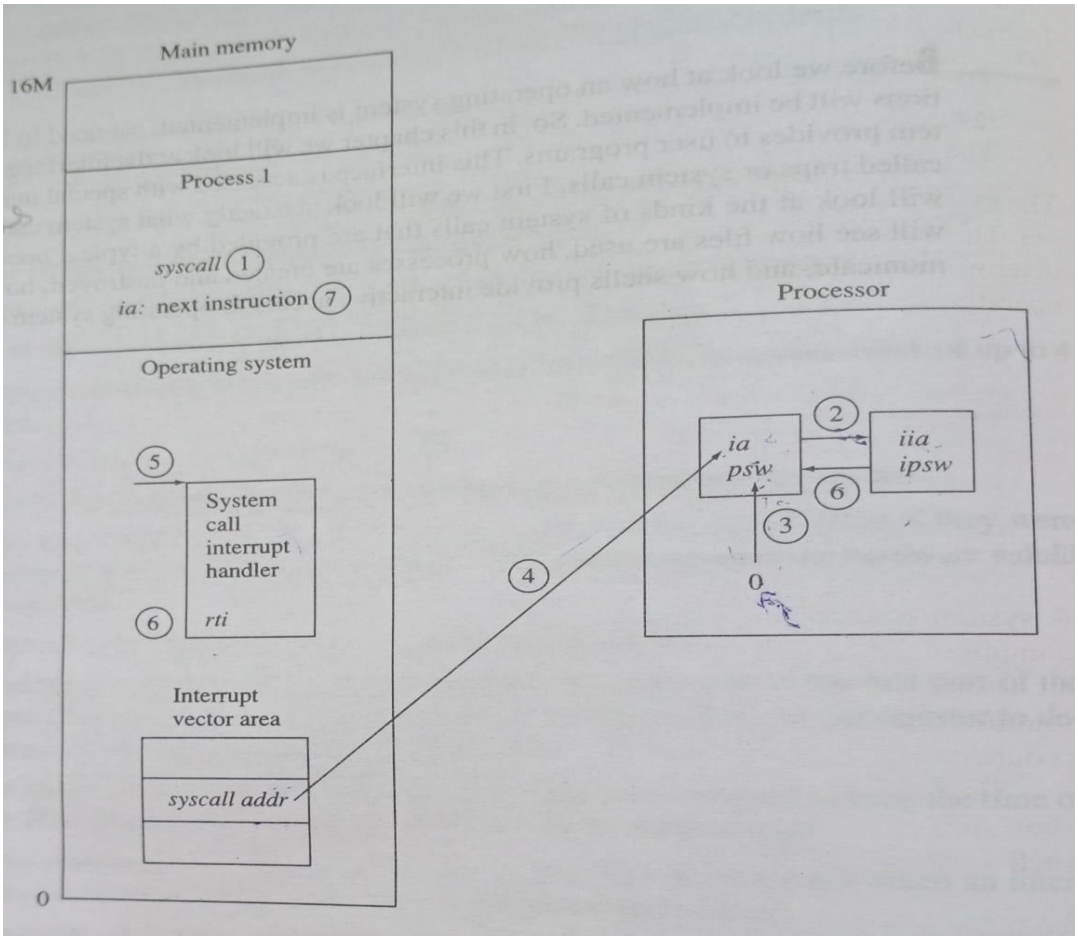
UNIT -2: CHAPTER 1: OPERATING SYSTEMS

Topic 1: The Operating System Interface

1.1 System Call:

A System call instruction is an instruction that does not execute a specific function in hardware but instead generates an interrupt that causes the operating system to gain control of the processor. The operating system then determines what kind of system call it is and performs the appropriate service for the system caller.

The below illustrates the flow of control during a system call. A system call goes through the following steps:



1. The program executes the system call instruction.
2. The hardware saves the current ia and psw registers in the iia and ipsw registers.

3. The hardware loads a value of 0 into the psw register, which puts the machine in system mode with interrupts disabled.
4. The hardware loads the ia register from the system call interrupt vector location. This completes the execution of the system call instruction by the hardware.
5. Instruction execution continues at the beginning of the system call interrupt handler.
6. The system call handler completes and executes a return from interrupt instruction. This restores the ia and psw from the iia and ipsw.
7. The process that executed the system call instruction continues at the instruction after the system call.

