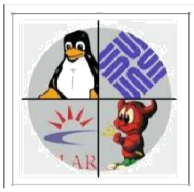

Assignment 4

Problems on Concurrency

Deadline: **October 6th, Friday, 9pm.**

This assignment is supposed to be done individually.

The goal of this assignment is to better understand the concepts of concurrency which has been taught in the class.

Problem 1: The Gas Station Problem

We have a gas station with **three gas pumps**, **three** smartly dressed **attenders** and **a waiting lane** that can accomodate **four cars**. This is a premium gas station and their clientele consists only of car owners. Being located in a busy area, space is at a premium and is available only for seven (3 + 4) cars inside the gas station. A car will not enter the gas station if it is filled to the capacity with cars. Once inside, the car enters the waiting lane (a "queue") When an attender (attached to a gas pump) is free, the car at the head of the queue drives up to that gas pump and gets served. When a car's filling is done, **any** attender can accept the payment, but because there is only one ATM machine, payment is accepted for one car customer at a time (note that attenders are needed for this) The attenders divide their time among serving cars, accepting payment and dreaming about having their own cars whilst waiting for a car to serve or a payment to accept.

- The Cars invoke the following functions in order: **enterStation, waitInLine, goToPump, pay, exitStation**.
- Attenders invoke **serveCar** and **acceptPayment**.
- Cars cannot invoke **enterStation** if the gas station is at capacity.
- If all the three pumps are busy, a car in the wait lane cannot invoke **goToPump** until one of the cars being served invokes **pay**.
- The car customer has to **pay** before the attender can **acceptPayment**.
- The attender must **acceptPayment** before the car can **exitStation**.

Write code that enforces the above with appropriate **print** statements for each of the above seven actions (and also when a car is unable to **enterStation**)
Input: number of cars which will arrive **n**. Use sleep(1) (1 second) for **serveCar** and sleep(0.5) for **acceptPayment**. Make the cars arrive with a random, modulo 3 second delay and each is a thread. Each attender 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 Problem 3. Add the performance comparison to the above report.

Submission Guidelines

- Upload format **rollnum_assgn4.tar.gz**.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.
- **Do not copy**. The emphasis is on how you approach the situation. If your code works but you're unable to explain it, you may end up with neglible marks and not so negligible regret.

OSTAs,ICS231