|  |  |  |
| --- | --- | --- |
| Inter Process Communication | **LAB 06**  A look into shared memory and pipes in Linux | |
| **NATIONAL UNIVERSITY OF COMPUTER AND EMERGING SCIENCES**  **Spring 2023** | |  |

Contents

[Aim 3](#_Toc128455813)

[Objective 3](#_Toc128455814)

[Tools Needed 3](#_Toc128455815)

[Prior Knowledge 3](#_Toc128455816)

[Introduction 3](#_Toc128455817)

[Shared Memory 4](#_Toc128455818)

[System Calls Needed 4](#_Toc128455819)

[Producer and Consumer Code 5](#_Toc128455820)

[PIPES in Linux 10](#_Toc128455821)

[Unnamed Pipes 10](#_Toc128455822)

[Named Pipes 12](#_Toc128455823)

# Aim

Introduce some IPC mechanisms available in Linux namely, Shared Memory and Pipes.

# Objective

1. Successful implementation of shared memory mechanism to share data between two processes.
2. Understanding and using pipes (named and un-named) to send data between two processes.

# Tools Needed

1. An editor to write a C program (sublime, geany, vi, vim, gedit, VSCode etc.)
2. Linux Terminal (WSL, VMBox, Dual Boot) with GNU C (gcc compiler) installed.
3. A brilliant mind (such as one you possess) to focus and understand the purpose of this lab by completing it.