1.2 Making A System Call:

A System call is made using the system call machine language instruction. Few compilers will allow you to make system calls directly, so they must be made from an assembly language program**.**

1.3 System Call Interface:

The Operating system provides a set of operations which are called system calls. These extend the native hardware instructions. All services that the operating system provides can be requested through system calls. From the point of view of a user, the entire functionality of the operating system is defined by the system calls. A system call interface is the description of the set of system calls implemented by the operating system.

Topic 2: AN EXAMPLE SYSTEM CALL INTERFACE

2.1 UNIX SCI: This interface is quite similar to a subset of the UNIX system call interface. Many of these calls will run unchanged on a UNIX system. We will generally not mention instances where the UNIX interface provides additional functions beyond what we define in this interface.

File and I/O System Calls:

- ✓ **Open**-get ready to read or write a file.
- ✓ **Create**-create a new file and open it.
- ✓ **Read**-read bytes from an open file.
- ✓ **Write**-write bytes to an open file.
- ✓ **Lseek**-change the location in the file of the next read or write.
- ✓ **Close**-indicate that you are done reading or writing a file.
- ✓ **Unlink**-remove a file name from a directory.
- ✓ **Stat**-get information about a file.

Process Management System Calls:

- ✓ CreateProcess-create a new process.
- ✓ Exit-terminate the process to Exit.
- ✓ Wait-wait for another process to Exit
- ✓ Fork-create a duplicate of the process making the system call.
- ✓ Execv-run a new program in the process making the system call.

Interprocess Communication System Calls:

- ✓ CreateMessageQueue-create a queue to hold messages.
- ✓ SendMessage-send a message to a message queue.
- ✓ ReceiveMessage-receive a message from a message queue.
- ✓ DestroyMessageQueue-destroy a message queue.

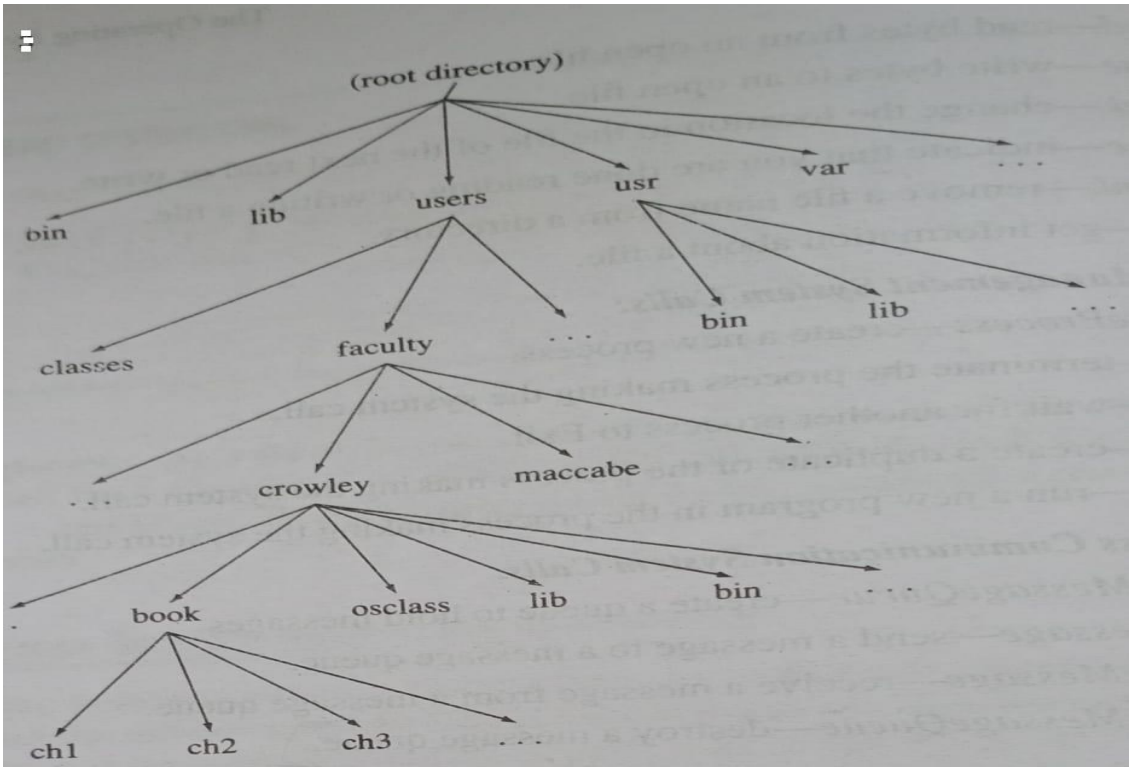
2.2 HIERARCHICAL FILE NAMING SYSTEMS:

A basic concept in a hierarchical file naming system is a directory. A directory is a collection of names, each referring to a file or another directory. If a name leads to another directory, then we can use that directory as a collection of names, each referring to a file or another directory. This creates a tree structure

where leaf nodes are files and interior nodes are directories. There is a special directory that is the root of the tree and it is called the root directory.

Each directory is a collection of names, and all the names in a directory are unique. A directory forms a name space, and a directory also includes the mapping from each name to the file or directory it refers to. So, given a name and a directory, you can find the file or directory the name refers to. This is called a name lookup.

Files in a hierarchical file system are named with structured, multipart file names. A file name represents the path from the root directory to the file being named. A special character is used to spate the names in the path. The below figure illustrates the naming conventions used in UNIX



2.3 FILE AND I/O SYSTEM CALLS

The following table summarizes the system calls for file operations. We assume a hierarchical file system like the one used in UNIX and most other operating systems.

Call	Parameters	Returns	Notes
open	name,flags	fid	Create open file connected to a file
creat	name,mode	fid	Create file and connect to open file
read	fid,buffer,count	count	Read bytes from open file
write	fid,buffer,count	count	Write bytes to open file
lseek	fid,offset,mode	offset	Move position of next read or write
close	fid	code	Disconnect open file from file
unlink	name	code	Delete the named file

2.4 Complete specification of the file-related system calls:

- **int open(char*pathName, int openFlags):-**The named file is opened and an integer, the open file identifier, is returned. The file location is initialized to 0. The flags argument can be one of:

  0-open for reading.
  1-open for writing.
  2-open for reading and writing.

***int creat(char*pathName, int fileMode):-** The named file is created as an empty file and opened for writing, and an integer, the open file identifier, is returned. The possible error codes are:

-1-the file path name was invalid.
-2-the fileMode argument is invalid.

Topic 3: INFORMATION AND META-INFORMATION

Information is generally not a pure, complete, self-describing entity. Usually, we want to record information about the information, that is, meta-information. A good example is a file. The file contains information itself, but the operating system also keeps meta-information about the file:

*Who owns the file.
*Who can use the file and how
*When the file was created.
*When the file was last modified.
*When the file was last read.
*How long the file is.

Topic 4: NAMING OPERATING SYSTEM OBJECTS

File system uses hierarchical names for naming files and directories. In a file system, the names map into either or directories. A file is an object that you then do some operation on while a directory is another name map that you use for the next step of the file name look up.

Names in a file system really map into file object descriptors on disk and this file object descriptors point to the file objects themselves we can easily extended this file naming system to name other kinds of operating system objects as well. When a user or program needs to refer to any object it needs a name for that object. So, it is convenient to use the file naming system to name all the objects that need names.

Topic 5: DEVICES AS FILES

A File is a collection of information on disk and an open file is an object that you can read data from or write data to so we can think of an open file as a source of bytes or a sink for bytes

But objects other than files can also provide data and accept data. For example, the keyboard on a terminal or work station is a source of bytes, and a terminal screen or a window on a work station can accept bytes and display them on the screen. In fact, all i/o devices either receive data or send data or both. Give

this similarity b/w devices and open files, it is useful to try to unify the concepts of file and device in an operating system.

Every file has a name in the hierarchical file naming system. Since devices need names also, we give them names in the file name hierarchy too. We will add special entries in the file systems for the devices. In UNIX, there is a specially named directory, /dev, where most devices names are kept.

5.1 UNIFICATION OF THE FILE AND DEVICE CONCEPTS:

The file system has two distinct parts to its services. The file part is the file naming system, which maps hierarchical names into objects. There is no particular reason that a name has to refer to a file since it just maps to some objects. Most operating system take advantage of that fact and use the file naming system to provide names that user can use to refer to a number of different objects. In particular, you can use the file naming system to give names to i/o devices.

