UNIX PROGRAMMING (Effective from the academic year 2018 -2019) SEMESTER – V			
Course Code	**18CS56**	**CIE Marks**	40
Number of Contact Hours/Week	3:0:0	**SEE Marks**	60
Total Number of Contact Hours	40	**Exam Hours**	03
CREDITS – 3			

Course Learning Objectives: This course (18CS56) will enable students to

- Interpret the features of UNIX and basic commands.
- Demonstrate different UNIX files and permissions
- Implement shell programs.
- Explain UNIX process, IPC and signals.

Module 1	**Contact Hours**
Introduction: Unix Components/Architecture. Features of Unix. The UNIX Environment and UNIX Structure, Posix and Single Unix specification. General features of Unix commands/ command structure. Command arguments and options. Basic Unix commands such as echo, printf, ls, who, date,passwd, cal, Combining commands. Meaning of Internal and external commands. The type command: knowing the type of a command and locating it. The root login. Becoming the super user: su command. **Unix files:** Naming files. Basic file types/categories. Organization of files. Hidden files. Standard directories. Parent child relationship. The home directory and the HOME variable. Reaching required files- the PATH variable, manipulating the PATH, Relative and absolute pathnames. Directory commands – pwd, cd, mkdir, rmdir commands. The dot (.) and double dots (..) notations to represent present and parent directories and their usage in relative path names. File related commands – cat, mv, rm, cp, wc and od commands. **RBT: L1, L2**	08
Module 2	
File attributes and permissions: The ls command with options. Changing file permissions: the relative and absolute permissions changing methods. Recursively changing file permissions. Directory permissions. **The shells interpretive cycle:** Wild cards. Removing the special meanings of wild cards. Three standard files and redirection. **Connecting commands:** Pipe. Basic and Extended regular expressions. The grep, egrep. Typical examples involving different regular expressions. **Shell programming:** Ordinary and environment variables. The .profile. Read and readonly commands. Command line arguments. exit and exit status of a command. Logical operators for conditional execution. The test command and its shortcut. The if, while, for and case control statements. The set and shift commands and handling positional parameters. The here (<<) document and trap command. Simple shell program examples. **RBT: L1, L2**	08
Module 3	
UNIX File APIs: General File APIs, File and Record Locking, Directory File APIs, Device File APIs, FIFO File APIs, Symbolic Link File APIs. **UNIX Processes and Process Control:** **The Environment of a UNIX Process:** Introduction, main function, Process Termination, Command-Line Arguments, Environment List, Memory Layout of a C Program, Shared Libraries, Memory Allocation, Environment Variables, setjmp and longjmp Functions, getrlimit, setrlimit Functions, UNIX Kernel Support for Processes. **Process Control:** Introduction, Process Identifiers, fork, vfork, exit, wait, waitpid, wait3,	08

wait4 Functions, Race Conditions, exec Functions **RBT: L1, L2, L3**	
Module 4	
Changing User IDs and Group IDs, Interpreter Files, system Function, Process Accounting, User Identification, Process Times, I/O Redirection. **Overview of IPC Methods**, Pipes, popen, pclose Functions, Coprocesses, FIFOs, System V IPC, Message Queues, Semaphores. **Shared Memory**, Client-Server Properties, Stream Pipes, Passing File Descriptors, An Open Server-Version 1, Client-Server Connection Functions. **RBT: L1, L2, L3**	08
Module 5	
Signals and Daemon Processes: Signals: The UNIX Kernel Support for Signals, signal, Signal Mask, sigaction, The SIGCHLD Signal and the waitpid Function, The sigsetjmp and siglongjmp Functions, Kill, Alarm, Interval Timers, POSIX.1b Timers. Daemon Processes: Introduction, Daemon Characteristics, Coding Rules, Error Logging, Client-Server Model. **RBT: L1, L2, L3**	08

Course Outcomes: The student will be able to :

- Explain Unix Architecture, File system and use of Basic Commands
- Illustrate Shell Programming and to write Shell Scripts
- Categorize, compare and make use of Unix System Calls
- Build an application/service over a Unix system.

Question Paper Pattern:

- The question paper will have ten questions.
- Each full Question consisting of 20 marks
- There will be 2 full questions (with a maximum of four sub questions) from each module.
- Each full question will have sub questions covering all the topics under a module.
- The students will have to answer 5 full questions, selecting one full question from each module.

Textbooks:

1. Sumitabha Das., Unix Concepts and Applications., 4th Edition., Tata McGraw Hill (Chapter 1,2 ,3,4,5,6,8,13,14)
2. W. Richard Stevens: Advanced Programming in the UNIX Environment, 2nd Edition, Pearson Education, 2005 (Chapter 3,7,8,10,13,15)
3. Unix System Programming Using C++ - Terrence Chan, PHI, 1999. (Chapter 7,8,9,10)

Reference Books:

1. M.G. Venkatesh Murthy: UNIX & Shell Programming, Pearson Education.
2. Richard Blum , Christine Bresnahan : Linux Command Line and Shell Scripting Bible, 2ndEdition, Wiley,2014.

Faculty can utilize open source tools to make teaching and learning more interactive.

Chapter 1: Introduction

1. Unix Components/Architecture
2. Features of Unix
3. The UNIX Environment and UNIX Structure, Posix and Single Unix specification
4. General features of Unix commands/ command structure. Command arguments and options
5. Basic Unix commands such as echo, printf, ls, who, date,passwd, cal, Combining commands
6. Meaning of Internal and external commands
7. The type command: knowing the type of a command and locating it
8. The root login. Becoming the super user: su command

➢ Unix Components/Architecture

1.1 The Operating System

- An **operating system** is the software that manages the computer's hardware and provides a convenient and safe environment for running programs.
- It acts as an interface between user programs and the hardware resources that these programs access like – processor, memory, hard disk, printer & so on.
- It is loaded into memory when a computer is booted and remains active as long as the machine is up.

