****

INFORMATION TECHNOLOGY AND MEDIA VIETNAM – KOREA UNIVERSITY

**FACULTY COMPUTER SCIENCE**

**SYSTEM PROGRAMMING REPORT**

**Implementing and Demo for FIFO in IPC**

**Member of group 2: Tran Quoc Bao ID: 22IT021**

**Ngo Van Bao ID:**

**Ho The Bao ID:**

**Lecturer: TS. Hoang Huu Duc**

***Da Nang, date 12 month 10 year 2024***

COMMENTS OF LECTURER

**Lecture’s signature**

ACKNOWLEDGMENTS

To complete this project, our group has received a great deal of support and assistance from our advisor – Dr. Hoang Huu Duc.

With heartfelt and sincere feelings, we would like to express our gratitude to her and to all our friends who helped us and collaborated in researching throughout the process of carrying out this project. Due to time constraints and our limited experience, this report inevitably has shortcomings. We kindly hope to receive guidance and feedback from the instructors so that we can further improve our experience, serving us better in future projects.

Our group sincerely thanks you!

**TABLE OF CONTENTS**

[COMMENTS OF LECTURER i](#_Toc181734858)

[ACKNOWLEDGMENTS ii](#_Toc181734859)

[INTRODUCTION 1](#_Toc181734860)

[CHAPTER 1: OVERVIEW OF THE TOPIC 2](#_Toc181734861)

[1.1 The purpose of the topic 2](#_Toc181734862)

[1.2 Tools and operating systems used for the topic 2](#_Toc181734863)

[1.2.1 Information about oracle vm virtualbox 2](#_Toc181734864)

[1.2.2 Operating systems Ubuntu/Linux 5](#_Toc181734865)

[1.2.2.1 Introduction to Ubuntu 5](#_Toc181734866)

[1.2.2.2 The different between Ubuntu and Linux 5](#_Toc181734867)

[1.2.2.3 Reasons why ubuntu is so popular for both computer and private servers 6](#_Toc181734868)

[1.2.2.4 A brief history lesson 10](#_Toc181734869)

[1.2.2.5 Opening a terminal 12](#_Toc181734870)

[1.2.2.6 Conclusion 16](#_Toc181734871)

[CHAPTER 2: THEORETICAL BASIS FOR “FIFO IN IPC” 16](#_Toc181734872)

[2.1 Inter Process Communication (IPC) 16](#_Toc181734873)

[2.1.1 Introduction about Inter Process Communication 16](#_Toc181734874)

[2.1.2 Types of Process, Examples of IPC systems 16](#_Toc181734875)

[2.1.2.1 Types of Process: 16](#_Toc181734876)

[2.1.2.2 Shared Memory 17](#_Toc181734877)

[2.1.2.3 Messaging Passing 19](#_Toc181734878)

[2.1.3 Synchronous and Asynchronous Message Passing 20](#_Toc181734879)

[2.1.4 Synchronous and Asynchronous Message Passing 21](#_Toc181734880)

[2.1.5 Examples of IPC systems 22](#_Toc181734881)

[2.1.6 Communication in Client/Server Architecture 22](#_Toc181734882)

[2.1.6 Role of Synchronization , Adventages and Disadvantages of IPC 23](#_Toc181734883)

[2.1.7 Conclusion 24](#_Toc181734884)

[2.2 Named Pipe or FIFO in IPC 24](#_Toc181734885)

[2.2.1 Introduction to FIFO 24](#_Toc181734886)

[2.2.1.1 Definition about FIFO and Characteristics of FIFO 24](#_Toc181734887)

[2.2.1.2 Use Cases of FIFO 25](#_Toc181734888)

[2.2.1.3 Benefits of FIFO 26](#_Toc181734889)

[2.2.1.4 Drawbacks of FIFO 27](#_Toc181734890)

[2.2.1.5 Conclusion 28](#_Toc181734891)

[2.2.2 System Calls and Functions for FIFO in Unix/Linux 28](#_Toc181734892)

[2.2.2.1 FIFO Creation and Permissions 28](#_Toc181734893)

[2.2.2.2 Open Modes 29](#_Toc181734894)

[2.2.2.3 Writing Data 31](#_Toc181734895)

[2.2.2.4 Reading Data 32](#_Toc181734896)

[2.2.2.5 Closing the file descriptor 33](#_Toc181734897)

[2.2.2.6 Removing the FIFO 35](#_Toc181734898)

[CHAPTER 3: DEMO FOR FIFO IN IPC 36](#_Toc181734899)

[3.1 Demo calculate the total amount of random generated 36](#_Toc181734900)

[3.2 Demo input a string uses FIFO 36](#_Toc181734901)

INTRODUCTION

In the context of the ever-evolving landscape of information technology, inter-process communication (IPC) plays an increasingly crucial role. IPC enables independent processes to exchange data and synchronize operations, facilitating the creation of complex and efficient applications. To ensure that data exchange occurs smoothly and efficiently, selecting the appropriate data flow management mechanism is essential.

FIFO (First In, First Out) is one of the most common data flow management mechanisms in IPC. Adhering to the principle of "first in, first out," FIFO ensures that requests are processed in the order they are received, preventing bottlenecks and data loss. However, implementing FIFO effectively in real-world IPC systems is not a simple task, especially when faced with demands for high performance, reliability, and security.

# CHAPTER 1: OVERVIEW OF THE TOPIC

## The purpose of the topic

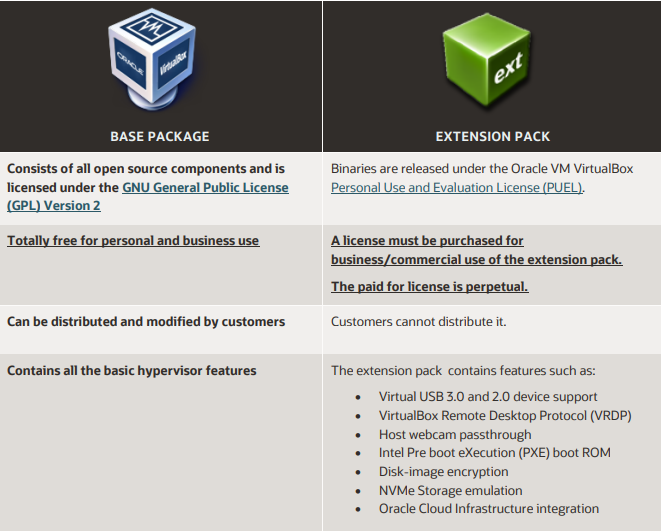
FIFO (First In First Out) is a fundamental mechanism for inter-process communication in Unix-like systems. Understanding FIFOs will help you:

* Build Efficient Multi-Process Applications:
* **Data Exchange:** FIFOs provide a simple, efficient channel for processes to exchange data, enabling the creation of complex applications.
* **Workflow Organization:** FIFOs help organize tasks in a specific order, ensuring that tasks are processed sequentially.
* **Performance Improvement:** By using FIFOs, you can optimize system resource utilization and enhance application execution speed.
* Deepen Your Understanding of Operating Systems:
* **Inter-Process Communication:** FIFOs are a core mechanism for process interaction. Studying FIFOs will help you grasp how processes communicate with each other.
* **Resource Management:** FIFOs play a vital role in managing system resources like CPU and memory.
* Solve Real-world Problems:
* **Pipeline Systems:** FIFOs are the foundation for creating pipeline systems where data is processed sequentially through multiple stages.
* **Messaging Systems:** FIFOs can be used to build simple messaging systems where processes can send and receive messages.
* **Process Synchronization:** FIFOs can be used to synchronize processes, ensuring that tasks are executed in the correct order.
* Expand Your Systems Programming Knowledge:
* **System Calls:** Working with FIFOs introduces you to system calls like mkfifo, open, read, and write, enhancing your understanding of system interactions.
* **Resource Management:** You'll learn how to efficiently manage system resources, preventing issues like deadlocks and race conditions.

## Tools and operating systems used for the topic

### Information about oracle vm virtualbox

**Introduction:** Oracle VM VirtualBox is cross-platform virtualization software. It allows users to extend their existing computer to run multiple operating systems including Microsoft Windows, Mac OS X, Linux, and Oracle Solaris, at the same time. Designed for IT professionals and developers, Oracle VM VirtualBox is ideal for testing, developing, demonstrating, and deploying solutions across multiple platforms from one machine. The following table summarizes each of the components:



Oracle VM VirtualBox has been designed to take advantage of the innovations introduced in the x86 modern hardware platform, and it is lightweight and easy to install and use. Yet, under the simple exterior lies an extremely fast and powerful virtualization engine. With a well-earned reputation for speed and agility, Oracle VM VirtualBox contains innovative features to deliver tangible benefits: excellent performance; a powerful virtualization system; and a wide range of supported guest operating systems.

Oracle VM VirtualBox is a bridge to open source and cloud development. The latest release allows users to create and deploy virtual machines nearly everywhere, upload to the cloud, download from the cloud, and review and make changes offline.

With thousands of downloads each day, Oracle VM VirtualBox is the world’s most popular free and open source, cross-platform virtualization software, based on vibrant community participation combined with world-class development and support supplied by Oracle.

A box with a logo

Description automatically generatedOracle VM VirtualBox simplifies cloud deployment by allowing developers to create multiplatform environments and to develop applications for container and virtualization technologies within Oracle VM VirtualBox on a single machine. Operating system and application updates can be done within Oracle VM VirtualBox virtual machines (VMs), and VMs can subsequently be deployed to server virtualization environments such as Oracle Linux KVM or Oracle Private Cloud Appliance.

Oracle VM VirtualBox Enterprise is an ideal choice for a next-generation development solution. The latest release introduces paravirtualization support for Linux and Windows virtual machines and support for xHCI/USB 3.0 devices and new platforms, and it provides enhanced CPU capabilities and support for bidirectional drag and drop between a host and its guest virtual machines. It also introduces disk-image encryption and many other enhancements.

Oracle VM VirtualBox Enterprise provides world-class support for both the base package and the extension pack and licenses for commercial use of the extension pack.

**Oracle VM VirtualBox Enterprise use cases:**

Development platform for the cloud. Software developers rely on Oracle VM VirtualBox Enterprise for the development and debugging of their applications in multiple operating systems and environments on one device. Developers can clone an environment on their personal desktop/laptop without impact to production services.

A computer screen with cloud computing and cloud computing

Description automatically generated

### Operating systems Ubuntu/Linux

#### 1.2.2.1 Introduction to Ubuntu

Ubuntu is a popular free and open-source Linux-based operating system you can use on a computer or virtual private server. Ubuntu was introduced in 2004 by a British company Canonical. It was based on Debian – a popular distro back then – which was difficult to install. As a result, Ubuntu was proposed as a more user-friendly alternative. Ubuntu has multiple editions, including core, server, and desktop, that allow it to run across different types of machines. It can be used on personal computers, servers, supercomputers, in cloud computing, and more.

#### 1.2.2.2 The different between Ubuntu and Linux

Linux is a family of operating systems based on the Linux kernel – the core of an operating system. It enables the communication between hardware and software components.

Linux is based on Unix and built around the Linux kernel. It was released in 1991 and is available for web servers, gaming consoles, embedded systems, desktops, and personal computers. It comes in many different versions called distributions.

Ubuntu is a Linux distro based on Debian. It is suitable for cloud computing, servers, desktops, and internet of things (IoT) devices. The main difference between Linux and Ubuntu is that the former is an operating system family based on Unix, while Ubuntu is a Linux distribution.

#### 1.2.2.3 Reasons why ubuntu is so popular for both computer and private servers

* **User-Friendliness:** Ubuntu uses Linux desktop environments for its interface. Since Ubuntu 17.10, GNOME is the default one. GNOME doesn’t clutter the screen with descriptions, instead using icons to facilitate navigation.

By default, GNOME features the Activities panel on the left taskbar.

**A screenshot of a computer

Description automatically generated**

Controls are situated on the top-right corner of the screen.

A screenshot of a computer

Description automatically generated

A full applications overview can be viewed by clicking the grid button on the bottom-left corner of the screen.

A screenshot of a computer

Description automatically generated

System navigation is made easy because all configuration and application elements are accessible from the main screen.

* **Strong Security:** Ubuntu is open-source, undergoing constant checks and reviews by its community members. As a result, any security vulnerabilities can be identified and eliminated quickly. Generally, Linux distributions have fewer security flaws compared to other operating systems.

What’s more, Ubuntu employs AppArmor, a kernel enhancement that restricts how programs behave and limits their resources. It works when you have profiles inserted into the kernel.