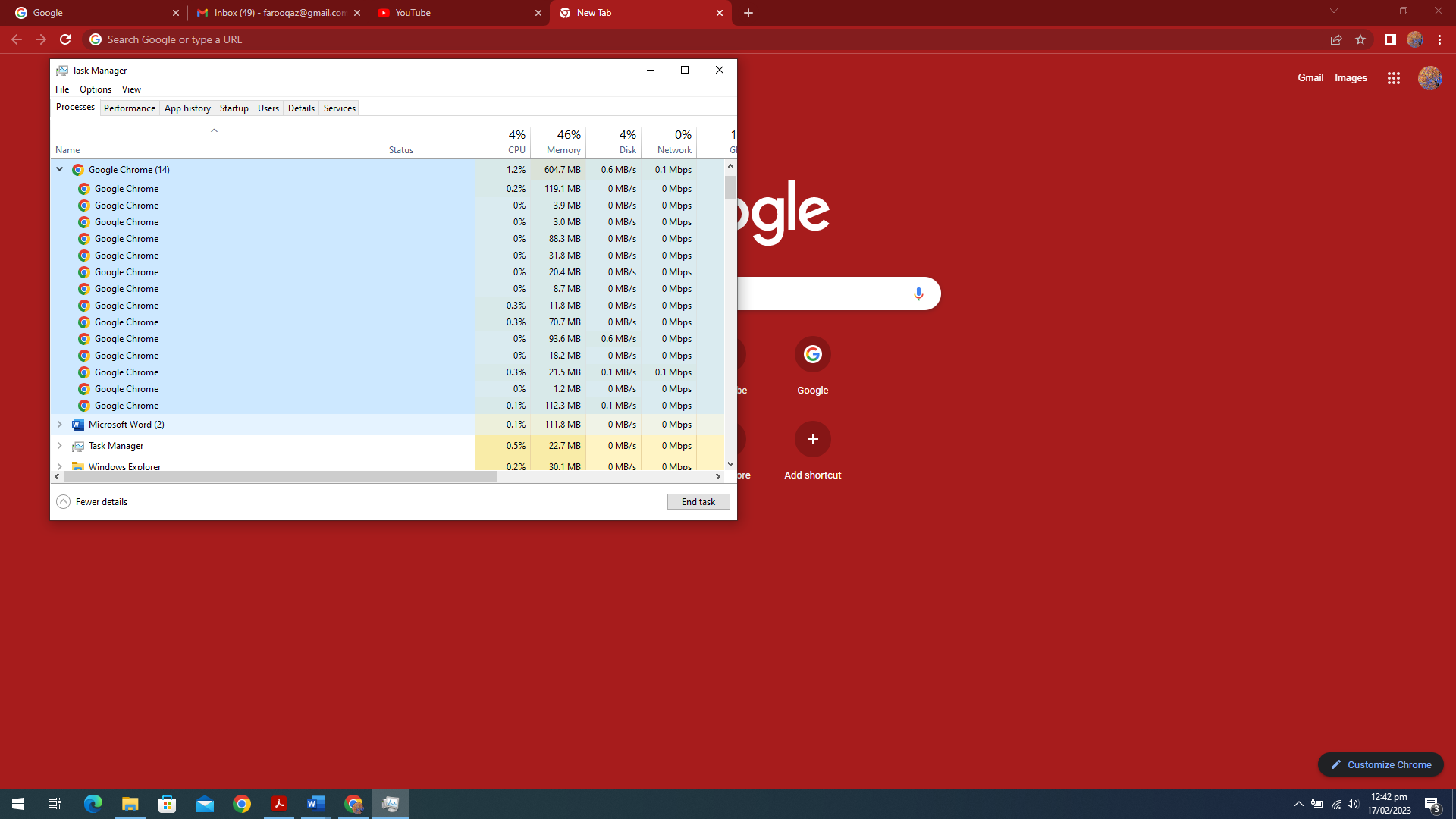
# Prior Knowledge

It is assumed that you have completed all previous tasks and have sound knowledge of: -

1. Using Linux terminal.
2. C language programming.
3. What are System Calls.

# Introduction

Big Applications usually run on several processes instead of one huge slow process. Check the screenshot below. For only 4 tabs of chrome there are 14 different processes running in windows.



Such applications require a means of communicating among its processes. Which OSes provide by several mechanisms of Inter process communication (IPC). In Linux we have: -

|  |  |  |
| --- | --- | --- |
| * + Pipes     1. Unnamed Pipe     2. Named Pipes | * + Signals   + Message Queue   + Shared Memory | * + Semaphores   + Remote procedure calls   + Sockets |

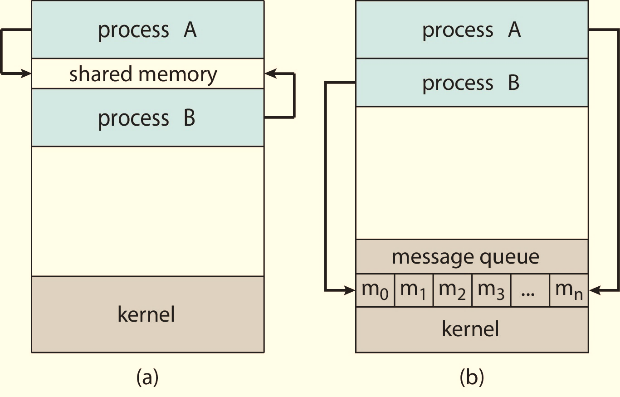
Let’s have a look at Shared Memory.

# Shared Memory

As the named suggests, IPC using Shared Memory is accomplished between 2 (or more process) by

* reading and writing to a portion of a memory
* which is not part of the process memory stack
* but both processes have access to it, as if it is part of its memory.

The job of the operating system (system calls) is to establish this shared region of memory and just keep a track of processes who are using it. Synchronization issues, such as race conditions (discussed in later chapters) are not the concern of the OS and the communicating processes must ensure this themselves. Although the support for this is provided by OS in the shape of Semaphores, mutexes etc. (again more details in later chapters.)



## System Calls Needed

For the Linux manual overview of shared memory type the following on the terminal or visit the link to [Online Man Pages](https://man7.org/linux/man-pages/man7/shm_overview.7.html):

https://man7.org/linux/man-pages/man7/shm\_overview.7.html

1. :~$ man shm\_overview

**💡Did you notice!** In the fine print under the Notes section

System V shared memory (shmget(2), shmop(2), etc.) is an older shared memory API. POSIX shared memory provides a simpler, and better designed interface; on the other hand, POSIX shared memory is somewhat less widely available (especially on older systems) than System V shared memory.

