CS F372 - OPERATING SYSTEMS

SEM II 2023-2024

ASSIGNMENT 1: HOTEL MANAGEMENT SYSTEM

MAXIMUM MARKS: 30

HARD DEADLINE: 6 AM, 26th FEBRUARY 2024

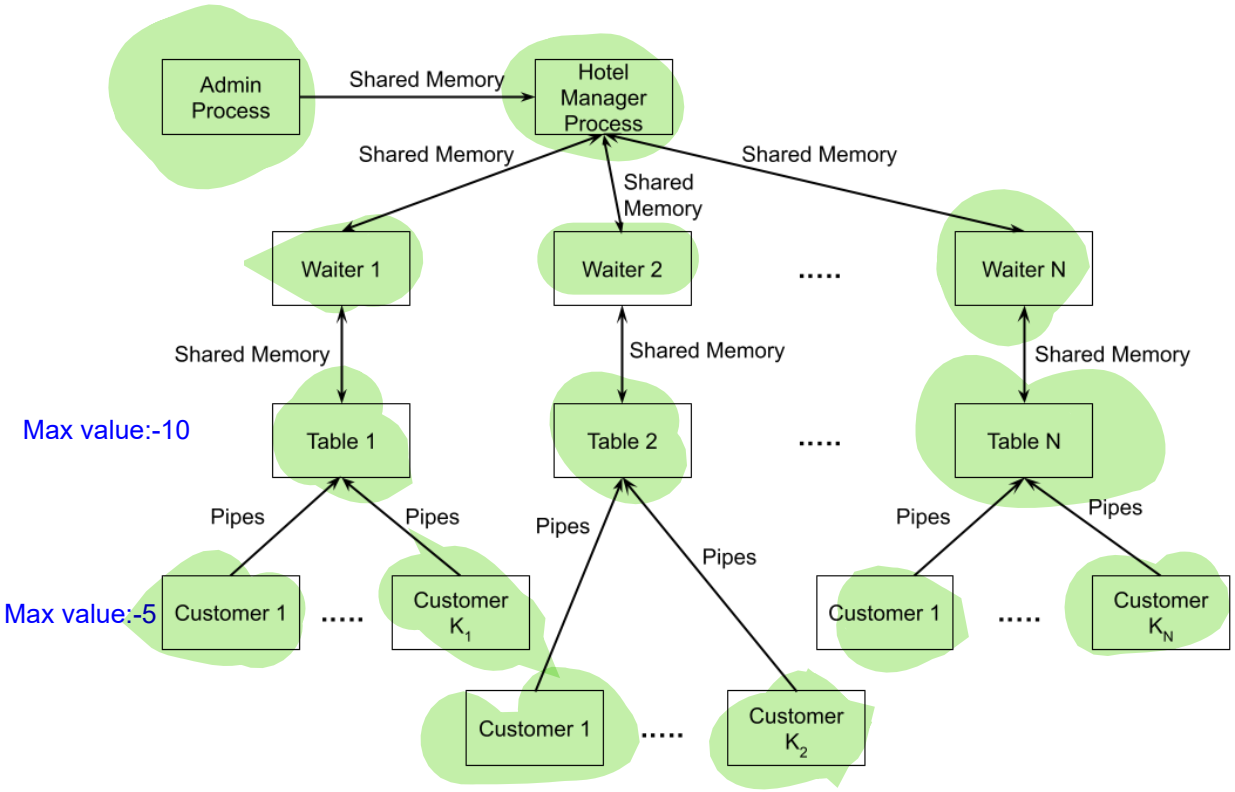
Note: This document contains the problem statement, assignment constraints, submission guidelines, plagiarism policy and demo guidelines. Please go through the entire document and not just the problem statement.

Problem Statement:

In this assignment, we are going to use the different OS concepts learnt such as Linux commands, process creation, inter-process communication, `wait()`, `sleep()`, etc. to create a hotel management system application. The problem statement of the assignment consists of the following parts.