These consist of text files containing access rules for each application. AppArmor can mitigate the extent of security breaches since programs don’t have unlimited permissions.

In addition, there are a number of security practices Ubuntu supports, such as automatically installing security updates, using sudo instead of root [Linux user](https://www.hostinger.com/tutorials/how-to-see-system-users-in-ubuntu-linux-vps/), implementing complex passwords, [setting up a VPN server](https://www.hostinger.com/tutorials/how-to-set-up-a-linux-vpn-server-with-openvpn/), configuring [firewall using ufw](https://www.hostinger.com/tutorials/how-to-configure-firewall-on-ubuntu-using-ufw/), and enabling [iptables](https://www.hostinger.com/tutorials/iptables-tutorial).

* **More Software Options:** Most of the popular macOS and Windows applications such as Slack, Spotify, and Firefox are also available for Linux users and can be installed via the Ubuntu Software Center. Even if you can’t find the application you want, chances are there is a quality alternative available. For example, Libre Office works just as well as Microsoft Office.

A screenshot of a computer

Description automatically generated

Another option you can use aside from the Ubuntu Software Center is Snapcraft. It is an application created by Canonical that contains open-source and proprietary [software packages](https://www.hostinger.com/tutorials/how-to-list-installed-packages-on-ubuntu/) available for Linux-based operating systems. One major advantage of Snapcraft is that it uses the snapd daemon that automatically checks and updates applications.

* **Enhanced Privacy:** Just like any other operating system, Ubuntu has its data privacy policy.

There are four fundamental principles that Ubuntu follows in terms of personal information processing:

* Ubuntu doesn’t ask for personal data unless it truly needs such information for legal purposes.
* Ubuntu doesn’t share its users’ personal information with anyone except to provide its customers with products and services, comply with the law, and protect its rights.
* Ubuntu doesn’t store personal information except if it is required for the operation of services, to provide products, comply with the law, or protect its rights.

Ubuntu also collects some hardware information as well as location and usage data. However, you can always stop it from doing so. For example, location services can be disabled via the Privacy settings, as shown below.

A screenshot of a computer

Description automatically generated

* **Lightweight Performance:** Ubuntu is not resource-intensive – it operates smoothly on low-end devices. The default interface can run on less than 1 GB of RAM. What’s more, a lot of Ubuntu desktop environments are even more lightweight. For example, Lubuntu can run on systems with as little as 512 MB of RAM.

In comparison, both Windows and macOS require considerably more resources – both macOS Big Sur and Windows 11 need a minimum of 4 GB of RAM to run. This is because these operating systems have resource-heavy user interfaces (UIs) with advanced features incorporated in them.

* **Free of Charge:** Ubuntu is a free open-source operating system that you can download from its official website. You can also modify its source code as you see fit – as of now, there are numerous projects based on Ubuntu.

In comparison, macOS and Windows are closed-source operating systems. To use Windows, you need to buy a computer that comes with it or purchase a license which starts at $139/license. Meanwhile, macOS is not available for purchase – it comes pre-installed with Mac devices.

#### 1.2.2.4 A brief history lesson

During the formative years of the computer industry, one of the early operating systems was called Unix. It was designed to run as a multi-user system on mainframe computers, with users connecting to it remotely via individual ***terminals***. These terminals were pretty basic by modern standards: just a keyboard and screen, with no power to run programs locally. Instead they would just send keystrokes to the server and display any data they received on the screen. There was no mouse, no fancy graphics, not even any choice of colour. Everything was sent as text, and received as text. Obviously, therefore, any programs that ran on the mainframe had to produce text as an output and accept text as an input.

Compared with graphics, text is very light on resources. Even on machines from the 1970s, running hundreds of terminals across glacially slow network connections (by today’s standards), users were still able to interact with programs quickly and efficiently. The commands were also kept very terse to reduce the number of keystrokes needed, speeding up people’s use of the terminal even more. This speed and efficiency is one reason why this text interface is still widely used today.

When logged into a Unix mainframe via a terminal users still had to manage the sort of file management tasks that you might now perform with a mouse and a couple of windows. Whether creating files, renaming them, putting them into subdirectories or moving them around on disk, users in the 70s could do everything entirely with a textual interface.

Each of these tasks required its own program or command: one to change directories (cd), another to list their contents (ls), a third to rename or move files (mv), and so on. In order to coordinate the execution of each of these programs, the user would connect to one single master program that could then be used to launch any of the others. By wrapping the user’s commands this “shell” program, as it was known, could provide common capabilities to any of them, such as the ability to pass data from one command straight into another, or to use special wildcard characters to work with lots of similarly named files at once. Users could even write simple code (called “shell scripts”) which could be used to automate long series of shell commands in order to make complex tasks easier. The original Unix shell program was just called sh, but it has been extended and superceded over the years, so on a modern Linux system you’re most likely to be using a shell called bash. Don’t worry too much about which shell you have, all the content in this tutorial will work on just about all of them.

Linux is a sort-of-descendent of Unix. The core part of Linux is designed to behave similarly to a Unix system, such that most of the old shells and other text-based programs run on it quite happily. In theory you could even hook up one of those old 1970s terminals to a modern Linux box, and access the shell through that. But these days it’s far more common to use a software terminal: that same old Unix-style text interface, but running in a window alongside your graphical programs.

#### 1.2.2.5 Opening a terminal

On a Ubuntu 18.04 system you can find a launcher for the terminal by clicking on the Activities item at the top left of the screen, then typing the first few letters of “terminal”, “command”, “prompt” or “shell”. Yes, the developers have set up the launcher with all the most common synonyms, so you should have no problems finding it.

A screen shot of a computer

Description automatically generated

Other versions of Linux, or other flavours of Ubuntu, will usually have a terminal launcher located in the same place as your other application launchers. It might be hidden away in a submenu or you might have to search for it from within your launcher, but it’s likely to be there somewhere.

If you can’t find a launcher, or if you just want a faster way to bring up the terminal, most Linux systems use the same default keyboard shortcut to start it: **Ctrl-Alt-T**.

However you launch your terminal, you should end up with a rather dull looking window with an odd bit of text at the top, much like the image below. Depending on your Linux system the colours may not be the same, and the text will likely say something different, but the general layout of a window with a large (mostly empty) text area should be similar.

