Midterm Project Report

CSE 344, Spring 2022

Sena Nur Ulukaya

1901042622

1. Design Decisions

1.1 ClientX

```
struct Matrix{
    int row;
    int col;
    int **matrix;
} Matrix;
```

Matrix struct is representing the data, with row, column, and 2D array.

Reading from File: I read from data.csv file to this matrix. I read the first row of the data file char by char and counted the commas to decide row and column count since it should be a square matrix. Then a matrix struct is initialized according to these values and filled in.

- If the matrix is not square or has an empty value program will be terminated.

Writing to FIFO: I converted the 2D matrix array to a 1D array and also added row, column, and client_id values to a buffer and wrote it to the FIFO.

Printing Response: I created a unique FIFO with process id to get a response. The server uses this FIFO and responds to it with a 1 or 0 and according to that, a response will be printed to a STDOUT.

1.2 ServerY

```
typedef struct worker{
    int pid;
    int pipe[2];
}Worker;
```

Worker struct represents worker process with PID and pipes. There's a poolNumber worker process.

The parent process creates a given number of processes and pipes for each of the child's processes. Also, it creates Server Z as a child process at the beginning.

The parent process read Fifo from the client and finds an available worker and writes the buffer to its pipe. If there's no worker available it will send it to Server Z.

Available Worker: A FIFO with PID is created if the child is working. Then I checked if this FIFO can be opened and checked if its error no is EEXIST to know the existing situation. If it exists then the worker is not available.

Child Processes: They read from the pipe and convert the 1D array to a 2D matrix again, do the calculation and send the response to the client via the client's FIFO.

Invertible Calculations: I calculated the determinant with co-factor, the matrix is invertible if the determinant is not zero.

1.3 Server Z

```c
typedef struct node {
    int start_address;
    int end_address;
} Node;

typedef struct queue {
    int front;  //index of the first element
    int rear; //index of the last element
    int size;   //node size of the queue
    int capacity; //max size of the queue
    Node nodes[MAX_QUEUE_SIZE];   //array of nodes
} Queue;
```

```c
typedef struct sharedmemory{
    struct queue queue;
    int size;
    int fd;
    sem_t available;
    sem_t critical;
    int busy;
}SharedMemory;
```

I used a circular queue with a constant capacity. I used the start and end addresses bytes for each element(request). SharedMemory is the info struct for shared memory. It has semaphores, count of busy processes, size, fd, and queue.

I divided all the shared memory's size byte to byte and each node has this start and end byte in it.

Queue: Queue is in a circular array format. It has enqueue, dequeue, is_empty functions in it separately.

- Enqueue: Adds an element to the rear. It's adding the starting and ending address byte to the rear of the queue.
- Dequeue: Removes the front element and returns the start and end addresses bytes.

SharedMemory: Shared Memory is opened with shm_open, and its address is obtained with mmap. SharedMemory struct is copied to this address. (memcpy)

- **Finding Place:** I checked for an available place in the queue by checking its rear and front addresses bytes with total size. If there's an available place for the request, it will return the starting address of the found place.
- **Reading to Shared Memory:** I simply used the dequeue function and copied the request between return address bytes.
- **Writing to Shared Memory:** I tried to find a place with a find place function. If it's successful, I used this starting place and start + size for the ending place. I copied the request to this address using memcpy.

Server Z creates its workers with the given value and initializes shared memory and the semaphores. The parent process reads the pipe to get the request from Server Y and writes it to the shared memory.

Semaphores "full" and "empty" control the available workers. Child processes increase the "empty" semaphore and decrease the "full" semaphore when they're taking care of a request. The parent process decreases the semaphore "empty"' and increases the "full" semaphore when a request comes. So it's in the producer-multiconsumer fashion.

Semaphore "Critical" controls the critical regions so no other process will read or write to the shared memory at the same time. Child processes and parent processes decrease when they're reading/writing to shared memory. Increase it after finishing, so no other process can interrupt during the I/O.

Its child processes act like Server Y's child. The only difference is they read the request from the shared memory and uses semaphores.

1.4 Logfile

Busy Pool Count: I checked the FIFOs of server Y processes as mentioned above and used count in the shared memory struct for server Z processes. Server Z child processes increment the value of it while working and decrement after they finish.

Invertible Matrix Count: I used SIGUSER1 to count invertible matrices, after each response.

1.5 Daemons

I created a FIFO for serverY at the beginning and while running that FIFO exists so no other instance of it can run on another terminal.

1.6 Signals

When SIGINT is received, all the clean-up after children processes, the closing of open files, pipes, and FIFOs are made.