We will only be using the following system calls in this manual. The POSIX shared memory framework works as follows: -

|  |  |  |
| --- | --- | --- |
|  | STEPS | System Calls |
| **1** | All participating process must know the string name of the shared memory. (In our example it will be “/OS”) |  |
| **2** | One process must create a shared memory with this name and mention its size. | shm\_open,  ftruncate |
| **3** | All process must open the file descriptor for the shared memory.  (Creator process does this in same system call in which it created the shared memory). | shm\_open |
| **4** | All process must get the mapping in a pointer of the shared memory from the file descriptor opened in the above step. | mmap |
| **5** | Everyone uses the shared memory.  Synchronization is the responsibility of the programmer. |  |
| **6** | All process must un-map the pointer and close the file descriptor. | munmap,  close |
| **7** | At least 1 process must also delete the shared memory object | shm\_unlink |

## Producer and Consumer Code

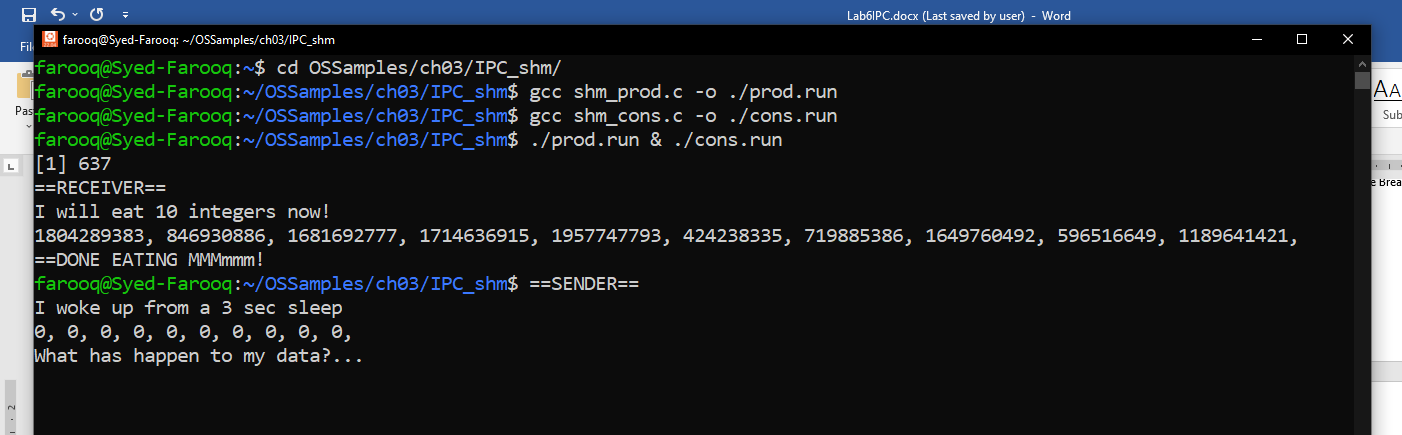
Code for the producer and consumer process for the Producer-Consumer Paradigm follows. Note that although the producer program must not read the buffer shared memory, we are doing so just to see whether the consumer was successful or not and that the consumer has set all the data to zero.

Similarly, the consumer must not write back into the buffer, but here we are reading and setting the data to zero so that the producer have some way of knowing that the data has been consumed.

Try coding in both the programs verbatim and then compile and run as shown in the screen shot.