A screenshot of a computer

Description automatically generated

Let’s run our first command. Click the mouse into the window to make sure that’s where your keystrokes will go, then type the following command, ***all in lower case***, before pressing the **Enter** or **Return** key to run it.

****

You should see a directory path printed out (probably something like /home/YOUR\_USERNAME), then another copy of that odd bit of text.

A screenshot of a computer

Description automatically generated

There are a couple of basics to understand here, before we get into the detail of what the command actually did. First is that when you type a command it appears on the same line as the odd text. That text is there to tell you the computer is ready to accept a command, it’s the computer’s way of prompting you. In fact it’s usually referred to as the *prompt*, and you might sometimes see instructions that say “bring up a prompt”, “open a command prompt”, “at the bash prompt” or similar. They’re all just different ways of asking you to open a terminal to get to a shell.

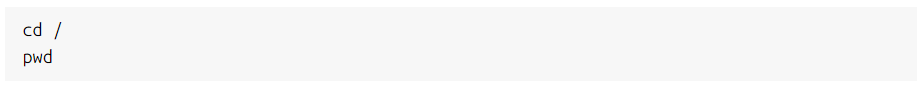
On the subject of synonyms, another way of looking at the prompt is to say that there’s a line in the terminal into which you type commands. A command line, if you will. Again, if you see mention of “command line”, including in the title of this very tutorial, it’s just another way of talking about a shell running in a terminal.

The second thing to understand is that when you run a command any output it produces will usually be printed directly in the terminal, then you’ll be shown another prompt once it’s finished. Some commands can output a lot of text, others will operate silently and won’t output anything at all. Don’t be alarmed if you run a command and another prompt immediately appears, as that usually means the command succeeded. If you think back to the slow network connections of our 1970s terminals, those early programmers decided that if everything went okay they may as well save a few precious bytes of data transfer by not saying anything at all.

**The importance of case:** Be extra careful with case when typing in the command line. Typing PWD instead of pwd will produce an error, but sometimes the wrong case can result in a command appearing to run, but not doing what you expected. We’ll look at case a little more on the next page but, for now, just make sure to type all the following lines in exactly the case that’s shown.

**A sense of location:** One important concept to understand is that the shell has a notion of a default location in which any file operations will take place. This is its working directory. If you try to create new files or directories, view existing files, or even delete them, the shell will assume you’re looking for them in the current working directory unless you take steps to specify otherwise. So it’s quite important to keep an idea of what directory the shell is “in” at any given time, after all, deleting files from the wrong directory could be disastrous. If you’re ever in any doubt, the pwd command will tell you exactly what the current working directory is.

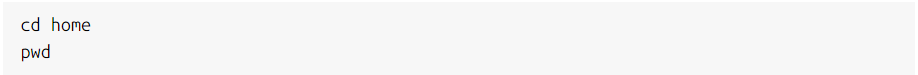
You can change the working directory using the cd command, an abbreviation for ‘change directory’. Try typing the following:



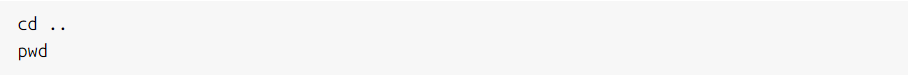
*Note: Note that the directory separator is a forward slash (“/”), not the backslash that you may be used to from Windows or DOS systems*

Now your working directory is “/”. If you’re coming from a Windows background you’re probably used to each drive having its own letter, with your main hard drive typically being “C:”. Unix-like systems don’t split up the drives like that. Instead they have a single unified file system, and individual drives can be attached (“mounted”) to whatever location in the file system makes most sense. The “/” directory, often referred to as the root directory, is the base of that unified file system. From there everything else branches out to form a tree of directories and subdirectories.

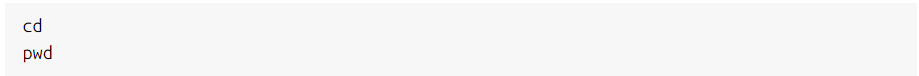
From the root directory, the following command will move you into the “home” directory (which is an immediate subdirectory of “/”):



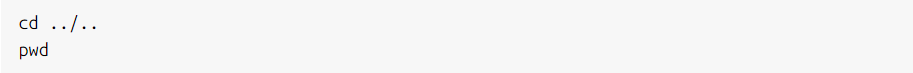
To go up to the parent directory, in this case back to “/”, use the special syntax of two dots (..) when changing directory (note the space between cd and .., unlike in DOS you can’t just type cd.. as one command):



Typing cd on its own is a quick shortcut to get back to your home directory:



You can also use .. more than once if you have to move up through multiple levels of parent directories:



Notice that in the previous example we described a route to take through the directories. The path we used means “starting from the working directory, move to the parent / from that new location move to the parent again”. So if we wanted to go straight from our home directory to the “etc” directory (which is directly inside the root of the file system), we could use this approach:

A white background with a black and white flag

Description automatically generated with medium confidence

#### 1.2.2.6 Conclusion

This section has only been a brief introduction to the Linux command line. We’ve looked at a few common commands for moving around the file system and manipulating files, but no tutorial could hope to provide a comprehensive guide to every available command. What’s more important is that you’ve learnt the key aspects of working with the shell. You’ve been introduced to some widely used terminology (and synonyms) that you might come across online, and have gained an insight into some of the key parts of a typical shell command.

# CHAPTER 2: THEORETICAL BASIS FOR “FIFO IN IPC”

## 2.1 Inter Process Communication (IPC)

### 2.1.1 Introduction about Inter Process Communication

Processes can coordinate and interact with one another using a method called inter-process communication (IPC) . Through facilitating process collaboration, it significantly contributes to improving the effectiveness, modularity, and ease of software systems.

### Types of Process, Examples of IPC systems

#### 2.1.2.1 Types of Process:

* Independent process
* Co-operating process

An independent process is not affected by the execution of other processes while a co-operating process can be affected by other executing processes. Though one can think that those processes, which are running independently, will execute very efficiently, in reality, there are many situations when cooperative nature can be utilized for increasing computational speed, convenience, and modularity. Inter-process communication (IPC) is a mechanism that allows processes to communicate with each other and synchronize their actions. The communication between these processes can be seen as a method of cooperation between them. Processes can communicate with each other through both: Methods of IPC

* Shared Memory
* Message Passing

An operating system can implement both methods of communication. First, we will discuss the shared memory methods of communication and then message passing. Communication between processes using shared memory requires processes to share some variable, and it completely depends on how the programmer will implement it. One way of communication using shared memory can be imagined like this: Suppose process1 and process2 are executing simultaneously, and they share some resources or use some information from another process. Process1 generates information about certain computations or resources being used and keeps it as a record in shared memory. When process2 needs to use the shared information, it will check in the record stored in shared memory and take note of the information generated by process1 and act accordingly. Processes can use shared memory for extracting information as a record from another process as well as for delivering any specific information to other processes.

