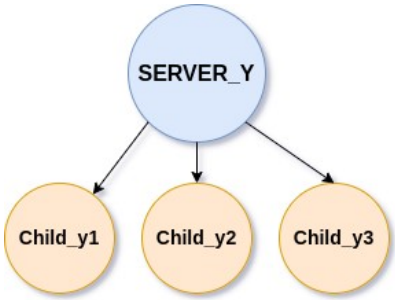
DESIGN:

Our task in this homework is to implement a server called Y and its worker process. If server Y's worker processes are not free to take a new request, we are running a new server called Z and its worker process. To take request from the clients.

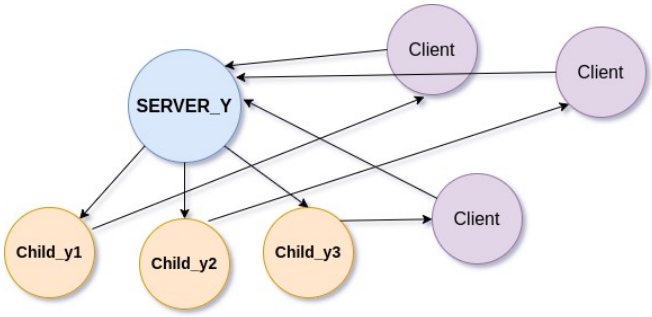
Initially our server start with the some child worker process. Their number can chance the command line argument.

Ex:



Our client start running and they are expected response from the server.

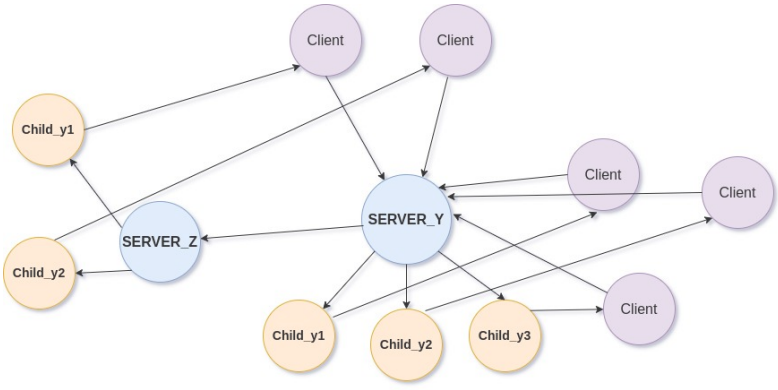
Server getting request from the client through the FIFO file and the the server transmitting
this request to the their child worker processes from the pipe.



Since child processes are blocked at the pipe in read end from the beginning of the program, they are ready to get command from the server Y,

The first thing in the child processes sending a signal to the parent processes through SIGUSR1 and this means for the parent processes "I have a job wait another signal from me in the mean time I cannot get another job". After that child starting the calculation about the request and starting a clock to measure the response time. After the calculation Child processes directly forwarding the response through the FIFO file. The name of the FIFO is Client process's id so It won't be any conflict between client and Child response process. Then the child process's sending another signal to the parent SIGUSR2 process's and tells "I can take another request I am done the previous request". And process's blocking the pipes read line. All activities of the child's process's saving the server_logs file.

If the all child's are full and still came the request to server Y. Then server Y decide to create another server called server Z. forwarding these request to him. Then Server Z creates its own child process's to handle these request. And child process's again the response directly forwarding the clients using FIFO named Client id.

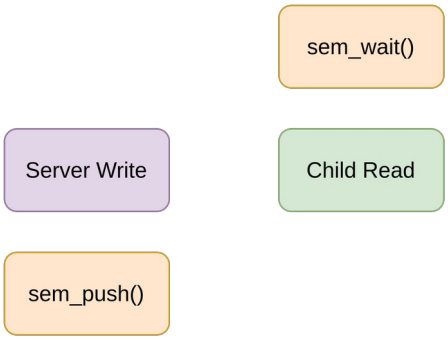


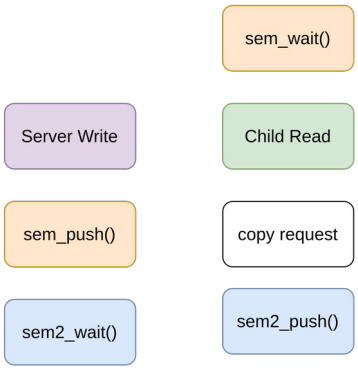
Server Y communicate with the Server Z using Pipes. And server Z communicate with its child process's using shared memory. Unlike others in shared memory we need to manage the synchronization between Server Z and its child the idea is simple first server Z write the data to shared memory and child process's reading request from the parents process's write. We are using the semaphores to manage the synchronization. We want to first write parent and then read the child.

So Determine the semaphore initially 0 and push after writing operation and wait before the child read operation.

In this way, the child process will not have a chance to read from the request parent process without writing to the shared memory.

Another semaphore was used in order not to write the newly received transactions by Server Z to the current location. As soon as the information received by the child is received, it is saved in a local variable and the memory address is released and Server Z can receive new requests.





