Indian Institute of Technology Guwahati
Department of Computer Science & Engineering

CS 588: Systems Lab
Assignment – 1

Instructions:

- Make sure that you read, understand, and follow these instructions carefully. Your cooperation will help to speed up the grading process.
- Following are generic instructions. Make sure that you also check carefully and follow any specific instructions associated with particular questions.
- In this assignment, you will explore the dynamics and challenges of process management in operating systems.
- Complete the assignment with the best of your own capabilities. Create a single zipped/compressed file containing all your programs and related files.
- The compressed file must contain:
  - All your program files
  - Makefile to compile your programs
  - Readme file to explain your program flow (individual)
  - No output files
- The file name should be as your roll no. Example: 174101001.zip. When done, submit the file in Moodle on or before the submission deadline.
- **Submission deadline: 11:55 PM, Saturday 10th February, 2018 (Hard Deadline).**

Ethical Guidelines (lab policy):

- Deadlines: Deadlines should be met. Assignments submitted after their respective deadlines will not be considered for evaluation.
- Cheating: You are expected to complete the assignment by yourself. Cases of unfair means and copying other's solution will not be tolerated, even if you make cosmetic changes to them. If we suspect any form of cheating, we are compelled to award ZERO marks.
- If you have problem meeting a deadline, it is suggested that you consult the instructor and not cheat.

Question-1:

Using **dd** command create files of varying size (1KiB - 1GiB) and store them. Flush the cache. Now read these files one after another using C. It generally means that the files will be brought from secondary memory to main memory, hence will experience page faults. Now, calculate the number of page faults occurred during each file read.

Plot and analyse how number of Page_Faults varies with File_Size variation.

Question-2: Simulation of Linux commands.

Create your own shell called **my_shell** which will read input lines from standard input, will parse them into a command name and arguments (if any), and run that command by starting a new process. Your **my_shell,** when requesting input should exactly look like that in linux.

i.e. user@hostname:path ----> abhijit@abhijitcse:~/Tutorial$

Write a C program to simulate your bash shell **my_shell** with commands **cd**, **mkdir**, **cat**, **top**, and **redirection(>, >>)**

* The commands are to be implemented explicitly using C and should be executed via **my_shell** process using exec system call. Use of already existing implementation of these commands in bash or any other shell is strictly prohibited.

* **my_shell** should show meaningful errors for any illegal command, unrecognised argument and argument count (exceeding 10). It (**my_shell)** should not get crashed or be killed.

* **my_shell** should also show correct usage of any command. Hence implement **help** command for your commands (**cd**, **mkdir**, **cat**, **top**, and **redirection(>, >>)**).

* **my_shell** should also remember the recently used commands. Hence implement **history** command for your shell. History function need not be explicitly implemented, but when using **UP** and **DOWN** arrows, the recently used commands must be available.

* **my_shell** should also support normal termination. Hence implement **exit** command, which causes **my_shell** to exit after printing an exit message.

* **my_shell** should be developed in a way where new commands / features can be augmented easily.

** **Hint:** Use GNU readline and history library.

Question-3: Airline Reservation System:

Queries available:

TYPE 1: Inquiry: Number of seats available in each flight

TYPE 2: Book: Book **n** tickets in a flight

TYPE 3: Cancel: Cancel a booked ticket

Parameters / Restrictions:

f=	Number of flights=	10
c=	Capacity of each flight=	150
n=	Number of tickets booked once=	2 - 5
MAX=	Maximum number of active queries at any given time=	5

[fork] The main process will create **f** flights and initialise them with **c** seats. This main process will create a child process to take care of the users. The main process then becomes a **daemon** and is responsible for the whole reservation portal.

[pthread] The child process then creates **t** threads (users) which run concurrently to execute various queries. The queries will be initiated based on a Discrete Event System (DES) log from an input file (DES.csv) with the following format.

10,Inquiry,1,1 //Query_Time, Query_Type, Flight_no, Thread_no

10,Book,1,2 //Query_Time, Query_Type, Flight_no, Thread_no

12,Cancel,5,3 //Query_Time, Query_Type, Flight_no, Thread_no

12,Book,5,7 //Query_Time, Query_Type, Flight_no, Thread_no

.

.

Where, Query_Time = {1, 2, 3, . . .}

Query_Type = {Inquiry, Book Cancel}

Flight_no = {1, 2, 3, . . . 10}

Thread_no = {1, 2, 3, . . . 20}

At most **MAX** queries / threads can be active / serviced at any given time. So, any new query (MAX+1) must wait until an active query finishes and make room. This restriction is to be implemented using appropriate condition variables.

Parameters / Restrictions:

t=	Number of working threads=	20
TYPE 1:	Inquiry:	**read**
TYPE 2:	Book:	**write**
TYPE 3:	Cancel:	**write**
T=	Total running time=	1 - 5 minutes

[Mutual Exclusion] A write query (**TYPE 2, TYPE 3**) needs careful implementation. Two or more concurrent write queries are not allowed on the same flight. Moreover, concurrent read and write queries are also not allowed on the same flight. However, concurrent read queries on same flight are allowed. Similarly, concurrent write queries on different flights are also allowed. Implementation using **mutex** (one for each flight) may not be practical if the number of flights are more.

[Shared Memory] One practical implementation can be the maintenance of a shared table with **MAX** entries. An entry in the shared table will be a triplet
(Query_Type, Flight_no, Thread_no)

A **read** query can proceed if there are no **write** queries for the same flight in the table. Similarly, a **write** query can proceed if there are no **read/write** queries for the same flight in the table. This restriction is to be implemented using appropriate synchronisation primitive.

Each working thread will maintain its private list of booking, which should be available for display upon request. Additionally, each thread will also print appropriate control messages (thread_no, query_input, query_output, wait_time, signal_time, time_out, etc.)

[Overall Flow] Main process creates the system with flights and their initial seat capacity. The main process then creates a child process to take care of users and works as a daemon in the background. The main process is also responsible for the overall reservation portal. The child process create multiple users in terms of threads to run concurrent queries. Every thread (query) goes to the parent process via the child process only (not directly). The parent process then takes appropriate actions in the shared table and returns appropriate control and output messages to the threads via the child.

After running for a stipulated time **T**, the reservation server needs to be shut down for maintenance (like real servers). This is when all the working threads need to be closed one by one. After all closed, the parent process displays the current status of the reservation system.

Flight Number	Booked Seats	Available Seats
1	50	100
.	.	.

Write a C program to implement the entire reservation system.