Unit 4

Inter Process Communication:

Pipe
Pipes come in two flavors, named and unnamed, and can be used either interactively from the command line or within programs.

In Linux, the pipe command lets you sends the output of one command to another. Piping, as the term suggests, can redirect the standard output, input, or error of one process to another for further processing.

It can also be visualized as a temporary connection between two or more commands/ programs/ processes. The command line programs that do the further processing are referred to as filters.

This direct connection between commands/ programs/ processes allows them to operate simultaneously and permits data to be transferred between them continuously rather than having to pass it through temporary text files or through the display screen
The syntax for the pipe or unnamed pipe command is the | character between any two commands:

```
Command-1 | Command-2 | …| Command-N
```

Here, the pipe cannot be accessed via another session; it is created temporarily to accommodate the execution of Command-1 and redirect the standard output. It is deleted after successful execution.

For Example:

```
ls | grep file.txt
```

In this first we are using `ls` to list all file and directories in the current directory, then passing its output to `grep` command and searching for file name `file.txt`. The output of the ls command is sent to the input of the grep command, and the result is a list of files that match the search term.

```
sleep 5 | echo "Hello, world!"
```

The *sleep* and *echo* utilities execute as separate processes, and the unnamed pipe allows them to communicate. However, the example is contrived in that no communication occurs. The greeting *Hello, world!* appears on the screen; then, after about five seconds, the command line prompt returns, indicating that both the *sleep* and *echo* processes have exited. What's going on?

In the vertical-bar syntax from the command line, the process to the left (*sleep*) is the writer, and the process to the right (*echo*) is the reader. By default, the reader blocks until there are bytes to read from the channel, and the writer—after writing its bytes—finishes up by sending an end-of-stream marker.

Named Pipe
Linux named pipes, also called FIFO (First In First Out) are special kind of files which are **used to establish two-way communications between two unrelated programs**. A named pipe is called so because it is a file in the filesystem, which can be listed with ls unlike the normal pipes in Linux exist only inside the kernel.

An unnamed pipe has no backing file: the system maintains an in-memory buffer to transfer bytes from the writer to the reader. Once the writer and reader terminate, the buffer is reclaimed, so the unnamed pipe goes away. By contrast, a named pipe has a backing file and a distinct API.

Creating a FIFO file: In order to create a FIFO file, a function calls i.e. mkfifo is used.

As the utility's name *mkfifo* implies, a named pipe also is called a FIFO because the first byte in is the first byte out, and so on. There is a library function named **mkfifo** that creates a named pipe in programs and is used in the next example, which consists of two processes: one writes to the named pipe and the other reads from this pipe.

Let's look at another command line example to get the gist of named pipes. Here are the steps:
Open two terminals. The working directory should be the same for both.

In one of the terminals, enter these two commands

```
$ mkfifo tester  ## creates a backing file named tester
$ cat tester     ## type the pipe's contents to stdout
```

At the beginning, nothing should appear in the terminal because nothing has been written yet to the named pipe.
In the second terminal, enter the command:

```
$ cat > tester  ## redirect keyboard input to the pipe
hello, world!   ## then hit Return key
bye, bye        ## ditto
<Control-C>     ## terminate session with a Control-C
```

Whatever is typed into this terminal is echoed in the other. Once **Ctrl+C** is entered, the regular command line prompt returns in both terminals: the pipe has been closed.

Clean up by removing the file that implements the named pipe:

```
$ unlink tester
```

- It is an extension to the traditional pipe concept on Unix. A traditional pipe is “unnamed” and lasts only as long as the process.
- A named pipe, however, can last as long as the system is up, beyond the life of the process. It can be deleted if no longer used.
- Usually a named pipe appears as a file and generally processes attach to it for inter-process communication. A FIFO file is a special kind of file on the local storage which allows two or more processes to communicate with each other by reading/writing to/from this file.
- A FIFO special file is entered into the filesystem by calling *mkfifo()* in C. Once we have created a FIFO special file in this way, any process can open it for reading or writing, in the same way as an ordinary file. However, it has to be open at both ends simultaneously before you can proceed to do any input or output operations on it.

When to Use Named Pipes and Anonymous Pipes

The decision to use named and anonymous pipes depends entirely on the requirements of the current situation. The situations can have characteristics like two-way communication, creating a filter etc.