Topic 6: THE PROCESS CONCEPT

When we execute a program on one of the virtual computers, we call the executing program a process. The principal goal of an operating system is to implement virtual computers and the processes that run on them the virtual computer is created for each process, thereby giving the process the illusion of running on its own physical computer. The virtual computer extends the hardware instructions set with a set of system calls that request operating system services. The virtual computer is the execution environment created by the operating system, and the process is the program execution that occurs in that environment.

Many users are not aware of the process concept because they just run a program in a single process on the computer. Normally other programs create processes for you. For example, a shell creates your first process “for free”, that is, it is automatically created for you and starts up running the program you requested. Many users use multiple processes in their work, even though they may not realize it. A shell pipe line is implemented by creating a process for each program in the pipe line. If we compile a program in the background and edit in the foreground, you are using multiple processes.

6.1 PROCESSES AND PROGRAMS:

It is important to understand to understand the distinction between a program and a process. A program is a static object that can exist in a file. It contains the instructions. A process is a dynamic object that is a program in execution. A program is a sequence of instructions. A process is a sequence of instruction execution

6.2 PROCESS MANAGEMENT SYSTEM CALLS:

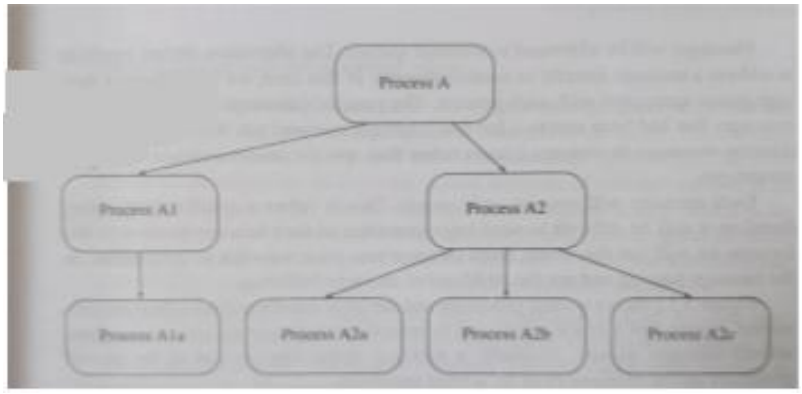
The operating system will allow a process to create new processes, to terminate itself, and to wait for a process it created to complete. There are just 3 system calls dealing with processes:

- ✓ **int SimpleCreateProcess: -** A new process is created and the program Program Name is run in that process. The return value is the process identifier of the new process.
- ✓ **void SimpleExit: -** The process making the system call is terminated.
- ✓ **void SimpleWait: -** The calling process will until the process specified by the process identifier Pid terminates

6.3 PROCESS HIERARCHY:

When one process creates another process, we say that creator is the parent process and the created process in the child process. The parent or child relationship can be shown as a tree structure of the processes in the system, and this tree structure is called the process hierarchy.

An example of process hierarchy:



Topic7: COMMUNICATION BETWEEN PROCESSES

Using the process system calls, we can send arguments to a new process and get a return code when the process completes. Sometimes, we will also want process to communicate with each other will they are running. We call this interprocess communication or IPC.

We will allow a process to send a message even go the receiver is not yet ready to receive it. We will do this by buffering the messages, that is, we will save the message the until the receiver ask for it. We buffer the message in a message queue, which holds messages that have been sent but not yet received. Message queues will be a type of objects supported by the operating system. The process can create a message queue, and then any processes can send message to it or receive messages from it.

Topic 8: UNIX-STYLE PROCESS CREATION

UNIX uses a very different method of process creation. In UNIX, two system calls are necessary to create a new process running a different program, one to do the actual creation of a new process and the other to run a specific program in the new process. We will present the UNIX process system calls and show how they are used.

The following table summarizes the UNIX process-related system calls:

Call	Parameters	Returns	Notes
fork	None	pid or 0	Duplicate process
execv	programFile, args	None	Execute program in process
exit	return code	Does not return	Destroy process
wait	return code address	pid	Wait for process to exit

***int fork: -** A new process is created that is an exact copy of the process making the system call. The return value to the parent is the process identifier of the new process. The return value to the child is 0.

*** int execv: -** The program Program Name is run in the calling process. The program in the calling process is overwritten and will no longer exist. The arguments are in the argument vector argv, which is an array of strings, that is, an array of pointers to characters.