1.2 The UNIX Operating System

- Like DOS and Windows, there's another operating system called UNIX.
- UNIX is a giant operating system with sheer power.
- Developed by Ken Thompson and Dennis Ritchie.
- It runs practically on every Hardware and provided inspiration to the Open Source movement.
- You interact with a UNIX system through a **command interpreter** called the **shell.**

Unix Components/Architecture

- The UNIX architecture has three important agencies-
  - Division of labor: Kernel and shell
  - The file and process

- The system calls

Division of labor: Kernel and shell

The Kernel

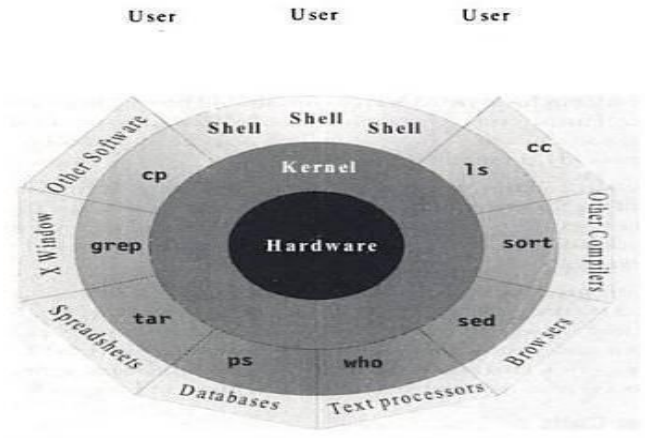
- ✓ The kernel interacts with the machine's hardware
- ✓ The core of the operating system - a collection of routines mostly written in C.
- ✓ It is loaded into memory when the system is booted and communicates directly with the hardware.
- ✓ User programs (the applications) that need to access the hardware use the services of the kernel, which performs the job on the user's behalf.
- ✓ These programs access the kernel through a set of functions called system calls.
- ✓ Apart from providing support to user's program, kernel also does important housekeeping.
- ✓ It manages the system's memory, schedules processes, and decides their priorities and so on.
- ✓ The kernel has to do a lot of this work even if no user program is running.
- ✓ The kernel is also called as the operating system - a programs gateway to the computer's resources.

The Shell

- ✓ Computers don't have any capability of translating commands into action.
- ✓ That requires a **command interpreter**, also called as the shell.
- ✓ Shell is acts interface between the user and the kernel.
- ✓ Most of the time, there's only one kernel running on the system, there could be several shells running – one for each user logged in.
- ✓ The shell accepts commands from user, if require rebuilds a user command, and finally communicates with the kernel to see that the command is executed.
- ✓ Example:

```
$echo VTU   Belagavi      #Shell rebuilds echo command by removing multiple spaces
VTU Belagavi
```

- The following figure shows the kernel-shell Relationship:



The file and process

- Two simple entities support the UNIX – the file and the process.
- "Files have places and processes have life"

The File

- A file is just an array of bytes and can contain virtually anything.
- A file forms a Hierarchical file system.
- Every file in UNIX is part of the one file structure provided by UNIX.
- UNIX considers directories and devices as members of the file system.

The Process

- The process is the name given to the file when it is executed as a program (Process is program under execution).
- We can say process is the "time image" of an executable file.
- UNIX provides tools to control processes move them between foreground and background and kill them.

The System Calls

- The UNIX system-comprising the kernel, shell and applications-is written in C.
- Though there are several commands that use functions called system calls to communicate with the kernel.
- All UNIX flavors have one thing in common – they use the same systemcalls. Eg: write, open

➢ Features of Unix

1. **A Multiuser System**
   - ➢ UNIX is a multiprogramming system, it permits multiple programs to run and compete for the attention of the CPU.
   - ➢ This can happen in two ways:
     - Multiple users can run separate jobs
     - A single user can also run multiple jobs
   - ➢ A single user system where the CPU, memory and hard disks are all dedicated to a single user.
   - ➢ In UNIX, the resources are shared between all users; UNIX is also a multiuser system.

2. **A Multitasking System**
   - ➢ A single user can also run multiple tasks concurrently.
   - ➢ UNIX is a multitasking system.
   - ➢ It is usual for a user to edit a file, print another one on the printer, send email to a friend and browse www- all without leaving any of applications.
   - ➢ The kernel is designed to handle a user's multiple needs.
   - ➢ In a multitasking environment, a user sees one job running in the foreground; the rest run in the background.
   - ➢ User can switch jobs between background and foreground, suspend, or even terminate them.

3. **The Building-block Approach**
   - ➢ The designer never attempted to pack too many features into a few tools.
   - ➢ Instead, they felt "small is beautiful", and developed a few hundred commands each of which performed one simple job.
   - ➢ UNIX offers the | (filters) to combine various simple tools to carry out complex jobs.
   - ➢ By interconnecting a number of tools , we can have a large number of combinations of their usage.
   - ➢ Example:

```
$ cat note                          #cat displays the file contents
WELCOME TO BRCE

$ cat note | wc          #wc counts number of lines, words & characters in the file
1 3 15
```

4. **The UNIX Toolkit**
   - ➢ There are general-purpose tools, text manipulation utilities (filters), compilers and interpreters , networked applications, system administration tools and shells.

5. **Pattern Matching**
   - ➢ UNIX features very sophisticated pattern matching features.
   - ➢ Example: The * (zero or more occurrences of characters) is a special character used by system

to indicate that it can match a number of filenames.

6. **Programming Facility**
   - The UNIX shell is also a programming language; it was designed for programmer, not for end user.
   - It has all the necessary ingredients, like control structures, loops and variables, that establish powerful programming language.
   - These features are used to design shell scripts – programs that can also invoke UNIX commands.
   - Many of the system's functions can be controlled and automated by using these shell scripts.
7. **Documentation**
   - The principal on-line help facility available is the **man** command, which remains the most important references for commands and their configuration files.
   - Apart from the man documentation, there's a vast ocean of UNIX resources available on the Internet.

UNIX Environment and UNIX Structure

- A variable that specifies how an operating system or another program runs, or the devices that the operating system recognizes.
- Set of dynamic values that can affect the way running processes will behave on a computer.
- A string consisting of environment information, such as a drive, path, or filename, associated with a symbolic name that can be used by MS-DOS and Windows.

POSIX AND THE SINGLE UNIX SPECIFICATION

- Dennis Ritchie's decision to rewrite UNIX in C didn't make UNIX portable.
- UNIX fragmentation and absence of single standard affected the development of portable applications.
- First, AT&T creates the System V Interface Definition (SVID).
- Later, X/Open – a consortium of vendors and users, created the X/Open Portability Guide (XPG).
- Products of this specification were UNIX95, UNIX98 and UNIX03.
- Yet another group of standards, the POSIX (Portable Operating System for Computer Environment) were developed by IEEE (Institute of Electrical and Electronics Engineers).
- Two of the most important standards from POSIX are:
- POSIX.1 – Specifies the C application program interface – the system calls (Kernel).