1. /\* FILE: shm\_prod.c \*/
2. #include <stdio.h>
3. #include <stdlib.h>
4. #include <sys/shm.h>
5. #include <sys/mman.h>
6. #include <fcntl.h>
7. #include <sysexits.h>
8. #include <unistd.h>
9. /\* NOTE: These must be same on both prod and cons \*/
10. //--CONSTANTS--//
11. #define SHMNAME "/OS" //Name for our shared memory
12. #define SIZE (sizeof(int)\*10) //size of shared memory
13. //10 integers only.
14. /\*\*
15. \* @brief main function entry point
16. \* @param[in] argc The count of arguments
17. \* @param argv The arguments array
18. \* @return normal exit or error.
19. \*/
20. int main(int argc, char const \*argv[]){
21. //--Creating shared memory object--//
22. int shmObj = **shm\_open**(SHMNAME, O\_CREAT | O\_RDWR , 0600);
23. //--Ask OS to reserve the sharem memory for 10 integers--//
24. **ftruncate**(shmObj,SIZE);
25. //--Ask the OS to MAP the memory to a pointer--//
26. int \*ptrToShm = (int \*)**mmap**(0, SIZE, PROT\_WRITE | PROT\_READ,
27. MAP\_SHARED, shmObj, 0);
28. /\*NOTE:
29. \*Now you can use ptrToShm as if it were a buffer dynamically allocated and
30. \*read and write to it. Also we have explicitly cast the returned memory location
31. \*from mmap as pointer to int. It can be anything char, double, float, etc.
32. \*or any custom struct. But we will need to specifify the SIZE accordingly.
33. \*/
34. //--Writing 10 random numbers to our shared memory--//
35. for (int i = 0; i < 10; ++i) {
36. ptrToShm[i] = rand();
37. }
38. //--sleep for 3 sec hoping consumer uses these numbers--//
39. sleep(3);
40. //--Print the shared memory--//
41. printf("==SENDER==\nI woke up from a 3 sec sleep\n");
42. for (int i = 0; i < 10; ++i) {
43. printf("%d, ",ptrToShm[i]);
44. } printf("\nWhat has happen to my data?...\n");
45. //--Safely release and close the shared memory--//
46. **munmap**(ptrToShm, SIZE);
47. **close**(shmObj);
48. **shm\_unlink**(SHMNAME);
49. return EXIT\_SUCCESS;
50. }
51. /\* FILE: shm\_cons.c \*/
52. #include <stdio.h>
53. #include <stdlib.h>
54. #include <sys/shm.h>
55. #include <sys/mman.h>
56. #include <fcntl.h>
57. #include <sysexits.h>
58. #include <unistd.h>
59. /\* NOTE: These must be same on both prod and cons \*/
60. //--CONSTANTS--//
61. #define SHMNAME "/OS" //Name for our shared memory
62. #define SIZE (sizeof(int)\*10) //size for our shared memory
63. //10 integers only.
64. /\*\*
65. \* @brief main function entry point
66. \* @param[in] argc The count of arguments
67. \* @param argv The arguments array
68. \* @return normal exit or error.
69. \*/
70. int main(int argc, char const \*argv[]){
71. //--Creating shared memory object--//
72. /\*NOTE: Not asking for creating \*/
73. int shmObj = shm\_open(SHMNAME, O\_RDWR , 0666);
74. //-- CHECK ERROR IF NOT ALREADY CREATED BY PRODUCER--//
75. if(shmObj < 0){
76. perror("Error Opening Sharem Memory: shm.open()\n");
77. return EXIT\_FAILURE;
78. }
79. //--Ask OS to map the sharem memory for 10 integers--//
80. int \*ptrToShm = (int \*)mmap(0, SIZE, PROT\_WRITE | PROT\_READ,
81. MAP\_SHARED, shmObj, 0);
82. /\*NOTE:
83. \*Now you can use ptrToShm as if it were a buffer dynamically allocated and
84. \*read and write to it. Also we have explicitly cast the returned memory location
85. \*from mmap as pointer to int. It can be anything char, double, float, etc.
86. \*or any custom struct. But we will need to specifify the SIZE accordingly.
87. \*/
88. //--Reading 10 random numbers from our shared memory and setting them to 0--//
89. printf("==RECEIVER==\nI will eat 10 integers now!\n");
90. for (int i = 0; i < 10; ++i) {
91. printf("%d, ",ptrToShm[i]);
92. ptrToShm[i] = 0;
93. }printf("\n==DONE EATING MMMmmm!\n");
94. //--Safely release and close the shared memory--//
95. munmap(ptrToShm, SIZE);
96. close(shmObj);
97. return EXIT\_SUCCESS;
98. }

Try compiling and running both programs as shown in the screenshot below: -



You are encouraged to check the Linux man command for each system call to understand the parameters passed to each function call. Most of the parameters are access constants and/or mode of operations like read and write or create or what type of sharing should be allowed.

The shm\_open system call requires a string name (kind of like a filename). Each process participating in the communication must know this string name. Notice that when the producer uses this system call it does a bitwise OR ‘O\_CREAT | O\_RDWR’ this means that the system call will create a shared memory if it does not already exist.

The mmap system call has quite a lot of parameters because it can also be used to map files or other objects. Let’s look at them. These are also available with Linux man command.

