GTU Department of Computer Engineering
CSE 344 – Spring 2023
Homework #4 Report

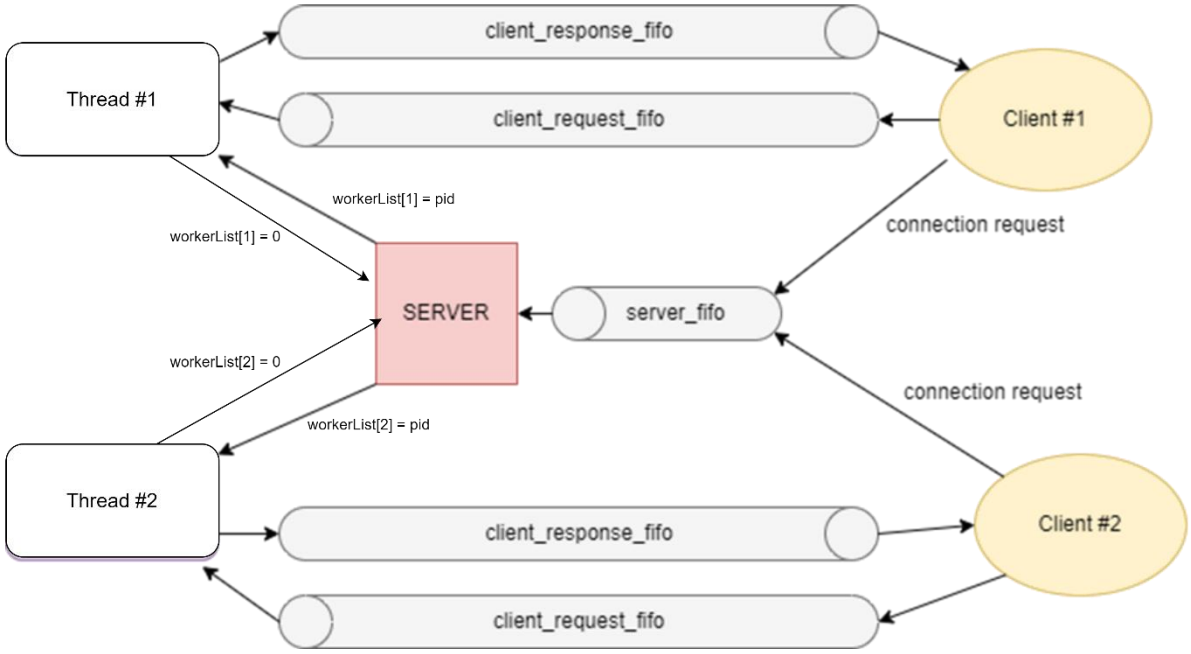
Hasan Mutlu
1801042673

Implementation Details and General Workflow

This version of the program works very similar to the midterm version. In this version, the shared memory is excluded. Threads, condition variables and mutexes are used in addition to the FIFO's and semaphores from midterm. Only the biboServer program is changed. The client and other utilities are exactly same.

In this version, a pool of threads are created according to the command line arguments. These threads then go on to wait until a client is assigned to them. When a new client arrives, server checks if there is an available worker. If there is, it assigns the client PID to that worker's index in the workerlist. The thread checks if a client is arrived to its index in and starts serving client if there is. After the client decides to exit, the serving worker marks itself available and goes on to wait the next client.

Inter Process Communication (IPC)



The IPC is established similarly to the midterm. The clients send connection request to server via server fifio. Server assigns the client's PID to the thread's index in the workerlist. The thread responds to client using client response fifo and client then sends file operation requests to the thread via client request fifo.

Synchronization and Worker Management

Synchronization between client and server threads are the same as midterm. On the other hand, there are new synchronization considerations are taken between the server and worker threads.

There are shared variables between server and server and worker threads. These are:

- ➔ Counter: Counts number of available workers.
- ➔ Workerlist: A list containing which thread is assigned to which client and which threads are currently available.
- ➔ clientNums: A list containing the connected client's number, e.g. client5.

In order to prevent race conditions using these data sections, a mutex called critMutex is used every time a modification on these fields is made.

In order to prevent busy waiting in threads, condition variables are used. A condition variable and a mutex to use alongside it are allocated for each thread. When thread is ready to serve a new client, it calls pthread_cond_wait to be blocked until a client is assigned to it. When a client arrives, the server wakes it up using pthread_cond_signal and thread starts serving the client.

When a client exits from a thread, it wakes up the server using server semaphore to notify that it is available to accept new clients.

Signal Handling

Signal handling is a little different than the midterm. In this version, all the cleanup is done by the main program.

When the client wants to terminate the server, it sends SIGUSR1 to the server. In this way, server can know what is the source of the termination. After printing required information, it continues termination by sending SIGINT to itself.