Encountered Problems:

- Keep track of all response in Server Y: Solved using semaphore as a counter
- Sending pointer in the IPC Communication: Convert it to static big size array
- Shared Memory overwrite unprocessed data: Solved adding another semaphore
- Child process's communicate with clients: Added client id to request.
- Signal was dividing the file open process's: Added while loop to open syscall and add a signal condition
- Signal handling not properly work outside of handler: Moving inside of the handler

SIGINT HANDLING:

We are storing the child process's ids right after the fork and sending SIGINT signal in the signal handler.

server.c

backup_server.c

for child process's:

```c
int id = 0;
pid_t child_pid = 1;
for(int i=0; i<pool_size; ++i){
    child_pid = fork();
    pid_arr[num_of_ids++] = child_pid;
    if(child_pid == 0){
        id = i+1;
        break;
    }
    if(child_pid == -1){
        write(child_log_fd,"Forking failed\n", sizeof("Forking failed\n"));
        exit(EXIT_FAILURE);
    }
}
```

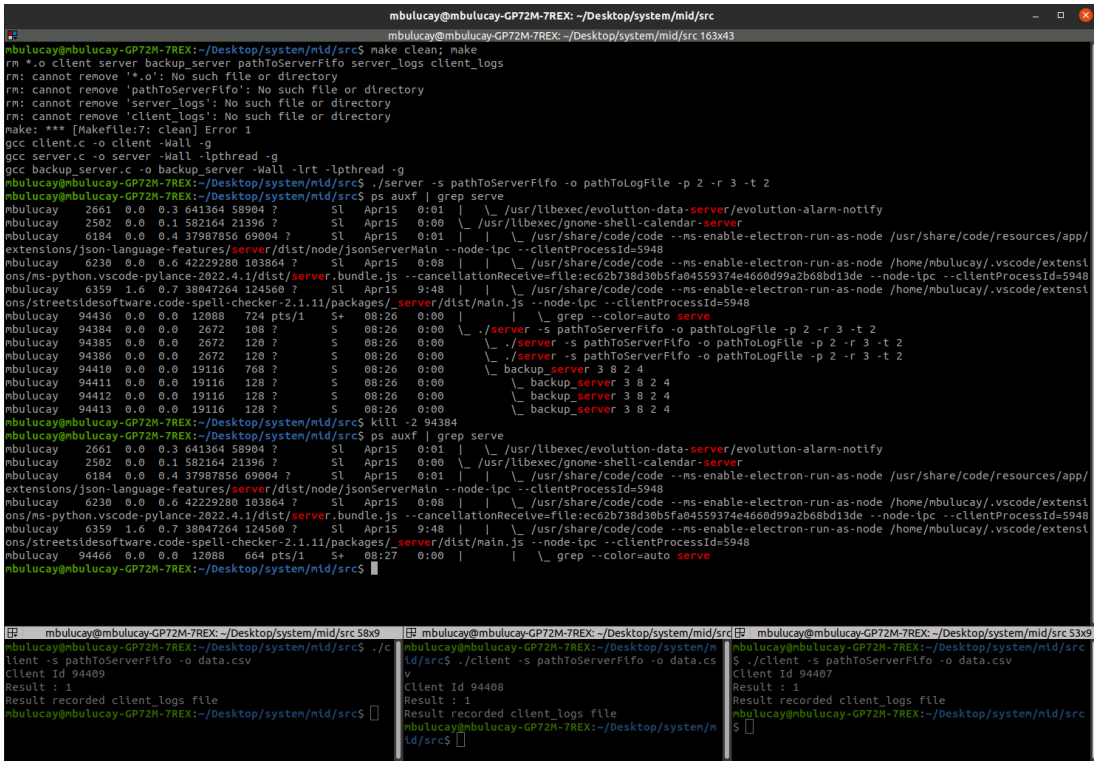
```c
int id = 0;
pid_t child_pid = 1;
for(int i=0; i<pool_size; ++i){
    child_pid = fork();
    pid_arr[num_of_ids++] = child_pid;
    if(child_pid == 0){
        id = i+1;
        break;
    }
    if(child_pid == -1){
        write(child_log_fd,"Forking failed\n", strlen("Forking failed\n"));
        exit(EXIT_FAILURE);
    }
}
```

for server Z:

```c
backup_server_run = 1;
pid_t backup_pid = fork();      You, 2 days ago •
pid_arr[num_of_ids++] = backup_pid;
```

Then when we get an sigint signal killing these child process's

```c
for(int i; i<num_of_ids; i++){
    kill(pid_arr[i], SIGINT);
    wait(&pid_arr[i]);
}
exit(EXIT_SUCCESS);
```



clints_log

```
You, 1 second ago | 1 author (You)
2022-04-16 08:26:31: Client PID#94407 (data.csv)  is submitting a 6x6  matrix
Client PID#94407: the matrix is invertible, total time 2.521 seconds, goodbye
2022-04-16 08:26:31: Client PID#94408 (data.csv)  is submitting a 6x6  matrix
Client PID#94408: the matrix is invertible, total time 2.753 seconds, goodbye
2022-04-16 08:26:32: Client PID#94409 (data.csv)  is submitting a 6x6  matrix
Client PID#94409: the matrix is invertible, total time 2.849 seconds, goodbye
```

server_logs

```
2022-04-16 08:26:23: Y:Worker PID#94385 is handling client PID#94407, matrix size 6x6, pool busy 1/2
2022-04-16 08:26:23: Y:Worker PID#94386 is handling client PID#94408, matrix size 6x6, pool busy 2/2
2022-04-16 08:26:32: Z:Worker PID#94411 is handling client PID#94409, matrix size 6x6, pool busy 1/3
2022-04-16 08:26:23: Y:Worker PID#94385 responding to client PID#94407: the matrix is invertable.
2022-04-16 08:26:23: Y:Worker PID#94386 responding to client PID#94408: the matrix is invertable.
2022-04-16 08:26:32: Z:Worker PID#94411 responding to client PID#94409: the matrix is invertable.
2022-04-16 08:26:54: SIGINT received, terminating Z and exiting server Y. Total requests handled: 3, 3 invertible, 0 not. 1 requests were forwarded.
2022-04-16 08:26:54: Z:SIGINT received, exiting server Z. Total requests handled 1, 1 invertible, 0 not.
```