**void \*mmap(void \****addr***, size\_t** *length***, int** *prot***, int** *flags***,** **int** *fd***, off\_t** *offset***);**

|  |  |
| --- | --- |
| Parameter | Description |
| void \**addr* | The starting address for the new mapping is specified in ***addr***.  If ***addr*** is NULL, then the kernel chooses the (page-aligned) address at which to create the mapping; this is the most portable method of creating a new mapping. If ***addr*** is not NULL, then the kernel takes it as a hint about where to place the mapping; |
| size\_t *length* | The ***length*** argument specifies the length of the mapping (which must be greater than 0). |
| int *prot* | The ***prot*** argument describes the desired memory protection of the mapping (and must not conflict with the open mode of the file in our case the shared memory).  It is either PROT\_NONE or the bitwise OR of one or more of the following flags:  PROT\_EXEC: Pages may be executed.  PROT\_READ: Pages may be read.  PROT\_WRITE: Pages may be written.  PROT\_NONE: Pages may not be accessed. |
| int *flags* | The ***flags*** argument determines whether updates to the mapping are visible to other processes mapping the same region, and whether updates are carried through to the underlying file.  There are many flag options. Check man or online for details.  For now, only MAP\_SHARED is of our concern. |
| int *fd* | The contents of a file mapping are initialized in the file (or other object) referred to by the file descriptor ***fd***.  In our case it is the shared memory descriptor created using the shm\_open. |
| off\_t *offset* | The contents of a file mapping, are initialized using ***length*** bytes starting at offset ***offset*** |
| RETURN VALUE | On success, **mmap()** returns a pointer to the mapped area. On error, the value MAP\_FAILED (that is, **(void \*) -1)** is returned, and errno global variable is set to indicate the error. |

The rest of the system calls are self-explanatory and easy.

# PIPES in Linux

Linux provides 2 flavors of pipes

1. Named and,
2. Unnamed.

We will look at both.

Pipes are message passing form of IPC. That is, all communications are handled by the operating systems and no direct access to the shared memory is given. Once again you are encouraged to visit the [linux man website for pipes](https://man7.org/linux/man-pages/man7/pipe.7.html). Here is a brief summary of the manual in bullet points: -

* Both named or unnamed are unidirectional IPC channels.
* The only difference between pipes and FIFOs (named pipes) is the manner in which they are created and opened. Once opened, I/O on pipes and FIFOs has exactly the same semantics.
* Attempt to read from an empty pipe ([read(2)](https://man7.org/linux/man-pages/man2/read.2.html)) blocks until data is available.
* Attempt to write to a full pipe ([write(2)](https://man7.org/linux/man-pages/man2/write.2.html)) blocks until sufficient data has been read from the pipe to allow the write to complete.
* The communication channel is a byte stream.
* A pipe has a limited capacity. Since Linux 2.6.35, the default pipe capacity is 16 pages, but the capacity can be queried and set using the fcntl(2) system call.

## Unnamed Pipes

Unnamed pipes are used in conjunction with fork calls. The pattern followed is to create and open a pipe then fork a child, so that the parent and child both know the file descriptors to use for communication.

**int pipe(int** *pipefd***[2]);**

“When a pipe call succeeds, zero is returned. On error, -1 is returned and the global errno is set to indicate the error, and pipefd is left unchanged. Check the manual for error numbers and their meanings.

The array ***pipefd*** is used to return two file descriptors referring to the ends of the pipe. ***pipefd[0]*** refers to the read end of the pipe. ***pipefd[1]*** refers to the write end of the pipe. Data written to the write end of the pipe is buffered by the kernel until it is read from the read end of the pipe.”

Below is a modified version of the example found in the man pages for understanding unnamed pipes.

