OPERATING SYSTEMS (CS F372)

SEM I 2023-2024

ASSIGNMENT 2

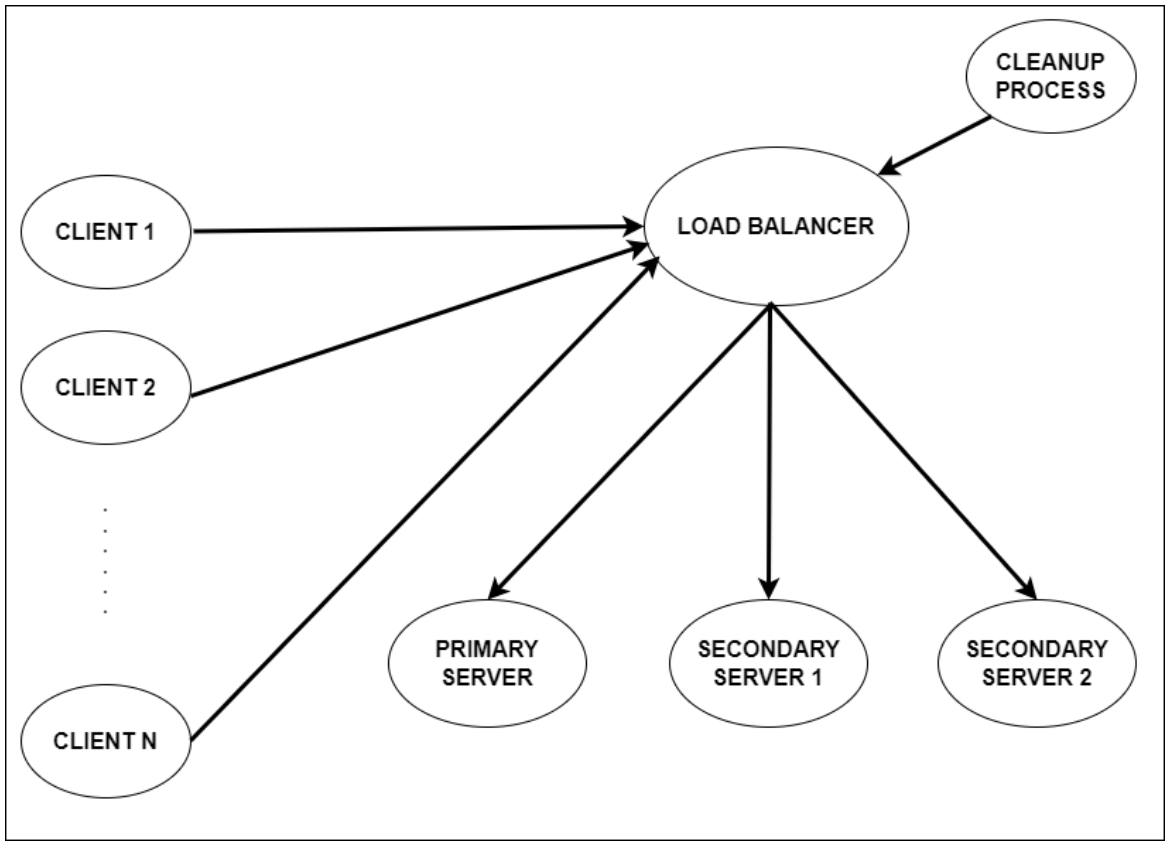
MAX MARKS: 45

DEADLINE: 17th NOVEMBER 2023, 11:59 PM (HARD DEADLINE)

NOTE: PLEASE READ THE ENTIRE DOCUMENT AND NOT JUST THE PROBLEM STATEMENT. THIS DOCUMENT CONTAINS IMPORTANT INFORMATION REGARDING SUBMISSION GUIDELINES, PLAGIARISM POLICY AND DEMO GUIDELINES IN ADDITION TO THE PROBLEM STATEMENT.