For example, if we want to filter multiple commands together in succession, anonymous or normal pipes are the best and easiest way to do this.

But, say we encounter a client-server-like situation where one user needs to create data and the other needs to process it, then Linux named pipes are the perfect solution for it.

Semaphore

Computational and memory resources are limited in a system, and multiple processes need to share these resources to ensure the performant working of a system. As multiple processes compete with each other for these limited resources, we need a method to control the sharing of these resources between multiple processes efficiently. Semaphores are one such technique.

Semaphore in Linux is a technique used for coordinating and synchronizing the activities of multiple processing competing for the same resources.

What are Semaphores?

A Semaphore in Linux is essentially an integer variable that is used to control access to a shared resource in Linux by multiple processes. Semaphores are simple control mechanisms that facilitate inter-process communications (IPC) and allow processes to coordinate their access to shared resources to avoid race conditions and excessive resource utilization.

A semaphore is usually implemented as an integer with two operations:

1 Increment (post or signal): The increment operation is used to increase the value of a semaphore.
2 Decrement (wait or acquire): The decrement operation is used to decrement or reduce the value of a semaphore.

A resource is indicated as 'available' when the value of a semaphore in Linux is positive. On the other hand, a resource is indicated as 'busy' or 'blocked' when the value of a semaphore is negative.

Semaphores are of two types: binary semaphores and counting semaphores.

As the name suggests, binary semaphores are semaphores that can only have two values: 0 and 1. These are used to synchronize access to a single shared resource. A resource is termed as 'available' when the value of its binary semaphore is one and is termed as 'unavailable' when the value of its binary semaphore is 0.

A counting semaphore can have a value greater than 1. These are used to synchronize access to multiple shared resources. The integer value of these semaphores is used to denote the number of available resources.

Implementing Semaphore in Linux

There are two ways to implement semaphores in Linux: using System V IPC capabilities and the modern POSIX-compliant semaphores.

The System V-based semaphores are implemented using the semget() function. We can use **semget()** to either create or access an existing semaphore in Linux.

Apart from this, we also use other functions like **semop()** to perform operations on the semaphore. Another function used to interact with the semaphore is the **semctl()** function which is used to control the semaphore.

Message Queues

A **message queue** is an inter-process communication (IPC) mechanism that allows processes to exchange data in the form of messages between two processes. It allows processes to communicate asynchronously by sending messages to each other where the messages are stored in a queue, waiting to be processed, and are deleted after being processed.

Message queues provide a buffer where processes can asynchronously send and receive messages in inter-process communication. By decoupling the sender and receiver, message queues enable seamless communication and empower developers to design robust and interconnected systems.

A message queue is a linked list of messages stored within the kernel and identified by a message queue identifier. A new queue is created or an existing queue opened by **msgget()**.

New messages are added to the end of a queue by **msgsnd()**. Every message has a positive long integer type field, a non-negative length, and the actual data bytes (corresponding to the length), all of which are specified to msgsnd() when the message is added to a queue. Messages are fetched from a queue by **msgrcv()**. We don't have to fetch the messages in a first-in, first-out order. Instead, we can fetch messages based on their type field.

All processes can exchange information through access to a common system message queue. The sending process places a message (via some (OS) message-passing module) 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.

System calls used for message queues:

- **ftok()**: is use to generate a unique key.
- **msgget()**: either returns the message queue identifier for a newly created message queue or returns the identifiers for a queue which exists with the same key value.
- **msgsnd()**: Data is placed on to a message queue by calling msgsnd().
- **msgrcv()**: messages are retrieved from a queue.
- **msgctl()**: It performs various operations on a queue. Generally it is use to destroy message queue.

Functions of Message Queue

There are four important functions that we will use in the programs to achieve IPC using message queues.

1. int msgget (key_t key, int msgflg);

We use the msgget function to create and access a message queue. It takes two parameters.

- The first parameter is a key that names a message queue in the system.
- The second parameter is used to assign permission to the message queue and is ORed with IPC_CREAT to create the queue if it doesn't already exist. If the queue already exists, then IPC_CREAT is ignored. On success, the msgget function returns a positive number which is the queue identifier, while on failure, it returns -1.

2. int msgsnd (int msqid, const void *msg_ptr, size_t msg_sz, int msgflg);

This function allows us to add a message to the message queue.