- POSIX.2 – Deals with the Shell and utilities.
- In 2001, a joint initiative of X/Open and IEEE resulted in the unification of two standards.

- This is the Single UNIX Specification, Version 3 (SUSV3).
- The "**Write once, adopt everywhere**" approach to this development means that once software has been developed on any POSIX machine it can be easily ported to another POSIX compliant machine with minimum or no modification.

General features of Unix Commands/Command structure

The following table lists keyboard commands to try when things go wrong.

Table 2.2 Keyboard Commands to Try When Things Go Wrong

Keystroke or Command	*Function*
[Ctrl-h]	Erases text (The *erase* character)
[Ctrl-c] or *[Delete]*	Interrupts a command (The *interrupt* character)
[Ctrl-d]	Terminates login session or a program that expects its input from the keyboard (The *eof* character)
[Ctrl-s]	Stops scrolling of screen output and locks keyboard
[Ctrl-q]	Resumes scrolling of screen output and unlocks keyboard
[Ctrl-u]	Kills command line without executing it (The *line-kill* character)
[Ctrl-\]	Kills running command but creates a core file containing the memory image of the program (The *quit* character)
[Ctrl-z]	Suspends process and returns shell prompt; use `fg` to resume job (The *suspend* character)
[Ctrl-j]	Alternative to *[Enter]*
[Ctrl-m]	As above
`stty sane`	Restores terminal to normal status (a UNIX command)

Basic Unix commands such as echo, printf, ls, who, date,passwd, cal, Combining commands

1. echo: Displaying The Message

- echo command is used is shell scripts to display a message on the terminal, or to issue a prompt for taking user input.

$ echo "Enter your name:\c" Enter your name:$_

$echo $SHELL
/usr/bin/bash

- Echo can be used with different escape sequences

Constant	Meaning
„a'	Audible Alert (Bell)
„b"	Back Space
„f"	Form Feed
„n'	New Line
„r'	Carriage Return
„t'	Horizontal Tab
„v'	Vertical Tab
„\'	Backslash
„\0n'	ASCII character represented by the octal value n

2. printf: An Alternative To Echo

- The printf command is available on most modern UNIX systems, and is the one we can use instead of echo. The command in the simplest form can be used in the same way as echo:

$ printf "Enter your name\n" Enter your name
$_

- printf also accepts all escape sequences used by echo, but unlike echo, it doesn't automatically insert newline unless the \n is used explicitly. printf also uses formatted strings in the same way the C language function of the same name uses them:

$ printf "My current shell is %s\n" $SHELL
My current shell is /bin/bash
$_

- The %s format string acts as a placeholder for the value of $SHELL, and printf replaces %s with the value of $SHELL. %s is the standard format used for printing strings. printf uses many of the formats used by C's printf function. Here are some of the commonly used ones:

%s – String

%30s – As above but printed in a space 30 characters wide

%d – Decimal integer

%6d - As above but printed in a space 30 characters wide

%o – Octal integer

%x – Hexadecimal integer

%f – Floating point number

Example:

$ printf "The value of 255 is %o in octal and %x in hexadecimal\n" 255 255

The value of 255 is 377 in octal and ff in hexadecimal

3. ls:listing directories and files

- The command to list your directories and files is ls.
- With options it can provide information about the size, type of file, permissions, dates of file creation, change and access.

Syntax

ls [options] [argument]

- When no argument is used, the listing will be of the current directory. There are many very useful options for the ls command. A listing of many of them follows. When using the command, string the desired options together preceded by "-".
  - a Lists all files, including those beginning with a dot (.).
  - d Lists only names of directories, not the files in the directory
  - F Indicates type of entry with a trailing symbol: executables with *, directories with / and symbolic links
  - R Recursive list
  - u Sorts filenames by last access time
  - t Sorts filenames by last modification time
  - i Displays inode number
  - l Long listing: lists the mode, link information, owner, size, last modification (time). If the file is a symbolic link, an arrow (-->) precedes the pathname of the linked-to file.
  - x multicolumnar output

$ls –ld helpdir progs

- drwxr-xr-x 2 kumar metal 512 may 9 10:31 helpdir
- drwxr-xr-x 2 kumar metal 512 may 9 09:57

progs output in multiple columns(-x):

$ ls –x

Identifying Directories and executables(-F)

$ ls –Fx

Showing hidden files(-a)

$ ls -axF

Listing directory contents

$ ls -x

4. who: Who Are The Users

- UNIX maintains an account of list of all users logged on to the system. The who command displays an informative listing of these users:

```
$ who
root   tty7       2017-09-04 16:38 (:0)
root   tty17      2017-09-04 16:38 (:0)
$_
```

- Following command displays the header information with –H option,

```
$ who -H
NAME    LINE       TIME
            COMMENT root   tty7   2017-09-
04 16:38 (:0)
$_
```

- -u option is used with who command displays detailed information of users:

```
$ who -Hu
NAME    LINE       TIME           IDLE     PID
COMMENT root     tty7             2017-09-04 16:38   00:18
            1865  (:0)
$_
```

5. date: Displaying The System Date

- One can display the current date with the date command, which shows the date and time to the nearest second:

```
$ date
Mon Sep 4 16:40:02 IST 2017
```

- The command can also be used with suitable format specifiers as arguments. Each symbol is preceded by the + symbol, followed by the % operator, and a single character describing the format. For instance, you can print only the month using the format +%m:

$date +%m
09
Or
the month name name:

$ date +%h
Aug
Or
You can combine them in one command:
$ date + "%h %m"
Aug 08

- There are many other format specifiers, and the useful ones are listed below:
  - d – The day of month (1 - 31)
  - y – The last two digits of the year.
  - H, M and S – The hour, minute and second, respectively.
  - D – The date in the format *mm/dd/yy*
  - T – The time in the format *hh:mm:ss*

6. Passwd: Changing your password

- Password prevents from accessing the system
- If account doesn't have password or has one that is already known to others, should change it immediately.
- This can be done with **passwd** command:

  $ **passwd**
  passwd: changing password for Kumar
  Enter login password: ******* *Asks for old password*
  New password: ********
  Re-enter new password: ********
  passwd (SYSTEM): passwd successfully changed for Kumar
- Passwd expects you to respond three times. First, it primpts for the old password. Next, it schecks whether you have entered a valid password, and if you have, it then prompts for the new password.
- Enter the new password using the password naming rules.
- Finally, passwd asks you to reenter the new password. Later, the new password is registered by the system.
- Rules for framing your own password:
  - Don't choose a password similar to the old one.
  - Don't use commonly used names like names of friends, relatives, pets and so forth.
  - Use a mix of alphabetic or numeric characters.
  - Don't write a password in an easily accessible document.
  - Change the password regularly.