PROBLEM STATEMENT:

In this assignment, you will simulate an application for a distributed graph database system involving a load balancer process, a primary server process, two secondary server processes (secondary server 1 and secondary server 2), a cleanup process and several clients. The overall system architecture is shown below.



The overall workflow is as follows:

- Clients send requests (read or write) to the load balancer.
- Load balancer forwards the write requests to the primary server.

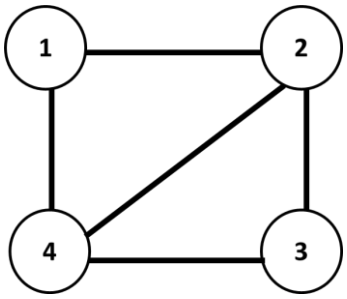
- Load balancer forwards the odd numbered read requests to secondary server 1 and the even numbered read requests to secondary server 2. Request numbering starts from 1.
- On receiving a request from the load balancer, the server (primary or secondary) creates a new thread to process the request. Once the processing is complete, this thread (not the server process) sends a message or an output back to the client.
- For terminating the application, a relevant input (explained later) needs to be given to the cleanup process. The cleanup process informs the load balancer. The load balancer performs the relevant cleanup activity, informs the servers to exit and terminates itself. On receiving the intimation from the load balancer, the servers perform the relevant cleanup activities and terminate.

Note that not all communications have been shown in the figure. In the rest of the document, wherever applicable, the primary and the secondary servers together will be referred to as servers only. More details regarding each component is provided below.

- **GRAPH DATABASE:** The database system can contain both cyclic and acyclic graphs. The graphs are unweighted and undirected in nature and may contain self loops. Each graph is represented as a text (ASCII) file. The naming convention of each file is `Gx.txt` where `x` is number (like `G1.txt` or `G2.txt`). The contents of each file are in the following format:
  - The first line contains a single integer `n` denoting the number of nodes present in the graph and n Ò 30.
  - The successive `n` lines represent the `n` rows of the adjacency matrix. The consecutive elements of a single row are separated by a whitespace character. The graph nodes are labeled as `1`, `2`, `3`, ... Thus, the first of the `n` lines corresponds to node `1`, the second of the `n` lines corresponds to node `2` and so on.

  A sample file and the corresponding graph are shown below.

```
4
0 1 0 1
1 0 1 1
0 1 0 1
1 1 1 0
```



  You can assume that the graph database system contains a maximum of 20 files. The database allows the clients to perform various read and write operations on the graphs. The write operations are facilitated by the primary server process and the read operations are performed by the secondary server processes, secondary server 1 and secondary server 2.

- **CLIENT PROCESS:** The clients send the requests to a load balancer process via a single message queue (message queue to be created by the load balancer only). Note that the clients do not send the requests

to the servers directly. You are not allowed to use more than one message queue in the entire assignment implementation.

Each client displays the following menu options.

```
1. Add a new graph to the database
2. Modify an existing graph of the database
3. Perform DFS on an existing graph of the database
4. Perform BFS on an existing graph of the database
```

Options 1 and 2 are write operations (to be performed by the primary server) and options 3 and 4 are read operations (to be performed by the secondary servers). For option 2, addition and/or deletion of nodes and/or edges can be requested by the client. Each client uses the following 3-tuple format for sending the request to the load balancer via the message queue: `<Sequence_Number Operation_Number Graph_File_Name>`. `Sequence_Number` will start from 1 and will keep on increasing monotonically till 100 for all the client requests sent across all the servers. This `Sequence_Number` corresponds to the request number mentioned earlier. **It is guaranteed that the `Sequence_Number` associated with each client request will be unique.** `Operation_Number` will be specified as per the menu options mentioned above. The client prompts the user to enter the `Sequence_Number`, `Operation_Number` and `Graph_File_Name` by displaying a prompt as follows:

```
Enter Sequence Number
Enter Operation Number
Enter Graph File Name
```