System Overview:

The system simulates a hotel environment where the different entities like an Admin, a Hotel Manager, Tables, Customers and Waiters are represented as processes. There are table processes (maximum number of concurrent table processes will be 10) in this system such that each table process represents a table of this hotel. As we all know, customers enter a hotel, sit at the different tables and order food items from a predefined menu. We will simulate this using processes as well. Each table process will create one or more child processes, each child process representing a customer sitting at that particular table. We will call each such child process as a customer process. Note that each table process needs to create at least one customer process and can create a maximum of five concurrent customer processes at a time. In this hotel system, we will also have waiter processes. Each waiter process takes orders for a particular table and calculates the total bill amount for that table. Each table has a Number and each waiter has a Waiter ID. A waiter having ID *x* will only attend table number *x*. Note that the number of table processes will always be equal to the number of waiter processes. Moreover, the table processes and the waiter processes do not have any parent-child relationship. This implies that during execution, you should run each table process in a different terminal and each waiter process should run in a separate terminal. Additionally, there is a hotel manager process responsible for overseeing the total earnings, calculating the total earnings of all the waiters and handling termination, upon receiving a termination intimation from the admin process. The overall workflow is depicted in the diagram below. Note that the arrows in the figure depict communications between the relevant processes. In the rest of the problem statement, the term user is used to refer to a person who will be executing this application.



1. Write a POSIX-compliant C program called **table.c**:
   - **a.** On execution, each instance of this program creates a separate table process, i.e., if the executable file corresponding to table.c is table.out, then each time table.out is executed on a separate terminal, a separate table process is created indicating a new table along with customers (customer process creation is mentioned later) who have entered the hotel and are sitting at that particular table. For every instance of table.out executed on the terminal, the same source code should be used. Multiple source files should not be created for different instances of table processes.
   - **b.** When a table process is run, first, it will ask the user to enter a positive integer as its unique table number. The prompt message should be as follows:

     ```
     Enter Table Number:
     ```

     Out of the total table processes being concurrently executed, it is guaranteed that the table number will be sequentially assigned by the user, will be monotonically increasing, will be a positive integer lying in the range 1 to 10 (1 and 10 inclusive) and will be unique for each table. You can assume that the user always gives a valid input for the table number.

   - **c.** Next, the table process requests to know the number of customers sitting at that table by asking the user to enter a positive integer. The prompt message should be as follows:

     ```
     Enter Number of Customers at Table (maximum no. of customers can be 5):
     ```

     For each table process being concurrently executed, this value will be input by the user and will be within the bounds from 1 to 5, both 1 and 5 inclusive. Each customer process will be a child of that table process. Thus, for each customer sitting at the table, the table process will create a child process and a pipe for IPC between the table process and the corresponding customer. It is to be noted that a separate pipe is used

table.c
hotelmanager.c
admin.c
waiter.c

for communication between a table process and each one of its customer processes. Thus, if there are four customers at a table, the table process will create four pipes. Note that you will be creating POSIX compliant ordinary pipes here.

d. The table process will read the menu from the **menu.txt** file which is a pre-created file and will have the format as below.

```
1. Veg Burger 30 INR
2. Chicken Burger 40 INR
3. Ostrich Eggs 25 INR
4. Egg Frankie 30 INR
```

This menu will be displayed to the user on the terminal in the above format after the user has entered the number of customers at that table. Assume that the file menu.txt is present in the same directory as your assignment source files and is already created.

e. Each customer process will take the input from the user regarding the food item(s) s/he wants to order using the prompt message below.

```
Enter the serial number(s) of the item(s) to order from the menu. Enter -1 when done:
```

The user will enter the corresponding integer value(s) for each customer process. A customer process is allowed to order multiple food items as well as multiple instances of a single food item (like two veg burgers). For each customer process, the end of order is represented with an input of -1. A customer process may not order anything at all. However, some input(s) has/have to be provided for every customer. All the customer processes will communicate the order items to the corresponding table process using the pipes mentioned in 1.(c). Note that a customer process may try to order (by mistake) an item which is not present in the menu. Handling of this scenario is explained later.

f. Only after gathering the orders from all the customers at the table, the table process communicates the entire order to the waiter process assigned for this table using shared memory. Note that a separate shared memory segment is used for communication between a distinct table process-waiter process pair. Thus, if there are four table processes, there will be four waiter processes and four corresponding shared memory segments.

g. Subsequently, the waiter process will communicate the total bill amount for the order (a valid one) placed to the table process via the shared memory segment. On receiving the total bill amount from the waiter via shared memory (bill calculation is explained later), the table process displays the bill amount in the following format.

```
The total bill amount is X INR.
```

Once the bill amount has been displayed, it implies that the current set of customers leave the hotel. You are free to handle the termination of the customer processes in an appropriate manner.

h. After this, the table process asks the user if s/he wishes to seat a new set of customers at the table (since the previous set has left) by re-displaying the message in 1.(c). If the value entered by the user is -1, the table process understands that no more customers will be sitting at this table and informs the corresponding waiter process

to terminate. After this communication, the table process terminates as well. If the value entered by the user is between 1 and 5 (both values inclusive), resume normal execution from part 1.(c) onwards. Note that it is guaranteed that the user never enters zero as the number of customers for a table.

i. It is implicit that a table process is never terminated as long as there are customers sitting at the table.

j. You may use your own logic regarding the termination of each customer process. However, note that in case any customer from a given set of customers sitting at a table places an invalid order (by mentioning one or more non-existent food items), all the customers will be asked to place the entire order again.