7. cal: The Calendar

- cal command can be used to see the calendar of any specific month or a complete year.

Syntax:

cal [[month] year]

- Everything within the rectangular box in optional. So cal can be used without any arguments, in which case it displays the calendar of the current month

$ cal

```
   September 2017
Su Mo Tu We Th Fr Sa
                1  2
 3 4 5   6     7 8  9
10 11 12  13  14 15 16
17 18 19  20  21 22 23
24 25 26  27  28 29 30
```

- The syntax show that cal can be used with arguments, the month is optional but year is not. To see the calendar of month August 2017, we need to use two arguments as shown below,

$ cal 8 2017

```
    August 2017
Su Mo Tu We Th Fr Sa
       1   2  3 4 5
  6 7 8   9  10 11 12
13 14 15  16  17 18 19
20 21 22  23  24 25 26
27 28 29  30  31
```

- You can't hold the calendar of a year in a single screen page; it scrolls off rapidly before you can use [ctrl-s] to make it pause. To make cal pause using pager using the | symbol to connect them.

$cal 2017 | more

8. Combining commands

- UNIX allows you to specify more than one command in the single command line.

Example:

$ wc note; ls -l note #Two commands combined here using ; (semicolon)

2 3 16 note

-rw-rw-r-- 1 mahesh mahesh 16 Jan 30 09:35 note

In the above example, **wc** command returns number of lines, words and characters in that file, and **ls –l** command returns list all the attributes of the file

- A command line can overflow or be split into multiple lines
- UNIX terminal width is restricted to maximum 80 characters.
- Shell allows command line to overflow or be split into multiple lines.

Example:

```
$ echo "This is                  # $ first prompt
> a three-line                  # > Second prompt
> text message"                 #Command line ends here

This is
a three-line text
message
```

INTERNAL AND EXTERNAL COMMANDS

- When the shell execute command(file) from its own set of built-in commands that are not stored as separate files in /bin directory, it is called **internal command.**
- They can be executed any time and are independent.
- If the command (file) has an independence existence in the /bin directory, it is called **external command.**
- External commands are loaded when the user requests for them. It will have an individual process.

Examples:

```
$ type echo                     # echo is an internal command
echo is shell built-in

$ type ls                       # ls is an external command
ls is /bin/ls
```

- If the command exists both as an internal and external one, shell execute internal command only.
- Internal commands will have top priority compare to external command of same name.

The type command: knowing the type of a command and locating it

- The **type** command is used to describe how its argument would be translated if used as commands. It is also used to find out whether it is built-in or external binary file.

- The type command is shell builtin.

Syntax:

type [Options] command names

- If you want to know the location of executable program (or command), use **type** command-

Example:

$ type date

date is /bin/date

$ **type echo**

echo is a shell builtin

- When you execute **date** command, the shell locates this file in the **/bin** directory and makes arrangements to execute it.

The root login. Becoming the super user: su command.

- The unix system provides a special login name for the exclusive use of the administrator; it is called root. This account doesn't need to be separately created but comes with every system. Its password is generally set at the time of installation of the system and has to be used on logging in:

Becoming the super at login time

login: root

Password: ******* [Enter]**

 # -

- The prompt of root is #
- Once you login as a root you are placed in **root's home directory.** Depending on the system, this directory could be / or /root.
- Administrative commands are resident in /sbin and /usr/sbi modern systems and in older system it resides in /etc.
- Roots PATH list includes detailed path, for example:

/sbin:/bin:/usr/sbin:/usr/bin:/usr/dt/bin

Becoming the super user using *su* command

$ su: Acquiring superuser status

- Any user can acquire superuser status with the su command if they know the root password. For example, the user ***abc*** becomes a superuser in this way.

$ su
Password: ******** #Password of root user**

#pwd
/home/abc

- Though the current directory doesn't change, the # prompt indicates that ***abc*** now has powers of a superuser. To be in root's home directory on superuser login, use **su –l.**
- User's often rush to the administrator with the complaint that a program has stopped running. The administrator first tries running it in a simulated environment. **Su** command when used with a **–** (minus), recreates the user's environment without the login-password route:

$su – abc

- This sequence executes **abc's .**profile and temporarily creates **abc's** environment. su runs in a separate sub-shell, so this mode is terminated by hitting **[ctrl-d]** or using **exit.**

Chapter 2: UNIX files

1. Naming files.
2. Basic file types/categories.
3. Parent child relationship.
4. The home directory and the HOME variable.
5. Reaching required files- the PATH variable, manipulating the PATH, Relative and absolute pathnames.
6. Directory commands – pwd, cd, mkdir, rmdir commands.
7. The dot (.) and double dots (..) notations to represent present and parent directories and their usage in relative path names.
8. File related commands – cat, mv, rm, cp, wc and od commands.

The File

- The file is the container for storing information.
- Neither a file's size nor its name is stored in file.
- All file attributes such as file type, permissions, links, owner, group owner etc are kept in a separate area of the hard disk, not directly accessible to humans, but only to kernel.
- UNIX treats directories and devices as files as well.
- All physical devices like hard disk, memory, CD-ROM, printer and modem are treated as files.

2.1 Naming files

- A filename can consist up to 255 characters.
- File may or may not have extensions, and consist of any ASCII character expect the / & NULL character.
- Users are permitted to use control characters or other unprintable characters in a filename.
- **Examples -** .last_time list. @#$%*abcd a.b.c.d.e
- But, it is recommended that only the following characters be used in filenames-
  - Alphabetic characters and numerals
  - the period(.), hyphen(-) and underscore(_).