`Graph_File_Name` will be a new file name for option 1 and an existing file name for the remaining options. You can assume that the client will not deliberately send an existing file name for option 1 and a non-existent file name for options 2, 3 and 4. Additionally, for a write operation, the client after sending the 3-tuple request to the load balancer, writes the number of nodes of the graph (new or the modified one) and the corresponding adjacency matrix to a shared memory segment. The client prompts the user to enter the information to be written to the shared memory segment by displaying the following prompt:

```
Enter number of nodes of the graph
Enter adjacency matrix, each row on a separate line and elements of a
single row separated by whitespace characters
```

For a read operation, the client specifies the starting vertex for the BFS/DFS traversal. The client writes this information in a shared memory segment. The client prompts the user to specify this starting vertex by displaying the following message: `Enter starting vertex.` Each client can create a separate shared memory segment for every new request. Note that only the client should create the

shared memory segment.

For every request, the client receives some sort of message or output from the server (actually it will be a thread created by the server as explained later) once the request has been serviced via the single message queue. Once this is received, the client deletes the shared memory segment.

- **LOAD BALANCER:**
    - The load balancer receives the client requests via the single message queue. This message queue should only be created by the load balancer. It sends the odd numbered requests to secondary server 1 and the even numbered requests to secondary server 2 via the single message queue. These numbers are as per the `Sequence_Number` of the requests as discussed above.

- **PRIMARY SERVER:**
    - The primary server process receives the write requests from the load balancer via the single message queue.
    - After receiving a request, the primary server creates a new thread to handle it. This thread reads the contents of the shared memory segment (the number of nodes of the graph and the adjacency matrix) written by the client, opens the corresponding graph file (a new one for option 1 and an existing one for option 2), adds/updates the contents to/of the file and closes it.
    - After closing the file, the thread sends the message `File successfully added` for option 1 or `File successfully modified` for option 2 back to the client through the single message queue. The thread exits after this.
    - The client then displays this message on its console. For each distinct write request from a client, you may use a separate shared memory segment.
    - The request should be handled by a thread created by the primary server, not the server process itself.

- **SECONDARY SERVER:**
    - The read operations are handled by the secondary servers.
    - Each secondary server spawns a new thread to handle a client request.
    - Each BFS or DFS traversal needs to be implemented using multithreading and you can assume that for these operations, **the clients will choose only acyclic graphs**.
    - The client will have specified a starting vertex for either type of traversals via a shared memory segment. The secondary server thread will read this vertex from the shared memory segment.
    - While processing a client request for DFS, the secondary server thread creates a new thread for every unvisited node. At the same time, you should ensure the depth-wise traversal of the graph.
    - While processing a client request for BFS, the different levels of the graph will be processed

serially but the nodes of a specific level will be processed concurrently rather than serially. Thus, for processing a node of a specific level, the secondary server thread creates a new thread.

- However, note that you are not allowed to create all the threads in one go for either of the traversals. Also, you need to ensure that in every case, each parent thread waits for all its child threads to terminate.
- For DFS, the thread created by the relevant secondary server sends to the client a list of vertices such that each vertex in the list is the deepest/last vertex lying on a unique path of the graph. Thus, if a path in the graph is 1-2-5-3, then for this path 3 should be returned and this needs to be done for every other path in the graph. The secondary server thread sends this list to the client via the single message queue and exits. The client prints this vertex list on its console as a space separated list on a single line. For this case, the order of the list is not important.
- For BFS, the thread created by the relevant secondary server thread sends to the client the list of vertices generated by the traversal via the single message queue and exits. The client prints this vertex list on its console as a space separated list. Note that the vertices are to be printed exactly in the order in which they are traversed and as a space separated list in a single line. Otherwise, the output will be deemed as incorrect.
- Note that a hierarchy of threads is created for the read operations since each thread created by the secondary server for handling a client request will in turn create multiple threads for performing DFS or BFS.
- Only the thread created by a secondary server should read the starting vertex from the shared memory segment and only this thread should send the output back to the client via the single message queue. These should not be done by the server process.