1. #include <errno.h>
2. #include <stdio.h>
3. #include <stdlib.h>
4. #include <string.h>
5. #include <sys/types.h>
6. #include <sys/wait.h>
7. #include <unistd.h>
8. /\*\*
9. \* @brief Main function, entry point to the program
10. \* @param[in] argc The count of arguments
11. \* @param argv The arguments array
12. \* @return EXIT\_SUCCESS on safe exit
13. \*/
14. int main(int argc, char \*argv[]) {
15. int pipefd[2];
16. pid\_t cpid;
17. **if** (pipe(pipefd) == -1) exit(EXIT\_FAILURE);
18. cpid = fork();
19. **if** (cpid == -1) exit(EXIT\_FAILURE);
20. **if** (cpid == 0) { /\* Child the CONSUMER reads from pipe \*/
21. char msgRcv[15]; *// Recieve buffer.*
22. // We know 15 Bytes will be send, else it should be dynamic
23. close(pipefd[1]); /\* Close unused write end \*/
24. printf("I am %d child of parent %d**\n**", getpid(), getppid());
25. printf("I will attempt to read 15 bytes.**\n**");
26. read(pipefd[0], msgRcv, 15); //We know 15 bytes will be send.
27. //Not always the case. A little cheating in our toy example.
28. printf("MSG GOT: %s**\n**" msgRcv);
29. close(pipefd[0]);
30. exit(EXIT\_SUCCESS);
31. } **else** {/\* Parent writes HelloWorldHello to pipe \*/
32. close(pipefd[0]); /\* Close unused read end \*/
33. printf("I AM %d PARENT OF CHILD %d", getpid(), cpid);
34. write(pipefd[1], 'HelloWorldHello', 15); *// 15 bytes in our string*
35. close(pipefd[1]); // Reader will see EOF
36. wait(NULL); // Wait for child
37. exit(EXIT\_SUCCESS);
38. }
39. }

## Named Pipes

You are encouraged to read the FIFO (named piped) manual found at: <https://man7.org/linux/man-pages/man7/fifo.7.html>

Here is summary from the manual and things to remember: -

* Named Pipes can be opened by multiple processes for reading or writing.
* When processes are exchanging data via the FIFO, the kernel passes all data internally without writing it to the filesystem.
* The filesystem entry merely serves as a reference point so that processes can access the pipe using a name in the filesystem.
* NOTE: This is very important. Even with NVMe PCI Hard disks writing to a file is way slower.

**💡Article to Read:** [**Make Computers Run Faster**](https://www.m-techlaptops.com/ShopOnline/pc/How-to-boost-your-computers-speed-and-longevity-d67.htm)

“A normal hard disk has a response time of about 16ms, a good SSD will respond in 0.05ms, RAM will respond in 50ns. (Notice ms vs. ns) To put that in perspective, 0.05ms is equal to 50,000ns. This means that RAM can serve up data in memory 1000 times faster than a NVMe drive even though the file size they can carry is about the same.”

* The FIFO must be opened on both ends (reading and writing) before data can be passed. Normally, opening the FIFO blocks until the other end is opened also.

**int mkfifo(const char \****pathname***, mode\_t** *mode***);**

“mkfifo() system call makes a FIFO special file with name specified by pathname. Mode specifies the FIFO's permissions. It is modified by the process's umask in the usual way: the permissions of the created file are (mode & ~umask).

A FIFO special file is similar to a pipe, except that it is created in a different way. Instead of being an anonymous communications channel, a FIFO special file is entered into the filesystem by calling mkfifo().”

The basic framework to work with mkfifo is to write 2 separate programs. One producer and the other consumer. Both must share a path to a filename. Lets say in our case “OS.tmp”. The producer must make the call to mkfifo. The consumer should only open the symbolic file. Here is the code to a toy example of producer, consumer programs. Include appropriate libraries.

1. /\* File prod.c Producer Code \*/
2. #define NAME "OS"
3. #define SIZE 15
4. int main(int argc, char \*argv[]) {
5. char msg[15] = "HelloWorldHello";
6. int pipefd; //file descriptor to be used for our pipe
7. printf("I %d, am a PROD.**\n**I will create and open a named pipe now.**\n**",
8. getpid() );
9. **if**(mkfifo(NAME,0666)<0) {
10. perror("Error in call to mkfifo('OS',0666): ");
11. exit(EXIT\_FAILURE);
12. }
13. pipefd = open(NAME,O\_WRONLY);
14. printf("I will now write 'HelloWorldHello' 15 bytes**\n**");
15. write(pipefd, msg, SIZE); *// 15 bytes in our string*
16. printf("I %d have been awaken atlast. I will now die",getpid());
17. close(pipefd); // Reader will see EOF
18. exit(EXIT\_SUCCESS);
19. }

1. /\* file cons.c Consumer Code \*/
2. #define NAME "OS"
3. #define SIZE 15
4. int main(int argc, char \*argv[]) {
5. char msg[SIZE]; //char buffer to receive message
6. int pipefd; //file descriptor to be used for our pipe
7. printf("I %d, am a CONS. I will only open a named pipe now.**\n**", getpid());
8. pipefd = open(NAME,O\_RDONLY);
9. printf("I will now read 15 bytes from the pipe**\n**");
10. sleep(5);
11. read(pipefd, &msg, SIZE);
12. printf("MSG RECV: %s**\n** I will now die",msg);
13. close(pipefd); // Writer will see Read Complete
14. exit(EXIT\_SUCCESS);
15. }

Compile and run both programs in separate terminals. Make sure both are located in the same directory. We have given a relative path to our symbolic file “OS”.