2.2 Basic file types/categories

- The UNIX has divided files into three categories:

1. **Ordinary file** – also called as regular file. It contains only data as a stream of characters.
2. **Directory file** – it contains files and other sub-directories.
3. **Device file** – all devices and peripherals are represented by files.

Ordinary File (Regular)

- The most common file type.
- Ordinary file itself can be divided into two types-

1. **Text File** – it contains only printable characters, and you can often view the contents and make sense out of them. e.g.: C and Java program files, shell and perl scripts.
2. **Binary file** – it contains both printable and unprintable characters that cover entire ASCII range. e.g.:- Most Unix commands, executable files, pictures, sound and video files are binary.

Directory File

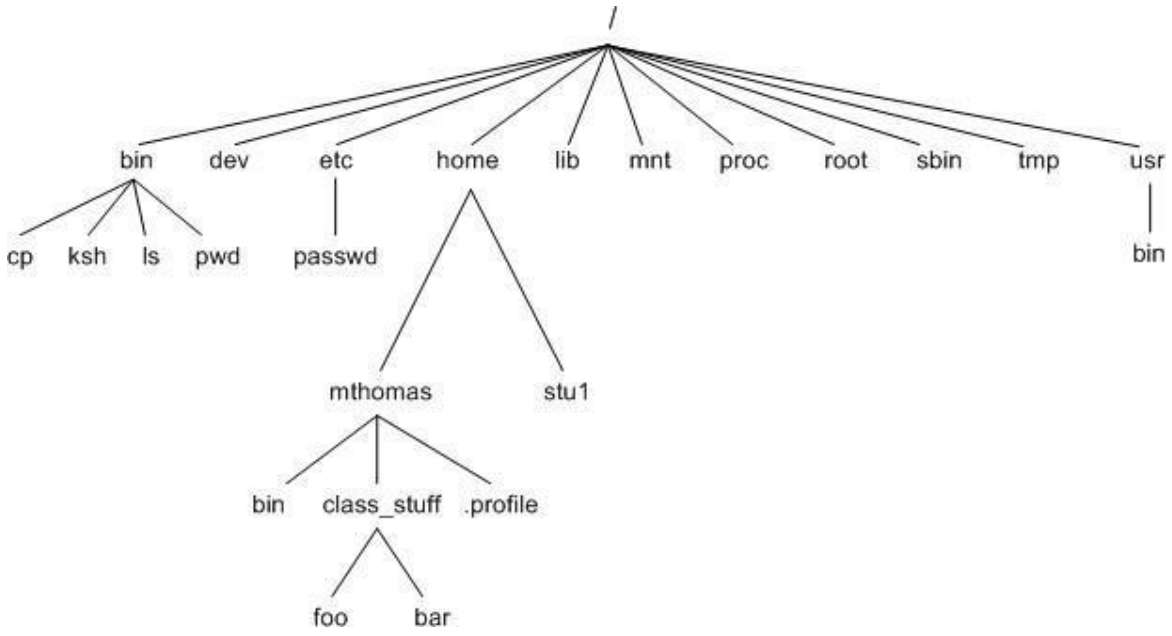
- A directory contains no data but keeps some details of the files and subdirectories that it contains.
- A directory file contains an entry for every file and subdirectories that it houses. If you have 20 files in a directory, there will be 20 entries in the directory.
- Each entry has two components-
  - the filename
  - a unique identification number for the file or directory (called as inode number).
- When any file is created or removed, the kernel automatically updates its corresponding directory by adding or removing the entry (inode number & filename) associated with that file.

Device File

- Installing software from CD-ROM, printing files and backing up data files to tape.
- All of these activities are performed by reading or writing the file representing the device.
- Advantage of device file is that some of the commands used to access an ordinary file also work with device file.
- Device filenames are generally found in a single directory structure, /dev.
- The attributes of every file is stored on the disk.
- The kernel identifies a device from its attributes and then uses them to operate the device.

2.3 Parent child relationship

- The files in UNIX are related to one another.
- The file system in UNIX is a collection of all of these files (ordinary, directory and device files) organized in a Hierarchical (an inverted tree) structure as shown in below figure,



.

- The feature of UNIX file system is that there is a top, which serves as the reference point for all files
- This top is called root and is represented by a / (Front slash).
- The root is actually a directory.
- The root directory (/) has a number of subdirectories under it.
- The subdirectories in turn have more subdirectories and other files under them.
- Every file apart from root, must have a parent, and it should be possible to trace the ultimate parentage of a file to root.
- In parent-child relationship, the parent is always a directory.
- The home directory is the parent of mthomas. / is the parent of home and the grandparent of mthomas.

2.4 The home directory and the HOME variable.

- When you logon to the system, UNIX places you in a directory called home directory.
- It is created by the system when the user account is created.

- If a user login using the login name kumar, user will land up in a directory that could have the path name /home/kumar.
- The shell variable HOME knows the home directory.

$echo $HOME

/home/kumar

2.5 Reaching required files- the PATH variable, manipulating the PATH, Relative and absolute pathnames.

PATH

- A shell variable that contains a colon-delimited list of directories that the shell will lok through to locate a command invoked by user.
- The PATH generally includes /bin and /usr/bin for nonprivileged users and /bin and /usr/sbin for the superuser.
- Use echo to evaluate this variable in which directory list separated by colons:

$ echo $PATH

/bin:/usr/bin:/usr/local/bin:/usr/ccs/bin:/usr/local/java/bin:.
There are six directories in this colon separated list.

- When you issue a command , the shell searches this list in the sequence specified to locate and execute it.

Absolute Pathnames:

- It shows a file's location with reference to the top, i.e., root
- It is simply a sequence of directory names separated by slashes
  e.g.: /home/kumar
- Suppose we are placed in /usr and want to access the file login.sql which is present in /home/kumar, we can give the pathname of the file as below:

cat /home/kumar/login.sql

- If the first character of a pathname is /, the file's location must be determined wrt root(the first /).such a pathname is called an absolute pathname.
- If you know the location of a particular command, you van precede its name with the complete path.
- Since **date** reside in/bin (or /usr/bin), you can use the absolute pathname:

$ **/bin/date**

Thu Sep 1 09:39:55 IST 2020

2.6 The dot (.) and double dots (..) notations to represent present and parent directories and their usage in relative path names.

