Tyler Trepanier-Bracken A00850517

Technical Report

Linux Interprocess comunication

Table of Contents

[Introduction 2](#_Toc442307910)

[Server 2](#_Toc442307911)

[Client 2](#_Toc442307912)

[Instructions for program usage 3](#_Toc442307913)

[Unzipping and compiling the program. 3](#_Toc442307914)

[Running the Client application 3](#_Toc442307915)

[Running the Server application 3](#_Toc442307916)

[Summary 4](#_Toc442307917)

[Test Cases 5](#_Toc442307918)

[Test type 1: Standard Smoke Test 5](#_Toc442307919)

[Test type 2: Standard intended program operation 6](#_Toc442307920)

[Test type 3: Introducing Multiple Clients 7](#_Toc442307921)

[Test type 4: Stress tests / testing the program limits 9](#_Toc442307922)

[Conclusion 10](#_Toc442307923)

# Introduction

The objective for this application is to take advantage of the Linux operating system’s internal message queue. This message queue can be used to communicate between processes that serve as a Client as well as the Server’s children that will handle the children.

This document holds information on basic operation of the programs as well as documenting the capabilities of this program.

The program works via two separate applications.

* The server runs automatically after you run it via command-line.
* The client requires the user to enter in their desired file, and if desired the client’s priority.

Included inside of every Client is a thread that is dedicated to the reading of messages directed at the specific client.

## Server

The server is a very simple application that accomplishes the task of reading all client-requested files and sending the contents of those files using the Linux message queue. Any improperly constructed filenames, the server will send a message to a client indicating that there was an error.

**For complete instructions on how to run the server, please go to the Instructions section.**

## 

## Client

The client application makes file requests to the server based on input received from the command-line (aka, upon program execution). The user may include a priority in between 1-1000, where 1 is the maximum priority, which will affect the rate of message content receiving.

**For complete instructions on how to run the client, please go to the Instructions section.**

# Instructions for program usage

## Unzipping and compiling the program.

1. Unzip the zip file given to you to the folder containing the zip file.
2. Navigate using your terminal application to the folder containing the zip file.
3. Using the terminal, type in “make all” and press enter.
4. The script will do all the work to create the file.
5. Now you have the Client program, and the Server program.

## Running the Client application

1. Move the desired file that contains ascii characters to the folder with the Compiled program.
   1. Note: all irregular files may cause undesired output.
2. Using command-line, enter in “./Client [filename] [priority]”.
   1. [filename]
      1. Your desired file to output to the terminal.
      2. Any invalid filenames (case sensitive) will return errors.
   2. [priority]
      1. Optional field that does not have to be filled.
      2. Defaults to maximum prioritys
      3. This field is a number that is in between 1 and 1000 where 1 is the maximum priority.
3. Watch the program return the contents of the file (or an error on invalid filename).
4. After the program is finished, it will terminate.

## Running the Server application

1. Move all relevant files that contain ascii characters to the folder with the Compiled program.
   1. Note: all irregular files may cause undesired output.
2. Using the command-line, enter in “./Server”
3. This will automatically run the application, displaying all incoming clients as well as their requested files.
4. Exit the application by pressing ctrl-c at any time.

# Summary

The maximum amount of processes this program can handle is four client processes. Anything more than four clients bog down the system too much and all the clients lock-up. My belief is that the message queue is flooded with too many requests and as a result, all of the processes become starved for information. In addition, both clients and servers can ctrl-c fine and eliminate resource usage. However, having a server activating sending messages will cause the clients to hang and forever wait for the final message.

# Test Cases

## Test type 1: Standard Smoke Test

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| TEST NUMBER | SCENARIO | EXPECTATIONS | RESULTS | CONCLUSION |
| 1 | Compiling the program creates two separate files: Server and Client | Two files created:  Server and Client | Server and Client have been created. | PASS |
| 2 | Recompiling the program deletes the previous programs and re-creates the Server and Client | Server and Client were deleted and re-created. | Server and Client were deleted and re-created. | PASS |
| 3 | Starting the Server and immediately exiting the program with no active Client. | Server displays nothing and waits for input.  After ctrl-c is hit, it exits. | Server displays nothing and waits for input.  After ctrl-c is hit, it exits. | PASS |
| 4 | Starting the Client and immediately exiting with no Server by pressing ctrl-c. | The Client hangs for a bit waiting for a server reply. When the user presses ctrl-c, the client will immediately fail. | No display. | PASS |
| 5 | Starting the Client without a filename. | Displays the usage instructions of the Client and immediately exits the program. | Displays the usage instructions of the Client and immediately exits the program. | PASS |
| 6 | Starting the Client with a proper file but there’s no priority and no listening server. | Client defaults to maximum priority but makes no visible output. | No visible change. | PASS |
| 7 | Starting the Client with a proper file and priority but no listening server. | A client receives the priority and sends the message to the message queue but makes no visible change. | No visible change. | PASS |