When server receives SIGINT, it first fetches the client PID's that are waiting on the queue to be connected and sends them SIGINT signal. After that, it fetches the list of clients that are currently being served from the workerlist and sends SIGINT signal to them. It also unlinks request fifos of each client since threads can't handle individual signals and clean them. Threads are canceled and condition variables-condition mutexes of each worker are destroyed as well. After that, the server frees the remaining resources and terminates.

File Handling

File handling is the same as midterm. The file handling functions are implemented in the “utils.h” file. When the thread handles a request from the server, it first parses and checks the validity of the request, and then call the functions defined on utils.h file to handle the request. The race conditions to write the same file between processes are avoided by using flock().

Requirements

As far as the program is tested, all requirements related to general workflow of such client-server model is working fine. The communication, synchronization, signal handling and services are satisfied. The resources are used efficiently and freed when terminated. No junk files are left behind.

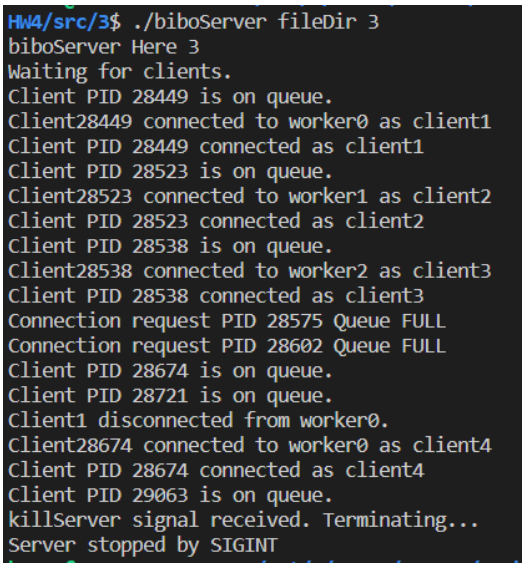
Test Plan

- ➔ IPC:
  - Testing the FIFO and shared data segment communication between the three processes.
  - Testing the client/queue management.
- ➔ Synchronization:
  - Testing the presence of busy waits.
  - Testing the use of waits and posts, blocking situations.
  - Testing the blocks on mutexes and condition variables.
  - Testing race conditions using same data segment.
- ➔ Signals:
  - Testing SIGINT signals from command line.
  - Testing killServer signal from a client.
  - Testing if reasources are freed and no junk files are left behind.
- ➔ File Operations:
  - Testing the validity of operations.
  - Testing exception situations.
  - Testing opening different directories.
  - Testing race conditions, writing to same file at the same time.

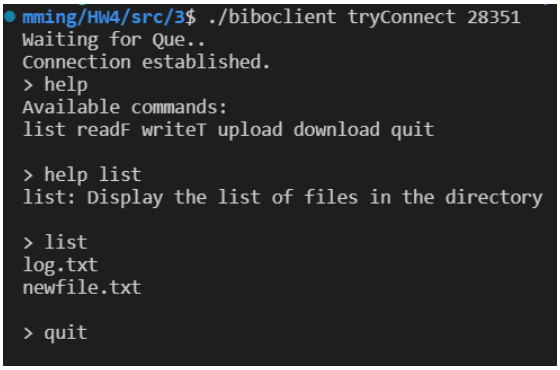
Test Results

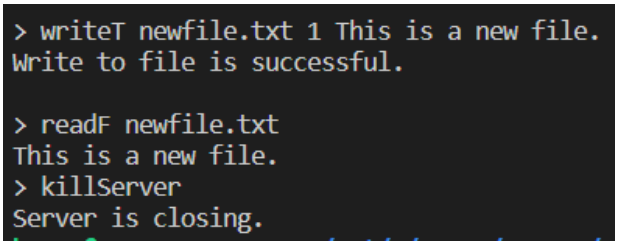
The program is tested with different pool sizes. Here in these examples, it is tested with 3 threads.

Server Output:



Client Outputs: Every screenshot is of a different client.





```
> readF thereisnofilelikethis
No such file or directory

> downlaod thereisnofilelikethis
Unknown command. Enter 'help' to see available commands.

> download invalidfile
No such file or directory

> Client stopped by SIGINT. Server may be terminated.
```

```
> download newfile.txt
Downloading file is successful.

> upload utils.h
Uploading file is successful.

> readF utils.h 1
#include <time.h>

> download utils.h
File exists
```

```
gramming/HW4/src/3$ ./biboclient tryConnect 28351
Waiting for Que..
Queue is full. Exiting.
```

```
mming/HW4/src/3$ ./biboclient Connect 28351
Waiting for Que..
Client stopped by SIGINT. Server may be terminated.
```