|  |
| --- |
| 05-May-18 |

|  |
| --- |
| Systems Programming Project Report |
| Server – Client Model Using Linux APIs |
| Syed Zain Zaidi - 11476 |

System Architecture:

Client:-

Each client is multi-threaded in nature. While the main thread of the program reads from the terminal and writes on the server’s socket file descriptor ( in an infinite loop ), the secondary thread remains in an infinite loop (called every second), checking if something has been written to the socket file descriptor.

Server:-

The server makes use of multiplexed I/O by initially starting an infinite loop and putting the Terminal’s Input file descriptor and the listening socket into the poll() API.

If there is a return event (POLLIN) on the Terminal’s Input (STDIN\_FILEIO) file descriptor, the server parses the command and writes to a pipe (this will be further discussed later) if necessary. It then reads from another pipe and displays an output on the terminal as required.

If there is a return event (POLLIN) on the listening Socket file descriptor, the server proceeds to accept the connection from a potential client. Once accepted, the server stores the listening socket file descriptor of the client. Now the server establishes two pipes, one that it will write on and another that it will be reading from. The server follows this by an immediate fork() which spawns a child process of the server. This child process will now specifically cater to the clients requests.

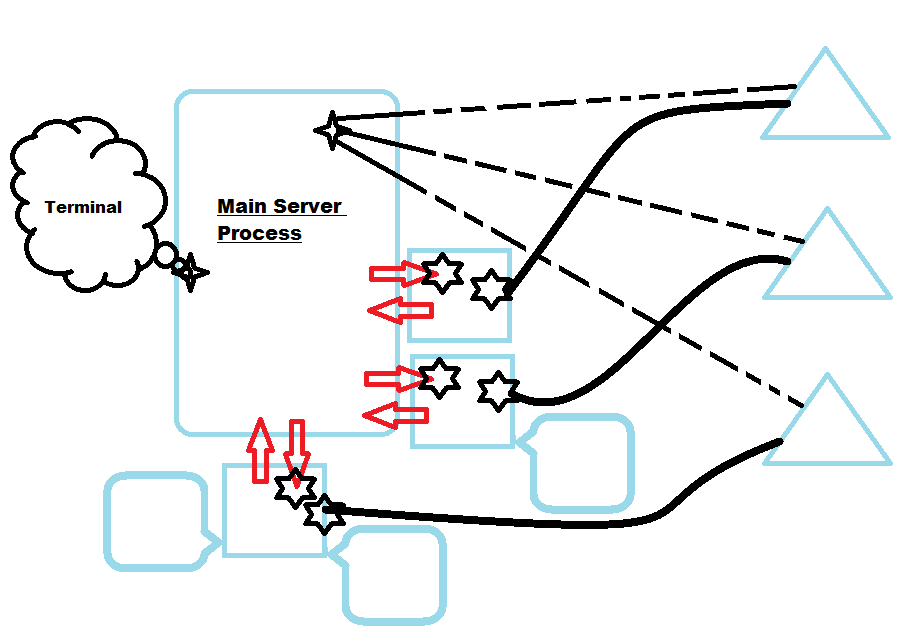
After creation, the child process starts another infinite loop. Now it puts the server’s writing pipe’s reading end’s file descriptor and the client’s listening socket’s file descriptors into the poll() API.

Now the child waits for a return event on either of the file descriptors.

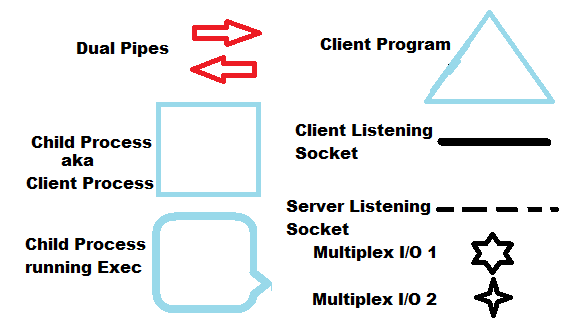
If the client program sends a request, returning an event for the client’s listening socket’s file descriptor, the child process responds to the command. In a special case, if the command is “run …” the child will perform another fork() and then execute the requested program in the child process’ memory space.

If the server writes to the pipe, the child process responds by writing to its own pipe so the server may sequentially read from that pipe.

Architectural Diagram



Key / Legend:



System Limitations:

1. Delayed “End Time” in process table entry for closed programs.  
   Cause: If a SIGCHLD is being handled and another occurs, it will be ignored. Therefore all existing table entries for potentially closed processes need to be rechecked when list command is called in client process.
2. Unified Process Table entries: If a client process exists, its process table is removed therefore when the server requests for a complete process table, it will not receive closed processes from the client process that has been closed too.  
   Cause: Quick coding due to time constraints.  
   Solution: Storing all client processes in a structure maintained by the server.
3. Certain limitations are caused by array sizes which cannot be changed at run time. This includes connections list array, reading input array from the terminal and displaying output array for the terminal. The process list is not an array therefore it doesn’t have this limitation.  
   Solution: reassignment of array sizes or use of more linked lists.

Help:

A (not so) comprehensive list of commands:  
  
**Server Side:**

-list [ conn | proc | IP] [Port]

Prints out a table. If conn argument is passed, prints out all the connections table. This includes all previously established connections. If proc argument is passed, prints out a table of all the programs that were started by the clients. If a valid client program IP address is passed, it should be followed by a space and the port of that client program. This will print a table of all the programs executed by that specific client.

-msg [ all | IP] [Port] [msg]

Sends a specified message to the client. If all argument is passed, the message is broadcasted to all clients. If a valid client IP address is passed, a port number is expected with a space in between. The last argument is a message sentence.

-exit

Gracefully terminates the server and all its child processes (including executed programs).

**Client Side:**

-help  
Prints a list of all client side commands and their usage

-add [number1] [number2] …. : All provided numbers are added.

-div [number1] [number2] …. : Number1 is divided by all the numbers provided after it.

-sub [number1] [number2] …. : All provided numbers are subtracted from the previous number.

-mul [number1] [number2] …. : All provided numbers are multiplied

-kill [all | pid |pname] : If all argument is passed, client terminates all executed programs. If a valid pid is passed that belongs to a program executed by the client, it will be terminated. If a valid name of the program is passed as the first argument, the program is terminated.

-run [pname] : Executed the program specified in pname variable.   
Note: program must be in path variable directory.

-list [all] : If all argument is provided, the program prints a complete list of all programs that were executed by the client. If no argument is provided, only active executed programs are printed.

-con [IP] [Port] : Connect to the IP address and port number provided.

-discon : Disconnect from the server, terminating the client process on the server.

-exit : Terminate the Client Program and the Client Process on the server.

Extra Features:

1. Color coded messages outputted on server terminal to represent connection establishment and de-establishment along with flow of traffic to and from the server.
2. Friendly user interface on Client program indicating when program is ready for input and all outputting all appropriate messages on the terminal.
3. msg command on server side.
4. Client can provide multiple commands to the server in a batch using the ; delimiter. For example add 4 2;sub 9 3;run leafpad\n
5. Process List designed using linked list therefore it can grow in size without limitations.
6. Error Checking and Input Validation for multiple inputs.