A screenshot of a computer

Description automatically generated

#### 2.1.2.2 Shared Memory

Ex: Producer-Consumer problem

There are two processes: Producer and Consumer . The producer produces some items and the Consumer consumes that item. The two processes share a common space or memory location known as a buffer where the item produced by the Producer is stored and from which the Consumer consumes the item if needed. There are two versions of this problem: the first one is known as the unbounded buffer problem in which the Producer can keep on producing items and there is no limit on the size of the buffer, the second one is known as the bounded buffer problem in which the Producer can produce up to a certain number of items before it starts waiting for Consumer to consume it. We will discuss the bounded buffer problem. First, the Producer and the Consumer will share some common memory, then the producer will start producing items. If the total produced item is equal to the size of the buffer, the producer will wait to get it consumed by the Consumer. Similarly, the consumer will first check for the availability of the item. If no item is available, the Consumer will wait for the Producer to produce it. If there are items available, Consumer will consume them. The pseudo-code to demonstrate is provided below:

Shared Data Between the two Processes

A screenshot of a computer program

Description automatically generated

Producer Process Code

A screenshot of a computer program

Description automatically generated

Consumer Process Code

A screenshot of a computer program

Description automatically generated

#### 2.1.2.3 Messaging Passing

In this method, processes communicate with each other without using any kind of shared memory. If two processes p1 and p2 want to communicate with each other, they proceed as follows:

* Establish a communication link (if a link already exists, no need to establish it again.).
* Start exchanging messages using basic primitives.

We need at least two primitives:

– send (message, destination) or send (message).

– receive (message, host) or receive (message).

A green rectangular box with white text

Description automatically generated with medium confidence

The message size can be of fixed size or of variable size. If it is of fixed size, it is easy for an OS designer but complicated for a programmer and if it is of variable size then it is easy for a programmer but complicated for the OS designer. A standard message can have two parts: header and body. The header part is used for storing message type, destination id, source id, message length, and control information. The control information contains information like what to do if runs out of buffer space, sequence number, priority. Generally, message is sent using **FIFO** style.

### Synchronous and Asynchronous Message Passing

A link has some capacity that determines the number of messages that can reside in it temporarily for which every link has a queue associated with it which can be of zero capacity, bounded capacity, or unbounded capacity. In zero capacity, the sender waits until the receiver informs the sender that it has received the message. In non-zero capacity cases, a process does not know whether a message has been received or not after the send operation. For this, the sender must communicate with the receiver explicitly. Implementation of the link depends on the situation, it can be either a direct communication link or an in-directed communication link.

Direct Communication links are implemented when the processes use a specific process identifier for the communication, but it is hard to identify the sender ahead of time.

For example the print server.

In-direct Communication is done via a shared mailbox (port), which consists of a queue of messages. The sender keeps the message in mailbox and the receiver picks them up.

### Synchronous and Asynchronous Message Passing

A process that is blocked is one that is waiting for some event, such as a resource becoming available or the completion of an I/O operation. IPC is possible between the processes on same computer as well as on the processes running on different computer i.e. in networked/distributed system. In both cases, the process may or may not be blocked while sending a message or attempting to receive a message so message passing may be blocking or non-blocking. Blocking is considered synchronous and blocking send means the sender will be blocked until the message is received by receiver. Similarly, blocking receive has the receiver block until a message is available. Non-blocking is considered asynchronous and Non-blocking send has the sender sends the message and continue. Similarly, Non-blocking receive has the receiver receive a valid message or null. After a careful analysis, we can come to a conclusion that for a sender it is more natural to be non-blocking after message passing as there may be a need to send the message to different processes. However, the sender expects acknowledgment from the receiver in case the send fails. Similarly, it is more natural for a receiver to be blocking after issuing the receive as the information from the received message may be used for further execution. At the same time, if the message send keep on failing, the receiver will have to wait indefinitely. That is why we also consider the other possibility of message passing.

There are basically three preferred combinations:

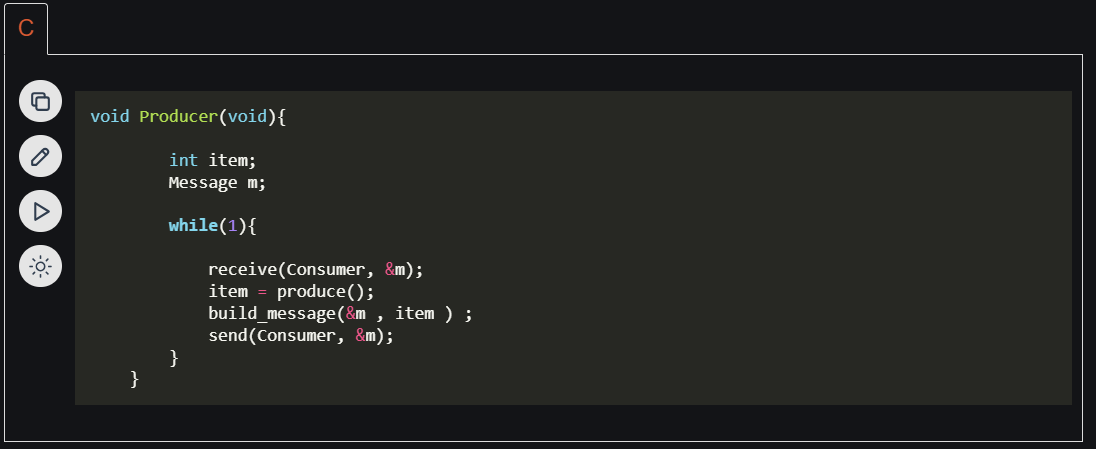
* Blocking send and blocking receive
* Non-blocking send and Non-blocking receive
* Non-blocking send and Blocking receive (Mostly used)

Ex: send(p1, message) means send the message to p1.

