

**Lab Report**

|  |  |
| --- | --- |
| **Course**: | Operating System Principle |
| **Semester**: | 2nd semester of the academic year **2020-2021** |
| **Major**: | Software Engineering |
| **Class**: | 2019 |
| **Student Name**: | Fu Ruoxuan |
| **Student ID:** | 222019321062060 |
| **Teacher:** | ZHAO, Hengjun (赵恒军) |

**School of Computer and Information Science**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Name | | Interprocess communication in Linux --- Named Pipe | | | |
| Date | | April, 2021 | Type | | □Confirmatory  √ Design  √ Comprehensive |
| 1. **Objective & Requirements**    1. Understand named pipe inter-process communication (IPC) in Linux    2. Grasp named pipe operations    3. Can use named pipe to write application programs    4. Review multithreaded programming | | | | | |
| 1. **Experimental environment (**platform and software**)**   Ubuntu 16.04 or higher versions | | | | | |
| 1. **Experimental content and design** (Main Content, Procedure, Codes and Results) 2. Task 1 3. Create two processes, called A and B 4. Create a named pipe (using mkfifo function call) , say **f**, that are shared by A and B. 5. A and B communicate through **f** as follows:    * 1. A repeatedly reads inputs from keyboard and then write the information to **f**      2. B repeatedly reads information from **f** and then output it to the screen   (hint: you many use fgets() to read inputs from keyboard)   1. Task 2 2. Create two processes, called A and B 3. Create two named pipes (using mkfifo function call) , say f1 and f2, that are shared by A and B. 4. Let A and B communicate through f1 and f2.    1. Inside A, create two threads, one reads from f1, and the other writes to f2    2. Inside B, create two threads, one writpres to f1, and the other reads from f2 5. The threads in A or B for writing will accept an inputted string from the keyboard and then write it to the corresponding fifo 6. The threads in A or B for reading will read the string from the corresponding fifo and print it to the screen 7. When a thread in A reads“88”from or writes“88”to a corresponding fifo, then A quit; the same for process B. 8. Please provide your procedure to perform the tasks and source codes.   Task1:   1. First write a program to create a fifo. Compile and run it. 2. #include <sys/types.h> 3. #include <sys/stat.h> 4. #include <fcntl.h> 5. #include <unistd.h> 6. #include <stdio.h> 8. **int** main() 9. { 10. **int** fd; 11. fd = mkfifo("./myFIFO",0644); 12. **if** (fd == 0) 13. { 14. printf("myFIFO created successfully!"); 15. } 16. **else** 17. { 18. printf("myFIFO created errorly!"); 19. } 21. **return** 0; 22. } 23. And then write the reader source file and writer source file using fgets() function to get input from the keyboard which includes ‘\0’and ‘\n’. Compile and run both of them.   fifo\_r.c   1. #include <sys/types.h> 2. #include <sys/stat.h> 3. #include <fcntl.h> 4. #include <unistd.h> 5. #include <stdio.h> 7. **int** main() 8. { 9. **char** buf[1024]; 11. **int** fd\_r; 13. fd\_r = open("./myFIFO",O\_RDONLY); 14. **while** (1) 15. { 16. read(fd\_r,buf,1024); 17. printf("%s",buf); 18. } 19. close(fd\_r); 20. }   fifo\_w.c   1. #include <sys/types.h> 2. #include <sys/stat.h> 3. #include <fcntl.h> 4. #include <unistd.h> 5. #include <stdio.h> 7. **int** main() 8. { 9. **char** buf[1024]; 11. **int** fd\_w; 13. fd\_w = open("./myFIFO",O\_WRONLY); 14. **while** (1) 15. { 16. fgets(buf,1024,stdin); 17. write(fd\_w,buf,1024); 18. } 19. close(fd\_w); 20. } 21. Result     Task2:   1. Write the create\_pipe.c to create two pipes 2. #include <sys/types.h> 3. #include <sys/stat.h> 4. #include <stdio.h> 6. **int** main() 7. { 8. **int** fd1 = mkfifo("./f1",0644); 9. **int** fd2 = mkfifo("./f2",0644); 11. **if**(fd1 == 0 && fd2 == 0) 12. printf("Both of the pipes are created successfully!"); 13. **else** **if**(fd1 != 0 && fd2 == 0) 14. printf("fd1 created errorly!"); 15. **else** **if**(fd1 == 0 && fd2 != 0) 16. printf("fd2 created errorly!"); 17. **else** 18. printf("Both of the pipes are created errorly!"); 20. **return** 0; 21. } 22. And then write two c source file to contain two different processes which should have two thread. The threads in each process should imply reading and writing repeatedly by their own. When the reader thread or the writer thread receive “88\n”, the thread should terminate this whole process.   