- Relative path is defined as the path related to the present working directly(pwd). It starts at your current directory and **never starts with a /**.
- A pathname which specifies the location of a file using the symbols. and ..

  . *(single dot) –represents the current directory*

  .. *(two dots) –represents the parent directory*
- Assume that we are in /home/kumar/progs/data/text , we can use .. as an argument to cd to move to the parent directory, /home/kumar/progs/data as shown below:

```
$ pwd
/home/kumar/progs/data/text
$ cd ..                    // moves one level up
$ pwd
/home/kumar/progs/data
```

- To move to /home , we can use the relative path name as follows:

```
$ pwd
/home/kumar/pis
$ cd ../..          // moves two levels up
$ pwd
/home
```

2.7 Directory commands – pwd, cd, mkdir, rmdir commands.

pwd: CHECKING YOUR CURRENT DIRECTORY

- pwd is print working directory
- Any time user can know the current working directory using pwd command.

```
$ pwd
/home/kumar
```

- Like HOME, pwd displays the absolute path.

cd: CHANGING THE CURRENT DIRECTORY

- User can move around the UNIX file system using cd (change directory) command.

When used with the argument, it changes the current directory to the directoryspecified as argument, progs: **$ pwd** /home/kumar **$ cd** progs **$ pwd** /home/kumar/progs	cd can also be used without arguments: **$ pwd** /home/kumar/progs **$cd** **$ pwd** /home/kumar
Here we are using the relative pathname of progs directory. The same can be done with the absolute pathname also. **$cd** /home/kumar/progs **$ pwd** /home/kumar/progs **$cd /bin** **$ pwd** /bin	cd without argument changes the working directory to home directory. **$cd /home/sharma** **$ pwd** /home/sharma **$cd** /home/kumar

mkdir: MAKING DIRECTORIES

- Directories are created with **mkdir** (make directory) command.
- The command is followed by names of the directories to be created. A directory patch is created under current directory like this:

  $mkdir patch
- You can create a number of subdirectories with one **mkdir** command:

  $mkdir patch dba doc
- For instance the following command creates a directory tree:

  $mkdir progs progs/cprogs progs/javaprogs

- This creates three subdirectories – progs, cprogs and javaprogs under progs.
- The order of specifying arguments is important. You cannot create subdirectories before creation of parent directory.
- For instance following command doesn‘t work

  $mkdir progs/cprogs progs/javaprogs progs

  mkdir: Failed to make directory “progs/cprogs”; No such directory mkdir: Failed to make directory “progs/javaprogs”; No such directory

- System refuses to create a directory due to fallowing reasons:

- The directory is already exists.
- There may be ordinary file by that name in the current directory.
- User doesn‘t have permission to create directory

rmdir: REMOVING A DIRECTORY

- The **rmdir** (remove directory) command removes the directories. You have to do this to remove progs:

  $rmdir progs
- If **progs** is empty directory then it will be removed form system.
- Following command used with **mkdir** fails with **rmdir**

$mkdir progs progs/cprogs progs/javaprogs
rmdir: directory “progs”: Directory not empty

- First subdirectories need to be removed from the system then parent.
- Following command works with **rmdir**

$mkdir progs/cprogs progs/javaprogs progs

- First it removes **cprogs** and **javaprogs** form **progs** directory and then it removes **progs** fro system.
- **rmdir : Things to remember**
- You can‘t remove a directory which is not empty
- You can‘t remove a directory which doesn‘t exist in system.
- You can‘t removes a directory if you don‘t have permission to do so.

2.8 File related commands – cat, mv, rm, cp, wc and od commands.

cat: DISPLAYING AND CREATING FILES

- cat command is used to display the contents of a small file on the terminal.

$ cat cprogram.c # include
<stdioh> void main ()

{
Printf(–hello);
}

- As like other files cat accepts more than one filename as arguments

$ cat ch1 ch2

It contains the contents of chapter1
It contains the contents of chapter2

- The contents of the second files are shown immediately after the first file without any header information. So cat concatenates two files- hence its name

cat OPTIONS

Displaying Nonprinting Characters (-v)

- cat without any option it will display text files. Nonprinting ASCII characters can be displayed with –v option.

Numbering Lines (-n)

- -n option numbers lines. This numbering option helps programmer in debugging programs.

Using cat to create a file

- **cat** is also useful for creating a file. Enter the command **cat**, followed by > character and the filename.

$ cat > new

This is a new file which contains some text, just to Add some
contents to the file new
[ctrl-d]
$_

- When the command line is terminated with **[Enter]**, the prompt vanishes. Cat now waits to take input from the user. Enter few lines; press **[ctrl-d]** to signify the end of input to the system
- To display the file contents of new use file name with **cat** command.

$ cat new

This is a new file which contains some text, just to Add some
contents to the file new

mv: RENAMING FILES

- The mv command renames (moves) files. The main two functions are:
  1. It renames a file(or directory)
  2. It moves a group of files to different directory
- It doesn't create a copy of the file; it merely renames it. No additional space is consumed on disk during renaming.

  Eg : To rename the file csb as csa we can use the following command

  $ mv csb csa

- If the destination file doesn‘t exist in the current directory, it will be created. Or else it will just rename the specified file in mv command.
- A group of files can be moved to a directory.

  Eg : Moves three files ch1,ch2,ch3 to the directory module

  $ mv ch1 ch2 ch3 module
- Can also used to rename directory

  $ mv rename newname

- mv replaces the filename in the existing directory entry with the new name. It doesn't create a copy of the file; it renames it.
- Group of files can be moved to a directory
- mv chp1 chap2 chap3 **unix**

rm: DELETING FILES

- The rm command deletes one or more files.

  Eg: Following command deletes three files:

  $ rm mod1 mod2 mod3

  - Can remove two chapters from usp directory without having tocd

    Eg:

    $rm usp/marks ds/marks

  - To remove all file in a directory use ***

    $ rm *
  - Removes all files from that directory

rm options

- **Interactive Deletion (-i) :** Ask the user confirmation before removing each file:

  $ rm -i ch1 ch2

  rm: remove ch1 (yes/no)? ? y

  rm: remove ch1 (yes/no)? ? n [Enter]

- A y removes the file (ch1) any other response like n or any other key leave the file undeleted.
- **Recursive deletion (-r or -R):** It performs a recursive search for all directories and files within these subdirectories. At each stage it deletes everything it finds.