Similarly, receive(p2, message) means to receive the message from p2.

In this method of communication, the communication link gets established automatically, which can be either unidirectional or bidirectional, but one link can be used between one pair of the sender and receiver and one pair of sender and receiver should not possess more than one pair of links. Symmetry and asymmetry between sending and receiving can also be implemented i.e. either both processes will name each other for sending and receiving the messages or only the sender will name the receiver for sending the message and there is no need for the receiver for naming the sender for receiving the message. The problem with this method of communication is that if the name of one process changes, this method will not work.

Producer Code



Consumer Code

A screen shot of a computer

Description automatically generated

### Examples of IPC systems

* Posix: uses shared memory method.
* Mac : uses message passing.
* Windows XP: uses message passing using local procedural calls

### 2.1.6 Communication in Client/Server Architecture

There are various mechanism:

* Pipe
* Socket
* Remote Procedural calls (RPCs)

### Role of Synchronization , Adventages and Disadvantages of IPC

In Inter-Process Communication (IPC) on Unix systems, synchronization is crucial to ensure that multiple processes can interact effectively and avoid race conditions or deadlocks. It involves coordinating the access of processes to shared resources, such as memory or files.

Synchronization mechanisms in IPC typically include:

* **Semaphores:** These are integer variables that are used to control access to shared resources. A semaphore's value can be incremented (P operation) or decremented (V operation) atomically, ensuring that only one process can access the resource at a time.
* **Mutexes:** Mutual exclusion locks are used to protect critical sections of code that access shared data. Only one process can hold the mutex at a time, preventing other processes from entering the critical section until the mutex is released.
* **Message queues:** These are used to send and receive messages between processes. Synchronization is often employed to ensure that messages are not lost or duplicated.
* **Shared memory:** When processes share a region of memory, synchronization is necessary to avoid conflicts and ensure data consistency. Semaphores or mutexes can be used to control access to the shared memory.

**Advantages of IPC:**

* **Efficient communication:** IPC mechanisms can provide efficient communication between processes, reducing overhead compared to other methods like sockets.
* **Shared resources:** IPC allows processes to share resources, such as memory or files, which can improve performance and reduce memory usage.
* **Process coordination:** IPC can be used to coordinate the activities of multiple processes, enabling them to work together to achieve a common goal.
* **Flexibility:** IPC offers a variety of mechanisms, each with its own strengths and weaknesses, allowing you to choose the most appropriate method for your specific needs.

**Disadvantages of IPC:**

* Increases system complexity, making it harder to design, implement, and debug.

1. Can introduce security vulnerabilities, as processes may be able to access or modify data belonging to other processes.
2. Requires careful management of system resources, such as memory and [CPU](https://www.geeksforgeeks.org/difference-between-cpu-and-gpu/)time, to ensure that IPC operations don’t degrade overall system performance.  
   Can lead to data inconsistencies if multiple processes try to access or modify the same data at the same time.
3. Overall, the advantages of IPC outweigh the disadvantages, as it is a necessary mechanism for modern operating systems and enables processes to work together and share resources in a flexible and efficient manner. However, care must be taken to design and implement IPC systems carefully, in order to avoid potential security vulnerabilities and performance issues.

### Conclusion

A fundamental component of contemporary operating systems, IPC allows processes to efficiently coordinate operations, share resources, and communicate. IPC is beneficial for developing adaptable and effective systems, despite its complexity and possible security threats.

## Named Pipe or FIFO in IPC

### 2.2.1 Introduction to FIFO

#### 2.2.1.1 Definition about FIFO and Characteristics of FIFO

Named pipes, also referred to as FIFOs (First In, First Out), constitute essential IPC systems in software systems. They offer a quick and effective method for successfully transferring information between processes. Specialized kinds of files known as named pipes serve as a means for interaction among unconnected procedures that operate on an identical structure as well as on separate ones.

First-in, first-out (FIFO) named pipes ensure that information composed to the line by a single procedure is read from the pipe by another course in the identical order. Therefore, They are particularly advantageous when processes must communicate independently without sharing storage or direct dependency management.

A named pipe is an extension of the traditional pipe concept in Unix. While a traditional pipe is “unnamed” and exists only temporarily, a named pipe can persist as long as the system is up or until it is explicitly deleted. Named pipes appear as special files in the filesystem, and multiple processes can attach to them for reading and writing, facilitating inter-process communication.

A FIFO file allows two or more processes to communicate by reading from and writing to the same file. This file type is created using the ‘mkfifo()’ system call in C. Once created, any process can open the named pipe for reading or writing, similar to how it would handle an ordinary file. However, it is important to note that a named pipe must be opened simultaneously at both ends (for reading and writing) before any input or output operations can occur.

#### 2.2.1.2 Use Cases of FIFO

Let us look at some use cases for Named Pipes or FIFOs below:

**Interprocess Communication**− Named pipes are commonly used for interprocess communication (IPC) between independent processes. They provide a simple and efficient way for processes to exchange data without sharing memory or having direct dependencies. Named pipes enable communication between processes running on the same machine or even across different systems when they have access to a shared file system.

**Client-Server Communication**− Named pipes are often employed in client-server architectures to establish communication channels between clients and servers. The server creates a named pipe and listens for client requests, while clients can connect to the server by opening the named pipe. This allows for seamless communication and data exchange between clients and the server.

**Shell Scripting**− Named pipes can be used in shell scripting to connect different commands or processes. By using named pipes, the output of one command can be piped into another command as input, enabling the chaining of multiple commands together. This provides flexibility in scripting and allows for the efficient flow of data between different components.

**Data Streaming and Logging**− Named pipes are useful for streaming data or log files between processes. For example, a process generating log information can write the logs to a named pipe, and another process can read from the pipe and perform further processing or storage of the log data. This allows for real-time processing and analysis of log information.

**Distributed Computing**− Named pipes can facilitate communication between processes in distributed computing environments. In scenarios where multiple processes are running on different machines and need to exchange data, named pipes can provide a reliable and efficient communication mechanism. They can be used to transmit data between distributed components, such as in distributed data processing frameworks or parallel computing applications.

**System Monitoring and Control**− Named pipes can be utilized in system monitoring and control applications. For instance, a monitoring process can write system metrics to a named pipe, and other processes or systems can read from the pipe to collect and analyze the metrics. This enables efficient monitoring and control of system resources and performance.