## Test type 2: Standard intended program operation

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| TEST NUMBER | SCENARIO | EXPECTATIONS | RESULTS | CONCLUSION |
| 1 | Running the Server with a Client previously requesting a proper file.  Input is:  ./Client warandpeace | The Server almost immediately gets the Client’s file contents. | Sending warandpeace to client: 5388  Sending to 5388 complete… | PASS |
| 2 | Server operating before the execution of the Client.  Input is:  ./Client warandpeace | The Server immediately gets the Client’s file contents. | Sending warandpeace to client: 5586  Sending to 5586 complete | PASS |
| 3 | Running the Server application with no Client | The server will be forever waiting for a client to communicate. | The server displays nothing. | PASS |
| 4 | Running the Client without a listening Server and proper input with a valid file inside.  Input is:  ./Client warandpeace | The client sends a message to the queue and waits for a server’s response indefinitely. | The client displays nothing | PASS |
| 5 | Running the Client without a listening Server with an invalid file  Input is:  ./Client badinput | The client sends a message to the queue and waits for a server’s response indefinitely. | The client displays nothing. | PASS |
| 6 | Running the Client without a listening Server with improper format.  Input is:  ./Client | There is an error message displaying the usage instructions on how to use the application. | Usage: [Filename] [Priority].  Please note that the priority is optional. | PASS |

## Test type 3: Introducing Multiple Clients

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| TEST NUMBER | SCENARIO | EXPECTATIONS | RESULTS | CONCLUSION |
| 1 | Two clients using a servers with the same priority | Both clients will finish receiving their file at approximately the same time. | First time: First client finished first  Second time: Both finished at the same time.  Third time: Second client finished first. | Results inconclusive |
| 2 | Two clients using a server. One with max priority (1) and the other with minimum priority (1000) | The highest priority client finishes faster than the lowest priority client. | Highest priority finished at: 22:39:54pm  Lowest priority finished at: 22:40:15pm  21 second difference | PASS |
| 3 | Two clients using a server. One with medium priority (500), the other with max priority (1) | The medium priority will finish a few seconds after the highest priority. | Highest priority finished first  22:48:08pm  Medium priority  22:48:18pm  10 second difference. | PASS |
| 4 | Three clients at the same priority level. | They will finish in roughly the same amount of time. | All three clients finished within one second of each other. | PASS |
| 5 | Thee clients, two at high priority and one with the lowest priority. | The two high priority ones will finish first before the lowest priority. | A (high) finished at 22:54:42pm  B (high) finished at  22:54:41pm  C(low) finished at  22:55:02pm  10 second difference between each client. | PASS |
| 6 | Three clients priorities at:   1. High [1] 2. Medium [500] 3. Low [1000] | High finish first.  Medium finish second.  Low finish third | Refer to test 6’s picture located after this table. | PASS |
| 7 | Four clients at the same highest priority. | They should finish in roughly the same amount of time. | Refer to test 7’s picture  All of the clients finished in roughly the same amount of time. | PASS |
| 8 | Four clients with descending priority   1. High[1] 2. Medium[333] 3. Low [666] 4. Min [1000] | High: 1st  Medium: 2nd  Low: 3rd  Min: 4th | The first client finished really fast however the other clients took an extra 3min minutes and only finished a minute away from each other. | PASS |
| 9 | Five clients at the same highest priority. | They should finish in roughly the same amount of time. | Segmentation fault. | FAIL |
| 10 | Five clients with descending priority   1. Max[1] 2. High[5] 3. Medium[20] 4. Low [100] 5. Min [1000] | Max: 1st  High: 2nd  Medium: 3rd  Low: 4th  Min: 5th | All the processes froze. They were unable to receive message because there are five process and with a max buffer size of 4080, the queue filled up. | FAIIL |

## Test type 4: Stress tests / testing the program limits

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| TEST NUMBER | SCENARIO | EXPECTATIONS | RESULTS | CONCLUSION |
| 1 | Pressing ctrl-c in the middle of sending a file. | It will crash (experience) | The server successfully closes but the client is left hanging. | FAIL |
| 2 | Five clients at the same highest priority. | They should finish in roughly the same amount of time. | Segmentation fault. | FAIL |
| 3 | Five clients with descending priority   1. Max[1] 2. High[5] 3. Medium[20] 4. Low [100] 5. Min [1000] | Max: 1st  High: 2nd  Medium: 3rd  Low: 4th  Min: 5th | All the processes froze. They were unable to receive message because there are five process and with a max buffer size of 4080, the queue filled up. | FAIIL |

# Conclusion

With my priority system, there are 1000 levels of priority and each level of the priority has an effect on how soon a message arrives to the Client process. It appears that all the priorities in between 300 and 1000 do not have that much of a difference in terms of transmission time. Meanwhile, the priorities in between 1-100 make a huge difference and result in a speedy message.

During my testing, four clients was the magic number for my laptop to handle. However, as soon as I had five clients, my laptop was unable to process all the messages and some processes segmentation faulted. In addition, pressing ctrl-c on the server, while it was sending information, left all the clients in suspension where they were unable to terminate naturally. As a result, I had to manually input ctrl-c in order for them to terminate. On this error, I also had to clear the message queue because the queue was tainted with bad messages that would not allow any more messages to enter the queue.

In conclusion, the message queue is a great way for interprocess communication however