- The first parameter (msgid) is the message queue identifier returned by the msgget function.
- The second parameter is the pointer to the message to be sent, which must start with a long int type.
- The third parameter is the size of the message. It must not include the long int message type.
- The fourth and final parameter controls what happens if the message queue is full or the system limit on queued messages is reached. The function on success returns 0 and place the copy of message data on the message queue. On failure, it returns -1.

There are two constraints related to the structure of the message. First, it must be smaller than the system limit and, second, it must start with a long int. This long int is used as a message type in the receive function.

3. int msgrcv (int msqid, void *msg_ptr, size_t msg_sz, long int msgtype, int msgflg);

This function retrieves messages from a message queue.

- The first parameter (msgid) is the message queue identifier returned by the msgget function.
- As explained above, the second parameter is the pointer to the message to be received, which must start with a long int type.
- The third parameter is the size of the message.
- The fourth parameter allows implementing priority. If the value is 0, the first available message in the queue is retrieved. But if the value is greater than 0, then the first message with the same message type is retrieved. If the value is less than 0, then the first message having the type value same as the absolute value of msgtype is retrieved. In simple words, 0 value means to receive the messages in the order in which they were sent, and non zero means receive the message with a specific message type.
- The final parameter controls what happens if the message queue is full or the system limit on queued messages is reached. The function on success returns 0 and place the copy of message data on the message queue. On failure, it returns -1.

4. int msgctl (int msqid, int command, struct msqid_ds *buf);

The final function is msgctl, which is the control function.

- The first parameter is the identifier returned by the msgget function.
- The second parameter can have one out of the below three values.

Shared Memory

Inter Process Communication through shared memory is a concept where two or more process can access the common memory and communication is done via this shared memory where changes made by one process can be viewed by another process.
The problem with pipes, fifo and message queue – is that for two process to exchange information. The information has to go through the kernel.

- Server reads from the input file.
- The server writes this data in a message using either a pipe, fifo or message queue.
- The client reads the data from the IPC channel, again requiring the data to be copied from kernel's IPC buffer to the client's buffer.
- Finally, the data is copied from the client's buffer.

A total of four copies of data are required (2 read and 2 write). So, shared memory provides a way by letting two or more processes share a memory segment. With Shared Memory the data is only copied twice – from input file into shared memory and from shared memory to the output file.

Shared memory is a memory shared between two or more processes. Each process has its own address space; if any process wants to communicate with some information from its own address space to other processes, then it is only possible with IPC (inter-process communication) techniques.

Shared memory is the fastest inter-process communication mechanism. The operating system maps a memory segment in the address space of several processes to read and write in that memory segment without calling operating system functions.

Used System Calls
The system calls that are used in the program are:

Function	Signature	Description
ftok()	key_t ftok()	It is used to generate a unique key.
shmget()	int shmget(key_t key,size_t size, int shmflg);	Upon successful completion, shmget() returns an identifier for the shared memory segment.
shmat()	void *shmat(int shmid ,void *shmaddr ,int shmflg);	Before you can use a shared memory segment, you have to attach yourself to it using shmat(). Here, shmid is a shared memory ID and shmaddr specifies the specific address to use but we should set it to zero and OS will automatically choose the address.
shmdt()	int shmdt(void *shmaddr);	When you're done with the shared memory segment, your program should detach itself from it using shmdt().
shmctl()	shmctl(int shmid,IPC_RMID,NULL);	When you detach from shared memory, it is not destroyed. So, to destroy shmctl() is used.

1. shmget() Function - is used to create the shared memory segment,

int shmget (key_t key, size_t size, **int** shmflg);

The first parameter specifies the unique number (called key) identifying the shared segment. The second parameter is the size of the shared segment, e.g., 1024 bytes or 2048 bytes. The third parameter specifies the permissions on the shared segment. On success, the shmget() function returns a valid identifier, while on failure, it returns -1.

2. shmat() Function is used to attach the shared segment with the process's address space.
void *shmat(**int** shmid, **const void** *shmaddr, **int** shmflg);

shmat() function is used to attach the created shared memory segment associated with the shared memory identifier specified by ***shmid*** to the calling process's address space. The first parameter here is the identifier which the shmget() function returns on success. The second parameter is the address where to attach it to the calling process. A NULL value of the second parameter means that the system will automatically choose a suitable address. The third parameter is '0' if the second parameter is NULL. Otherwise, the value is specified by SHM_RND.