#### 2.2.1.3 Benefits of FIFO

The following benefits of named pipes (FIFOs) for inter-process communication

* **Simple and easy to use**− Named pipes offer a simple and uncomplicated method for communication between processes. They operate on a straightforward FIFO model when information created by a single procedure is retrieved through another procedure in the exact same order.
* **Process independence**− Operations with named pipes can communicate with one another. The procedures can talk to one another with no previous understanding or reliance so long as they're granted the right authorization for using the named pipe.
* **Interprocess communication**− When a named pipe exists in a file system that is shared that both systems can access, it allows interaction among procedures that are operating on the exact same machine as well as among procedures that are operating on various systems.
* **Efficiency**− Named pipes are a productive way to communicate between processes. They can effectively handle enormous quantities of data and depend on a buffering system.

#### 2.2.1.4 Drawbacks of FIFO

The following drawbacks of named pipes (FIFOs) for inter-process communication

* **Unidirectional communication**− Named pipes support a one-way exchange from the writer's perspective to the audience member or the reverse because they are unidirectional.
* **Limited functionality**− Named pipes offer a straightforward method of transmitting information and are comparatively simple. Advanced features like communication limitations, structures for data, and complex data serialization are not supported by them.
* **Blocking behavior**− Both reading and writing activities on named pipes are perpetually blocked by default. A program will be restricted until the information is accessible or time is freed if it attempts to read from or write to a devoid pipe or a pipe that is full.
* **Limited error handling**− The embedded abilities of named pipes could be improved. Additionally, there isn't an integrated system to deal with or recuperate from such mistakes if an error in communication takes place, like a pipe breaking or an operation ending abruptly.

#### 2.2.1.5 Conclusion

The Named pipes (FIFOs) are a robust method for the inter-process communication allowing data to be passed between the processes using a named file. This mechanism is useful in the scenarios where processes need to exchange data without direct knowledge of the each other. The example demonstrates a simple producer-consumer model where one process writes data to a FIFO and another reads it showcasing the basic usage of the named pipes in C.

### System Calls and Functions for FIFO in Unix/Linux

#### 2.2.2.1 FIFO Creation and Permissions

The mkfifo() function creates a special file that behaves like a pipe. The file can be read from and written to by different processes.



* + - * pathname is the name of the FIFO file.
      * mode sets the permissions (e.g., 0666 for read and write access for everyone).

Return Value:

* 0: On success, indicating that the FIFO was created.
* -1: On failure, with errno set to an error code indicating the reason for failure, such as insufficient permissions or an already existing file at the specified path.

Explanation of mode Permissions:

* The mode argument specifies the file permissions, similar to the permissions used in other filesystem operations.
* 0666 is a common permission setting, which allows read and write access to all users (owner, group, and others).
* To limit access to only the creator, 0600 (owner read/write) may be used.

Key Points about mkfifo():

* Persistent in Filesystem: Unlike anonymous pipes (created by pipe()), named pipes persist in the filesystem. They appear as special files that remain until they are manually deleted, even if no process is actively using them.
* Multiple Processes: Named pipes allow unrelated processes to communicate since they don’t rely on a parent-child relationship. Any process with appropriate permissions and access to the path can open, read, or write to the FIFO.
* Sequential Communication: Data written to a FIFO is read in the same order, preserving a first-in, first-out (FIFO) structure suitable for sequential data flow.
* Blocking Behavior: By default, read() on an empty FIFO and write() on a full FIFO block until there is data to read or space to write, allowing for simple synchronization between producer and consumer processes.

#### 2.2.2.2 Open Modes

Opening a FIFO for writing (O\_WRONLY): The producer process opens the FIFO for writing. This operation may block if there is no process reading from the FIFO.

Opening a FIFO for reading (O\_RDONLY): The consumer process opens the FIFO for reading. This operation blocks until there is data available in the FIFO.



* pathname is the name of the FIFO.
* flags specify the mode (read or write).

Note: The argument flags must include one of the following access modes: O\_RDONLY, O\_WRONLY, or O\_RDWR. These request opening the file read-only, write-only, or read/write, respectively.

Opening a FIFO for Reading (O\_RDONLY):

When a process opens a FIFO with the O\_RDONLY flag, it’s considered a consumer, ready to receive data from the FIFO:

* Blocking Behavior: By default, open() with O\_RDONLY blocks if there is no producer (no process has opened the FIFO for writing). This ensures that the consumer waits until there is data or a connected producer. This blocking behavior is essential in ensuring that the consumer does not read an empty stream.
* Non-blocking Mode: If a consumer wants to avoid blocking on an empty FIFO, it can use O\_RDONLY | O\_NONBLOCK. In this mode:
  + open() will return immediately, even if no writer is connected.
  + However, if no data is available when read() is called, read() will return -1, and errno will be set to EAGAIN (indicating that the FIFO is empty).
* Use Case: This blocking behavior is typically helpful in producer-consumer scenarios, where the consumer must wait for the producer to send data, ensuring data synchronization between processes. For example, a logging service (consumer) may wait to receive log data from an application (producer).

Opening a FIFO for Writing (O\_WRONLY):

When a process opens a FIFO with the O\_WRONLY flag, it’s considered a producer, responsible for sending data into the FIFO:

* Blocking Behavior: open() with O\_WRONLY will block if there is no consumer (no process has opened the FIFO for reading). This means the producer will wait until there is a reader to receive the data, ensuring that any data written has a destination.
* Non-blocking Mode: If a producer wants to avoid blocking when no reader is present, it can use O\_WRONLY | O\_NONBLOCK. In this mode:
  + open() will return immediately, even if no consumer is connected.
  + However, attempts to write() to the FIFO will fail if no reader is available, returning -1 and setting errno to EPIPE. This error signifies a broken pipe, alerting the producer that there is no active consumer for its data.
* Use Case: This behavior helps prevent wasted writes in scenarios where data might otherwise go unconsumed. For instance, in a messaging system, the producer might wait for a consumer before sending any data to ensure the message has a recipient.

Advantages of Using open() with FIFOs

* Synchronization: The blocking nature of open() for FIFOs allows for natural synchronization between producers and consumers.
* Resource Efficiency: By blocking until a consumer is present, producers avoid wasting resources by writing data that would otherwise go unread.
* Error Handling: The non-blocking option with O\_NONBLOCK provides a way to handle cases where consumers or producers are not ready, allowing for more flexible and responsive error handling.