- **CLEANUP PROCESS:** The cleanup process keeps running along with the clients, the load balancer and the servers.
  - The cleanup process keeps displaying a menu as:

    ```
    Want to terminate the application? Press Y (Yes) or N (No)
    ```

  - If N is given as input, the process keeps running as usual and will not communicate with any other process. If Y is given as input, the process will inform the load balancer via the single message queue that the load balancer needs to terminate.
  - After passing on the termination information to the load balancer, the cleanup process will terminate.
  - When the load balancer receives the termination information from the cleanup process, the load balancer informs all the three servers to terminate via the single message queue, sleeps for 5 seconds, deletes the message queue and terminates. If you can think of any other cleanup activity required for the correct execution of the application, you can do that.
  - On receiving the termination information, the servers perform the relevant cleanup activities and terminate.

- Note that the cleanup process will not force the load balancer to terminate while there are pending client requests. Moreover, the load balancer will not force the servers to terminate in the midst of servicing any client request or while there are pending client requests.

- **HANDLING CONCURRENT CLIENT REQUESTS:** Multiple read operations can be performed on the same graph file simultaneously. However, you need to be careful about simultaneous write operations as well as simultaneous read and write operations on the same graph file. Such conflicting operations have to be performed serially. You have to ensure this by using either semaphore or mutex. You need to use some locking mechanism on the graph files. You are free to use any synchronization construct between semaphore or mutex.

Tip: For handling inter-process synchronization, you can explore using Named Semaphore. However, you are free to design your own solution to the critical section problem.

NOTE:

- **The implementation of the entire assignment should be done using POSIX compliant C programs only.**
- **The load balancer, the cleanup process, the primary server and the secondary servers are to be implemented as individual processes. Use filenames as load_balancer.c, cleanup.c, primary_server.c, and secondary_server.c. For the secondary server, you will need to execute two instances of the compiled output of secondary_server.c and keep track of which one is your secondary server 1 and which one is secondary server 2.**
- **There can be several instances of clients running simultaneously.**
- **You can assume the graph files to be present in the same directory as your C files.**
- **`Sequence_Number` of client requests will range from 1 to 100.**
- **In case of any potential starvation, you can avoid the situation by processing client requests in increasing order of `Sequence_Number`. However, you should ensure that non-conflicting requests are processed concurrently.**
- **All IPC mechanisms are to be implemented as mentioned above in the problem statement. You should only use a single message queue in the entire assignment. The message queue based communications are to be done using message queues only. The shared memory based communications are to be done using shared memory segments only.**
- **The message queue should be created and deleted only by the load balancer process.**
- **The shared memory segments should be created by the corresponding clients. A client creates separate shared memory segments for each request. Once the request is serviced, the client destroys the shared memory segment.**
- **It is assumed that a client after sending a request, waits for the servicing of that request to complete before sending any further requests. However, multiple clients can send requests simultaneously.**
- **It is assumed that no client will send any request before the load balancer and the servers start**

executing.

- **Assume that**
  - **No client is terminated while it has a pending request. Clients will be terminated by pressing Ctrl+C.**
  - **None of the servers are terminated in the midst of executing client requests and while there are pending client requests.**
  - **The load balancer will not be terminated during servicing of client requests and while there are pending client requests.**
- **Wherever you are using multithreading, you need to make sure that each parent thread waits for the children threads to terminate.**
- **You should not create any child process to implement any aspect of the assignment.**
- **You should not make use of `sleep()` anywhere else other than where mentioned in the problem statement for the load balancer.**
- **For the primary server, the contents of the shared memory are read by the thread handling the client request and the message is sent by that thread to the client and not the main thread of the primary server and also not by the server process itself.**
- **For each secondary server, the contents of the shared memory are read by the thread handling the client request and the output is sent by that thread to the client and not the main thread of the secondary server and also not by the server process itself.**
- **You should do proper error handling for the relevant API calls, wherever applicable.**

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 2 section.**
- **There should be only one 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_A2 (X stands for the group number). You will be notified of your group number after a few days.**
- **Once your group composition freezes, you cannot add or drop any group member.**

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 the 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 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 tentatively during mid of November. 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.**

****