2. Write a POSIX-compliant C program called **waiter.c**:

a. On execution, each instance of this program creates a separate waiter process, i.e., if the executable file corresponding to waiter.c is waiter.out, then each time waiter.out is executed on a separate terminal, a separate waiter process is created representing a new waiter for the corresponding table. For every instance of the waiter.out executed on the terminal, the same source code should be used. Multiple source files should not be created for different instances of waiter processes.

b. When a waiter process is run, first, it will ask the user to enter a positive integer as its unique Waiter ID. The prompt message should be as follows:

```
Enter Waiter ID:
```

Out of the total waiter processes being concurrently executed, the Waiter ID will be sequentially assigned by the user, will be monotonically increasing, will be a positive integer lying in the range 1 to 10 (1 and 10 inclusive) and will be unique for each waiter. You can assume that the user always gives a valid input for waiter ID, i.e., numbers in the range 1 to 10. There will always be a one-to-one mapping between the number of waiters and the number of tables. A waiter having ID X will only attend table having number X. Note that the number of waiters is exactly equal to the number of tables.

c. Each waiter process receives the order from its corresponding table process using a shared memory segment. Note that for each table-waiter pair, a separate shared memory segment is to be used. These segments have been mentioned in 1.(f) as well.

d. Each waiter process checks whether the order is a valid order i.e., the serial number(s) of all the item(s) in the order exist in the menu as well. If not, the waiter process communicates the same to the corresponding table process via the shared memory segment. The table process then has to retake the entire order (across all customers for that table) by showing the prompt message in 1.(e) again.

e. Once the correct order is received by the waiter, s/he calculates the total bill amount of the table, displays it to the user on the terminal as per the format given below and communicates the amount to the hotel manager process as well as the table process using two different shared memory segments.

```
Bill Amount for Table X: 60 INR
```

Note that the shared memory segment between the waiter and the table is the same that has been mentioned above in 1.(f). Moreover, each waiter process communicates

the bill amount to the hotel manager process using a unique shared memory segment.

f. Each waiter process will terminate when the corresponding table process informs it regarding the termination as mentioned in 1.(h) via the shared memory segment.

g. It is implicit that a waiter process is never terminated as long as there are customers sitting at the corresponding table.

3. Write a POSIX-compliant C program called **hotelmanager.c**:

a. Only one instance of this program is executed in the entire application.

b. The hotel manager receives the total number of tables at the restaurant from the user by displaying the below prompt message.

```
Enter the Total Number of Tables at the Hotel:
```

Note that this number is guaranteed to be equal to the number of instances of table (as well as waiter) processes to be executed. If the user enters X as the total number of tables, exactly X tables and X waiters should be created. However, not all X instances of tables and waiters are guaranteed to be executed in the beginning itself, i.e., initially, Y out of the X instances may be executed and after sometime, the remaining X - Y instances can be executed.

c. The hotel manager process receives the earnings for a particular table from the corresponding waiter for a single set of customers using shared memory and writes the same into an output file called **earnings.txt** in the format below.

```
Earning from Table X: 60 INR
Earning from Table Y: 50 INR
```

Thus, the file **earnings.txt** will contain the earnings from different tables of the hotel in separate lines. Note that there should be only one file recording all the earnings. Also, with each waiter process, the hotel manager process communicates using a separate shared memory segment (as mentioned in 2.(e)). Each table will have an earning for each set of customers.

d. On receiving intimation of termination (discussed below) from admin process and only when there are no customers at the hotel sitting at any table, the hotel manager calculates the total earnings of the hotel (sum of the individual earnings), the total wages of all the waiters which is 40% of the total earnings of the hotel and the profit made by the hotel which is the difference between the total earnings and total wages. These values are displayed to the user in the below format.

```
Total Earnings of Hotel: 500 INR
Total Wages of Waiters: 200 INR
Total Profit: 300 INR
```

These values should also be added in the same format to the output file **earnings.txt**, i.e., first, all individual earnings should be listed and then total earnings, total wages and total profit should be recorded.

e. Before the hotel manager process exits, it should display the terminating message as

below (after displaying the earnings, wages and profit message).