#### 2.2.2.3 Writing Data

The write() system call writes data from a process to a file or other output location specified by a file descriptor, such as a FIFO (First-In, First-Out) special file. Here’s a breakdown of how write() works and its parameters:



* fd: The file descriptor of the file or FIFO to write to, which is obtained from a successful call to open().
* buf: A pointer to the data buffer containing the bytes to be written.
* count: The number of bytes to write from the buffer.

Return Value:

* + - * On success, write() returns the number of bytes actually written, which may be less than count if an error occurs or if the file is full.
      * On failure, it returns -1 and sets errno to indicate the error.

How write() Works:

* Writing to a Regular File: The function writes count bytes from buf to the file described by fd.
* Writing to a FIFO: When writing to a FIFO, write() will block (wait) if no process is reading from it. If a reading process exists, data from the buffer is written into the FIFO.
* Handling Partial Writes: If the system can only write part of the requested data (due to disk space, memory limits, or network issues), write() may return a smaller number than count, indicating only partial data was written. You may need to handle this case by calling write() again to complete the remaining data.

#### 2.2.2.4 Reading Data

The read() system call allows a consumer process to read data from the FIFO into a buffer. The FIFO mechanism ensures that data is read in the same order it was written by the producer, allowing for synchronized communication between processes.



Parameters:

* fd: The file descriptor for the FIFO, obtained by opening the FIFO for reading.
* buf: A pointer to a buffer where the read data will be stored.
* count: The maximum number of bytes to read, representing the size of the buffer.

Return Value:

* On success, read() returns the number of bytes actually read, which may be less than count if fewer bytes are available.
* If read() reaches the end of the file (EOF), it returns 0.
* On failure, it returns -1, and errno is set to indicate the error.

How read() Works with FIFOs:

* Sequential Reading: FIFOs are designed for sequential reading, so data is read in the exact order it was written by the producer. This preserves the "first-in, first-out" nature, making it ideal for producer-consumer applications.
* Blocking Behavior: When a consumer calls read() on an empty FIFO, the read() call blocks (waits) until a producer writes data into it. This ensures that the consumer only reads meaningful data.
* End of Data: If the producer closes its file descriptor, and all data has been read, read() will return 0, signaling the end of data.

**Important Notes:**

* Blocking Behavior: By default, read() on an empty FIFO will block until data is available. To avoid blocking, you can open the FIFO in non-blocking mode using O\_NONBLOCK.
* Atomicity: For smaller reads (typically up to PIPE\_BUF, often 4096 bytes), reads are atomic, meaning the data won't be interleaved with other writes if multiple producers are writing to the FIFO.
* Error Handling: Always check the return value of read(); it can return -1 in cases of errors such as invalid file descriptors or if the FIFO is disconnected.

#### 2.2.2.5 Closing the file descriptor

The close() system call is used to close a file descriptor, releasing the resources associated with it and making it available for reuse by the process. After calling close() on a file descriptor, any further attempts to use that file descriptor will result in an error.



Parameters:

* fd: The file descriptor to be closed. This is the integer returned by open(), pipe(), socket(), or other system calls that create file descriptors.

Return Value:

* 0: Returned on success, indicating that the file descriptor has been closed successfully.
* -1: Returned on failure, with errno set to indicate the specific error. Common reasons for failure include invalid file descriptors (e.g., if fd is already closed or was never valid).

How close() Works:

* Releases Resources: When close() is called, any system resources associated with the file descriptor are released. For example, if the file descriptor references an open file, the operating system can free up memory or file locks associated with it.
* Flushes Pending Writes: If there’s buffered data that hasn’t been written to disk (e.g., in buffered output), close() will attempt to flush this data to the file. If the flush fails, close() will return an error, though this typically happens for networked or special files rather than regular files.
* Decrements Open File Description Reference Count: In Unix-like operating systems, each file descriptor is an entry in the process's file descriptor table, pointing to a system-wide open file description. close() decrements the reference count of this open file description. When the reference count reaches zero (i.e., no other file descriptors are pointing to it), the system fully closes the file.

Why close() is Important:

* Resource Management: File descriptors are a limited resource, and each process has a limited number available. Failing to close file descriptors can lead to “file descriptor leaks,” eventually causing the process to run out of descriptors, leading to errors on further open() or write() calls.
* Concurrency and Consistency: In network programming or when working with IPC (inter-process communication) like pipes and FIFOs, closing a file descriptor signals the end of data transmission. For example, when a producer process closes its FIFO write descriptor, the reader can detect that there’s no more data.

#### 2.2.2.6 Removing the FIFO

The unlink() system call is used to delete a file or symbolic link from the filesystem. In the context of a FIFO (First-In, First-Out special file), calling unlink() removes the FIFO file from the filesystem after it is no longer needed, ensuring that it does not persist as an unnecessary resource.



Parameters:

* pathname: The path to the file or FIFO to be deleted. This is the same path used to create the FIFO with mkfifo().

Return Value:

* 0: Returned on success, indicating that the file has been successfully removed from the filesystem.
* -1: Returned on failure, with errno set to indicate the specific error. Common errors include insufficient permissions or an invalid path.

How unlink() Works with FIFOs:

* Removing the FIFO File: When unlink() is called on a FIFO, it removes the file entry from the directory structure, effectively deleting it from the filesystem.
* Delayed Deletion: If there are processes still using the FIFO (i.e., processes with an open file descriptor for the FIFO), unlink() does not immediately remove it from memory. Instead, the filesystem keeps the FIFO until all file descriptors referencing it are closed, allowing current reads and writes to complete.
* Reclaiming Resources: Once all processes have closed their file descriptors for the FIFO, the operating system frees the resources associated with it, fully removing the FIFO.

Why unlink() is Used with FIFOs:

* Temporary Communication Channel: FIFOs are often used as temporary communication channels for inter-process communication (IPC). After the producer and consumer processes finish using the FIFO, unlink() cleans up the resource, avoiding clutter in the filesystem.
* Preventing Accidental Reuse: By deleting the FIFO when it’s no longer needed, you prevent other processes from accidentally accessing or writing to an old FIFO that’s no longer in use.

# CHAPTER 3: DEMO FOR FIFO IN IPC

## 3.1 Demo calculate the total amount of random generated

## 3.2 Demo input a string uses FIFO