  $ rm -r * ***Works as rmdir***

- It deletes all files in the current directory and all its subdirectories.
- **Forcing Removal (-f): rm** prompts for removal if a file is **write-protected.** The **-f** option overrides this minor protection and forces removal.

  rm -rf* ***Deletes everything in the current directory and below***

cp: COPYING A File

- The **cp** command copies a file or a group of files. It creates an exact image of the file on the disk with a different name. The **syntax takes two filename** to be specified in the command line.
- When both are ordinary files, **first file is copied to second.**

  $ cp csa csb

- If the destination file (csb) doesn't exist, it will first be created before copying takes place. If not it will simply be overwritten without any warning from the system.

Example to show two ways of copying files to the cs directory:

$ cp ch1 cs/module1 #ch1 copied to module1 under cs

$cpch1cs #ch1 retains its name under cs

- cp can also be used with the shorthand notation, .(dot), to signify the current directory as the destination. To copy a file **„new** from /home/user1 to your current directory, use the following command:

  $cp /home/user1/new **#**new *destination is a file*

  $cp /home/user1/new . *#destination is the current directory*

- cp command can be used **to copy more than one file with a single invocation of the command.** In this case the last filename must be a directory.
  Eg: To copy the file ch1,chh2, ch3 to the module , use cp as

  $ cp ch1 ch2 ch3 module

- The files will have the same name in **module**. If the files are already resident in **module**, they will be overwritten. In the above diagram module directoryshould already exist and cp doesn‘t able create a directory.
- UNIX system uses * as a shorthand for multiple filenames.

  Eg:

  $ cp ch* usp #Copies all the files beginning with ch

cp options

- **Interactive Copying(-i) :** The **–i** option warns the user before overwriting the destination file, If unit 1 exists, cp prompts for response

  $ cp -i ch1 unit1

  $ cp: overwrite unit1 (yes/no)? Y

- A y at this prompt overwrites the file, any other response leaves it uncopied.

Copying directory structure (-R) :

- It performs recursive behavior command can descend a directory and examine all files in its subdirectories.
- ***-R : behaves recursively to copy an entire directory structure***

  $ cp -R usp newusp

  $ cp -R class newclass

- If the **newclass/newusp** doesn‘t exist, **cp** creates it along with the associated subdirectories.

wc: COUNTING LINES,WORDS AND CHARACTERS

- wc command performs Word counting including counting of lines and characters in a specified file. It takes one or more filename as arguments and displays a four columnar output.

$ wc ofile

4 20 97 ofile

- Line: Any group of characters not containing a newline
- Word: group of characters not containing a space, tab or newline
- Character: smallest unit of information, and includes a space, tab and newline
- **wc** offers 3 options to make a specific count. –l option counts only number of lines, - w and –c options count words and characters, respectively.

  $ wc -l ofile

  4 ofile

  $ wc -w ofile

  20 ofile

  $ wc -c ofile

  97 ofile
- Multiple filenames, **wc** produces a line for each file, as well as a total count.

  $ wc chap01 chap02 chap03

od: DISPLAYING DATA IN OCTAL

- **od** command displays the contents of executable files in a **ASCII octal value.**

  $ more ofile

  this file is an example for od command

  ^d used as an interrupt key
- -b option displays this value for each character separately.
- Each line displays 16 bytes of data in octal, preceded by the **offset in the file of the first byte in the line.**

  $ od –b file

  0000000 164 150 151 163 040 146 151 154 145 040 151 163 040 141 156 040
  0000020 145 170 141 155 160 154 145 040 146 157 162 040 157 144 040 143
  0000040 157 155 155 141 156 144 012 136 144 040 165 163 145 144 040 141
  0000060 163 040 141 156 040 151 156 164 145 162 162 165 160 164 040 153
  0000100 145 171

- **-c character option**

- Now it shows the printable characters and its corresponding ASCII octal representation

$ od –bc file

```
od -bc ofile
0000000 164 150 151 163 040 146 151 154 145 040 151 163 040 141 156 040
          T   h   i   s       f   i   l   e           i   s       a   n 0000020 145 170
141 155 160 154 145 040 146 157 162 040 157 144 040 143
          e   x   a   m   p   l   e       f   o   r       o   d       c 0000040 157

155 155 141 156 144 012 136 144 040 165 163 145 144 040 141
          o   m   m   a   n   d  \n   ^   d       u   s   e   d       a 0000060 163
040 141 156 040 151 156 164 145 162 162 165 160 164 040 153
          s       a   n       i   n   t   e   r   r   u   p   t       k 0000100 145
171
          e   y
```

Some of the representation:

- The tab character, [ctrl-i], is shown as \t and the octal value 011
- The bell character , [ctrl-g] is shown as 007, some system show it as \a
- The form feed character,[ctrl-l], is shown as \f and 014
- The LF character, [ctrl-j], is shown as \n and 012
- od makes the newline character visible too.

Chapter 1: File attributes and permissions

1. The ls command with options.
2. Changing file permissions: the relative and absolute permissions changing methods.
3. Recursively changing file permissions.
4. Directory permissions.

1. The ls command with options.

ls –l :LISTING FILE ATTRIBUTES

- ls command is used to obtain a list of all filenames in the current directory. The output in UNIX lingois often referred to as the listing.
- Sometimes we combine this option with other options for displaying other attributes, or ordering thelist in a different sequence.
- ls look up the file‘s inode to fetch its attributes.
- It lists **seven attributes of all files** in the current directory and theyare:

- **File type and Permissions**
  - The file type and its permissions are associated with each file.
- **Links**
  - Links indicate the number of file names maintained by the system. This does not mean that there are so many copies of the file.
- **Ownership**
  - File is created by the owner. The one who creates the file is the owner of that file.
- **Group ownership**
  - Every user is attached to a group owner. Every member of that group can access the file depending on the permission assigned.
- **File size**
  - File size in bytes is displayed. It is the number of character in the file rather than the actual size occupied on disk.
- **Last Modification date and time**
  - Last modification time is the next field. If you change only the permissions or ownership of the file, the modification time remains unchanged. If at least one character is added or removed from the file then this field will be updated.
- **File name**
  - In the last field, it displays the file name.