**💡FOOD FOR THOUGHT:**

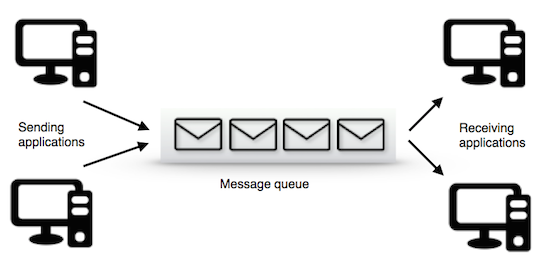
Try running the producer program again. What happens. How can you avoid this?

# Message Queue

Two or more processes can exchange information via access to a common system message queue. The sending process places via some (OS) message-passing module a message onto a queue which can be read by another process. Each message is given an identification or type so that processes can select the appropriate message. Process must share a common key in order to gain access to the queue in the first place.

Message queues provide an asynchronous way of communication possible, meaning that the sender and receiver of the message need not interact with the message queue at the same time. Message queue has a wide range of applications. Very simple applications can be taken as example here.

1. Taking input from the keyboard
2. To display output on the screen
3. Voltage reading from sensor etc.



A task which has to send the message can put message in the queue and other tasks. A message queue is a buffer-like object which can receive messages from ISRs (Interrupt Service Routine), tasks and the same can be transferred to other recipients. In short, it is like a pipeline. It can hold the messages sent by sender for a period until receiver reads it. And biggest advantage which someone can have in queue is receiver and sender need not use the queue on same time. Sender can come and post message in queue, receiver can read it whenever needed. Message queue basically composed of few components. A message queue should have a start and it should have an end as well. Starting point of a queue is referred as head of the queue and terminating point is called tail of the queue. Size of the queue has to be decided by the programmer while writing the code. And a queue cannot be read if it is empty. Meanwhile, a queue cannot be written into if it is already full. And a queue can have some empty elements as well.

The message queue can be implemented in Linux machine with available system calls. The basic operations to be carried out in queue are

* Creation/Deletion of queue
* Sending/Receiving of message

Two different files have to be written here: one for sender and another one for receiver. Receiver will wait until the sender writes into the queue. One important advantage with message queue is, it supports automatic synchronization between the sender and receiver. Receiver will wait until sender writes. Another advantage is memory can be freed after usage which is very essential in all software system.

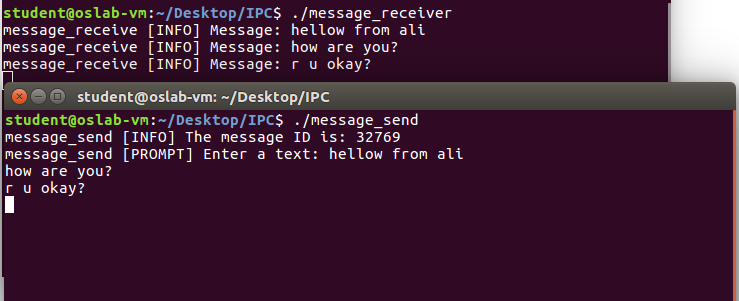
Few things can be taken into consideration before writing code for queue.

1. An Identifier has to be generated (key)
2. msgsnd() -> will initialize the queue.
3. msgrcv() -> will be used to receive the message
4. msgclt() -> Control action can be performed with this call i.e. deletion can be done with msgclt().

Below codes are for demonstrating message queue, you may face administrative privileges if not having while running these codes.



The execution is shown below.



So, the code should prompt the sender for typing the data to be sent to receiver. In parallel, from another terminal message\_rcv would receive all the information that sender types. If receiver compiles and executes first, program will wait until sender drops the message.

# References

https://man7.org/linux/man-pages/index.html