allow multiple voters to board simultaneously (it must be possible for several voters to have called **voter_wait_for_evm**, and for that function to have returned for each of the voters, before any of the voters calls **voter_in_slot**)
- Your code must not result in busy-waiting (deadlocks)
- You can define more functions if needed. For the functions mentioned in the problem statement, you may extend them by adding new arguments for them but don't decrease them.

Kindly note that this problem's goal is make you realize the use of threads and synchronization techniques.Output doesn't matter. It only depends how you simulate. Make reasonable assumptions so that your solution doesn't deviate a lot from the problem.

Problem 3 Concurrent Merge Sort

- Given a number **n** and n numbers, sort the numbers using Merge Sort.
- Recursively make two child **processes**, one for the left half, one for the right half. If the number of elements in the array for a process is less than 5, perform a selection sort.
- The parent of the two children then merges the result and returns back to the parent and so on.
- Compare the performance of the merge sort with a normal merge sort implementation and make a **report**.
- You must use the **shmget, shmat** functions as taught in the tutorial.

Bonus:

- Use threads in place of processes for Concurrent Merge Sort. Add the performance comparison to the above report.

Precautions:

- Do not Copy, otherwise, you'll get a F
- You have to consider details of the algorithm yourself
- You are also entitled to assume certain viewpoints about your problem, which you have to write in your report
- Your report should be in PDF format. Be concise and do not explain unnecessarily.

Guidelines:

➔ Make sure you write a readme which briefly describes the implementation for Problems 1 and 2 (design idea) and tells how to run the code for each of the questions.