Process1.c   1. #include <sys/types.h> 2. #include <sys/stat.h> 3. #include <fcntl.h> 4. #include <unistd.h> 5. #include <stdio.h> 6. #include <pthread.h> 7. #include <string.h> 8. #include <stdlib.h> 10. **void** \*pwriter(**void** \*param) 11. { 13. **int** fd\_w = \*((**int**\*) param); 14. **char** buf[1024]={'\0'}; 15. **while** (1) 16. { 17. fgets(buf,1024,stdin); 18. write(fd\_w,buf,1024); 19. **if** (strcmp(buf,"88\n")==0) 20. { 21. **break**; 22. } 23. } 24. close(fd\_w); 25. exit(0); 26. } 28. **void** \*preader(**void** \*param) 29. { 31. **int** fd\_r = \*((**int** \*) param); 32. **char** buf[1024]={'\0'}; 33. **while** (1) 34. { 35. read(fd\_r,buf,1024); 36. **if** (strcmp(buf,"88\n")==0) 37. { 38. **break**; 39. } 40. printf("%s",buf); 41. } 42. close(fd\_r); 43. exit(0); 44. } 46. **int** main() 47. { 48. **int** fd\_w = open("./f1",O\_WRONLY); 49. **int** fd\_r = open("./f2",O\_RDONLY); 51. pthread\_t tid1,tid2; 52. pthread\_create(&tid1,NULL,pwriter,&fd\_w); 53. pthread\_create(&tid2,NULL,preader,&fd\_r); 54. pthread\_join(tid2,NULL); 55. pthread\_join(tid1,NULL); 56. }   Process2.c   1. #include <sys/types.h> 2. #include <sys/stat.h> 3. #include <fcntl.h> 4. #include <unistd.h> 5. #include <stdio.h> 6. #include <pthread.h> 7. #include <string.h> 8. #include <stdlib.h> 10. **void** \*pwriter(**void** \*param) 11. { 12. **int** fd\_w = \*((**int**\*) param); 13. **char** buf[1024]={'\0'}; 14. **while** (1) 15. { 16. fgets(buf,1024,stdin); 17. write(fd\_w,buf,1024); 18. **if** (strcmp(buf,"88\n")==0) 19. { 20. **break**; 21. } 22. } 23. close(fd\_w); 24. exit(0); 25. } 27. **void** \*preader(**void** \*param) 28. { 29. **int** fd\_r = \*((**int** \*) param); 30. **char** buf[1024]={'\0'}; 31. **while** (1) 32. { 33. read(fd\_r,buf,1024); 34. **if** (strcmp(buf,"88\n")==0) 35. { 36. **break**; 37. } 38. printf("%s",buf); 39. } 40. close(fd\_r); 41. exit(0); 42. } 44. **int** main() 45. { 46. **int** fd\_r = open("./f1",O\_RDONLY); 47. **int** fd\_w = open("./f2",O\_WRONLY); 49. pthread\_t tid1,tid2; 50. pthread\_create(&tid1,NULL,preader,&fd\_r); 51. pthread\_create(&tid2,NULL,pwriter,&fd\_w); 52. pthread\_join(tid1,NULL); 53. pthread\_join(tid2,NULL); 54. } 55. Result | | | | | |
| 1. **Result analysis and discussion**（Analysis of experimental results and summing up the harvest and the existing problems）   **Analysis of experimental results**  The experimental results are satisfying. In task1, typing strings in the writer process can be correctly showed in the reader process. In task2, typing strings in one process can be correctly showed in the other process. When the reader process or the writer process receives “88\n”, the process can be terminated.  **Harvest**  Pipes are an implementation of inter-process communication. With the idea that everything is a file in Linux, pipes are represented as a special kind of file in Linux. It is transferred in shell programs using "|". The main principle of pipes is that two file descriptors are obtained through this pipe, "0" for reading and "1" for writing, similar to reading and writing ordinary files.  There are two types of pipes, one is the normal pipe, which supports communication between parent and child processes and sibling processes. The normal pipe only allows one-way communication, and two normal pipes are needed for two-way communication. One type is the named pipe, this kind of pipe can be accessed by any process and does not require the existence of parent-child relationship, the named pipe is called FIFO in Linux. Although FIFO supports bi-directional communication, it only allows half-duplex transmission. If data is to be transmitted in two directions at the same time still need to create two FIFOs. Full-duplex communication support for named pipes in Windows.  Pipe file opening is blocked by default in Linux. When the open side is the read side, if there is no open write side, the read side will block until the write side with FIFO is opened. Similarly, when the open side is the write side, if there is no open read side, the write side will block until the read side with FIFO is open. When reading or writing from the FIFO, the read side remains until all open writes are closed. The same is true for the write side.  **Existing problems**  Both of processes have no hints to remind user pipes that be read or wrote. | | | | | |
| Comments & Evaluation | Content & Design (A-E) | | |  | |
| Procedure & Codes (A-E) | | |  | |
| Results (A-E) | | |  | |
| Analysis & Discussion (A-E) | | |  | |
| Score (A-E):  Feedback comments: | | | | |