***void exit:-**This system call will cause the calling process to wait until any process created by the calling process exits. The return value is the process identifier of the process that exited. The return code of that process is stored in returnCode.

The fork system call is somewhat unusual. A copy of the calling process is made and runs as a new process. The copy is of the memory of the calling process at the time of the fork system call, not the program the calling process was started from. The new process does not start at the beginning but at the exact point of the fork system call.

STANDARD INPUT AND STANDARD OUTPUT: -

Many programs read data, transform it, and write it out again. By convention, in UNIX, these programs read their input from standard input and write their output to standard output. When the shell starts a program, standard input and standard output are normally the user's terminal, but they can be changed by a method called redirection.

The program will run with standard input reading from the open file in file and standard output writing to the device named/dev/tty23. A program whose main function is to read a sequence of bytes from its standard input, transform the stream in some way, and then write it to standard output is called filter.

COMMUNICATING WITH PIPES: -

The idea to allow interprocess communication through a technique called named pipes. A pipe is an object that can act like a file, that is, it can be written to and read from. We can write bytes to the pipe. When we read from a pipe, we get the same bytes that were written and in the same order as they were written. A pipe is useful when we want to write from one process and read from another. This allows us to use pipes as a method of communication between processes, and alternative to messages.

With messages, there is a fixed message size which both sender and receiver must adhere to. With a pipe, a process writes to the other process as if it were writing to a file. There are no fixed-size blocks.

NAMING OF PIPES AND MESSAGE QUEUES: -

One difference between named pipes and message queues is how they are named. Named pipes are named by the file system naming system and they are implemented by an entry in the file system indicating that they are a named some kind of existence even when no process is using them. Any two processes can communicate if they both know the name of the pipe. They just both open the pipe and use it as an open file.

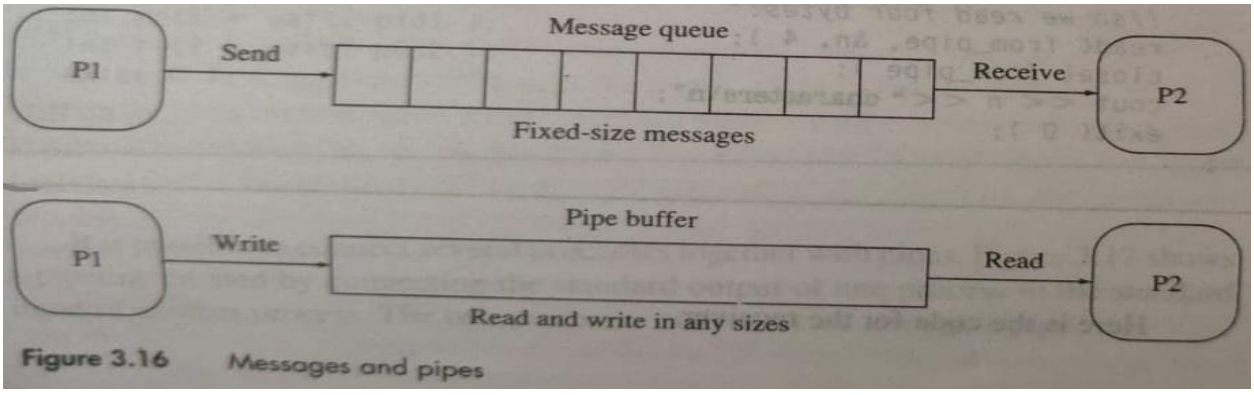


Figure 3.16 Messages and pipes

Message queues have no name and no existence before they are created by a process. When they are created, they are given an internal name for use by processes wanting to pass messages. This name can be passed to other processes, but passing the name requires some form of communication. That limits the processes that can receive the name to descendants of the process that created the message queue. So, two processes need a common ancestor in order to communicate. This is much less convenient than the named pipe approach.

SUMMARY OF SYSTEM CALL INTERFACES:

- ✓ File and I/O – open, create, read, write, lseek, close, unlink, stat. These gave us the ability to create and destroy files and to read and write them.
- ✓ Processes-CreateProcess, Wait, Exit. These gave us the ability to create and terminate processes and to wait for a child process.
- ✓ Interprocess communication-CreateMessageQueue, sendMessage, Receive message queues and to send and receive messages using message queues.