`Thank you for visiting the Hotel!`

f. You need to figure out your own logic regarding how the hotel manager process will determine that there are no customers left at the hotel.

4. Write a POSIX-compliant C program **admin.c**.

a. This admin process will keep running along with the table, customer, waiter and hotel manager processes and only one instance of this program is executed. This process will keep displaying a message as:

`Do you want to close the hotel? Enter Y for Yes and N for No.`

b. If N is given as input, the admin process keeps running as usual and will not communicate with any other process. If Y is given as input, the admin process will inform the hotel manager process using shared memory that the hotel needs to close. After passing the termination information to the hotel manager, the admin process will terminate. When the hotel manager process receives the termination request from the admin process, it begins the termination task only when there are no more customers at the hotel. The termination task involves 3.(c), 3.(d), cleaning up all the IPC constructs, terminating all the processes and other cleanup tasks as required.

5. There is no parent-child relationship between table, waiter, hotel manager and admin processes.

6. Use `wait()` and `sleep()` appropriately as required. Perform all relevant error handling for all system calls properly without which marks will be deducted.

7. Note the .txt files that are mentioned in 1 and 3 are simple ASCII (text only) files.

8. You can assume that all the C files as well as the .txt files are present in the same directory.

9. Only the specific IPC mechanisms (pipes and shared memory) mentioned in the problem statement should be used and no other IPC mechanism should be used in the entire implementation.

10. You are not allowed to take any additional inputs other than those mentioned in the problem statement. If you do, marks will be deducted.

11. No synchronization constructs like mutex and semaphore should be used.

12. Multithreading should not be used in the implementation.

13. You need to close unused pipe ends, wherever applicable.

14. Marks will be deducted if you violate any constraint mentioned in the problem statement.

SUBMISSION GUIDELINES:

- All programs should be POSIX-compliant C programs.
- All codes should run on Ubuntu 22.04 systems.
- Submissions are to be done on the CMS course page under the Assignment 1 section.

- There should be only submission per group.
- Each group should submit a zipped file containing all the relevant C programs and a text file containing the correct names and IDs of all the group members. The names and IDs should be written in uppercase letters only. The zipped file should be named as GroupX_A1 (X stands for the group number). You will be notified of your group number after a few days.
- If there are multiple submissions from a single group, any one of the submissions will be considered randomly.
- Your group composition for Assignment 1 has been frozen now. You cannot add or drop any group member now.
- No extensions will be granted beyond the given deadline.
- Do not attempt any last minute submissions. You need to be mindful of situations such as laptops not working at the last moment, CMS becoming unresponsive, etc.
- There will be penalty for late submissions.

PLAGIARISM POLICY:

- All submissions will be checked for plagiarism.
- Lifting code/code snippets from the Internet is plagiarism. Taking and submitting the code of another group(s) is also plagiarism. However, plagiarism does not imply discussions and exchange of thoughts and ideas (not code).
- All cases of plagiarism will result in awarding a hefty penalty. Groups found guilty of plagiarism may be summarily awarded zero also.
- All groups found involved in plagiarism, directly or indirectly will be penalized.
- The entire group will be penalized irrespective of the number of group members involved in code exchange and consequently plagiarism. So, each member should ensure proper group coordination.
- The course team will not be responsible for any kind of intellectual property theft. So, if anyone is lifting your code from your laptop, that is completely your responsibility. Please remember that it is not the duty of the course team to investigate cases of plagiarism and figure out who is guilty and who is innocent.
- Please be careful about sharing code among your group members via any online code repositories. Be careful about the permission levels (like public or private). Intellectual property theft may also happen via publicly shared repositories.

DEMO GUIDELINES:

- The assignment also consists of a demo component to evaluate each student's effort and level of understanding of the implementation and the associated concepts.
- The demos will be conducted in either the I-block labs or D-block labs. Therefore, the codes submitted on the CMS course page will be tested on the lab machines.
- No group will be allowed to give the demo on their own laptop.
- The codes should run on Ubuntu 22.04.
- All group members should be present during the demo.

- Any absent group member will be awarded zero.
- The demos will be conducted in person.
- The demos will not be rescheduled.
- Though this is a group assignment, each group member should have full knowledge of the complete implementation. During the demo, questions may be asked from any aspect of the assignment.
- Demo slots will be made available in due time. You need to book your demo slots as per the availability of your entire group.
- The code submitted on CMS will be used for the demo.
- Each group member will be evaluated based on the overall understanding and effort. A group assignment does not imply that each and every member of a group will be awarded the same marks.
- Any form of heckling and/or bargaining for marks with the evaluators will not be tolerated during the demo.

****