For example,

$ ls -l

total 72

-rw-r--r-- 1 kumar metal 19514 may 10 13:45 chap01

-rw-r--r-- 2 kumar metal 19555 may 10 15:45 chap02

drwxr-xr-x 2 kumar metal 512 may 09 12:55 helpdir

drwxr-xr-x 3 kumar metal 512 may 09 11:05 progs

The –d option: Listing Directory Attributes

$ls -d

This command will not list all subdirectories in the current directory .

For example,

$ls –ld helpdir progs

drwxr-xr-x 2 kumar metal 512 may 9 10:31 helpdir

drwxr-xr-x 2 kumar metal 512 may 9 09:57 progs

- Directories are easily identified in the listing by the first character of the first column, which here shows d.
- The significance of the attributes of a directory differs a good deal from an ordinary file.
- To see the attributes of a directory rather than the files contained in it, use ls –ld with the directory name. Note that simply using ls –d will not list all subdirectories in the current directory. Strange though it may seem, ls has no option to list onlydirectories.

2. Changing file permissions: the relative and absolute permissions changing methods.

File Ownership

- When you create a file, you become its owner. Every owner is attached to a group owner.
- Several users may belong to a single group, but the privileges of the group are set by the owner of the fileand not by the group members.
- When the system administrator creates a user account, he has to assign these parameters to the user:

The user-id (UID) – both its name and numeric representation. The group-id (GID) – both its name and numeric representation

File Permissions

- UNIX follows a three-tiered file protection system that determines a file‘s access rights.

- It is displayed in the following format: Filetype owner (rwx) groupowner (rwx) others (rwx).

For Example:

-rwxr-xr- - 1 kumar metal 20500 may 10 19:21 chap02

rwx	**r-x**	**r- -**
owner/user	group owner	others

- The first group(rwx) has all three permissions. The file is readable, writable and executable by the owner ofthe file.
- The second group(r-x) has a hyphen in the middle slot, which indicates the absence of write permission bythe group owner of the file.
- The third group(r- -) has the write and execute bits absent. This set of permissions is applicable to others.
- You can set different permissions for the three categories of users – owner, group and others.

Changing File Permissions

- A file or a directory is created with a default set of permissions, which can be determined by umask.
- Let us assume that the file permission for the created file is -rw-r-- r--. Using **chmod** command, we canchange the file permissions and allow the owner to execute his file.

The command can be used in two ways:

- In a **relative** manner by specifying the changes to the current permissions
- In an **absolute** manner by specifying the final permissions

Relative Permissions

- chmod only changes the permissions specified in the command line and leaves the other permissionsunchanged.
- Its **syntax** is:

  chmod *category operation permission filename(s)*
- chmod takes an expression as its argument which contains:
  - ✓ user category (user, group, others)
  - ✓ operation to be performed (assign or remove a permission)
  - ✓ type of permission (read, write, execute)
- **Category: u – user g – group o – others a - all (ugo)**
- **Operations : + assign - remove = absolute**
- **Permissions: r – read w – write x - execute**

Let us discuss some examples:

- Initially,

  -rw-r—r-- 1 kumar metal 1906 sep 23:38 xstart

  $chmod u+x xstart

  -rwxr—r-- 1 kumar metal 1906 sep 23:38 xstart

- The command assigns (+) execute (x) permission to the user (u), other permissions remain unchanged.

  $chmod ugo+x xstart or chmod a+x xstart or chmod +x xstart

  $ls –l xstart

  -rwxr-xr-x 1 kumar metal 1906 sep 23:38 xstart

- chmod accepts multiple file names in command line

  $chmod u+x note note1 note3

- Let initially,

  -rwxr-xr-x 1 kumar metal 1906 sep 23:38 xstart

  $chmod go-r xstart

- Then, it becomes

  $ls –l xstart

  -rwx—x--x 1 kumar metal 1906 sep 23:38 xstart

Absolute Permissions

- Here, we need not to know the current file permissions. We can set all nine permissions explicitly.
- A string of three octal digits is used as an expression.
- The permission can be represented by one octal digit for each category. For each category, we addoctal digits.
- If we represent the permissions of each category by one octal digit, this is how the permission can berepresented:

  Read permission – 4 (octal 100)

  Write permission – 2 (octal 010)

  Execute permission – 1 (octal 001)

Octal	Permissions	Significance
0	---	no permissions

1	--x	execute only
2	-w-	write only
3	-wx	write and execute
4	r--	read only
5	r-x	read and execute
6	rw-	read and write
7	Rwx	read, write and execute

- We have three categories and three permissions for each category, so three octal digits can describe a file‘s permissions completely. The most significant digit represents user and the least one represents others. chmod can use this three-digit string as the expression.
- Using relative permission, we have,

  $chmod a+rw xstart
- Using absolute permission, we have,

  $chmod 666 xstart

  -rw-rw-rw- 1 kumar metal 1906 sep 10 23:38 xstart

  $chmod 644 xstart

  -rw-r—r-- 1 kumar metal 1906 sep 10 23:38 xstart

  $chmod 761 xstart

  -rwxrw--x 1 kumar metal 1906 sep 10 23:38 xstart

- will assign all permissions to the owner, read and write permissions for the group and only execute permission to the others.
- 777 signify all permissions for all categories, but still we can prevent a file from being deleted.
- 000 signifies absence of all permissions for all categories, but still we can delete a file.
- It is the directory permissions that determine whether a file can be deleted or not.
- Only owner can change the file permissions. User cannot change other user‘s file‘s permissions.
- But the system administrator can do anything.

The Security Implications

- Let the default permission for the file xstart is

  -rw-r—r- - 1 kumar metal 1906 sep 10 23:38 xstart

  $chmod u-rw, go-r xstart or

$chmod 000 xstart

____- 1 kumar metal 1906 sep 10 23:38 xstart

- This is simply useless but still the user can delete this file.
- On the other hand,

$chmod a+rwx xstart or chmod 777 xstart

-rwxrwxrwx 1 kumar metal 1906 sep 10 23:38 xstart

- The UNIX system by default, never allows this situation as you can never have a secure system. Hence,directory permissions also play a very vital role here .

3. Using chmod Recursively

$chmod -R a+x shell_scripts

- This makes all the files and subdirectories found in the shell_scripts directory, executable by all users.

Chmod –R 755 . //work on hidden file also
Chmod –R a+X * //Leaves out hidden files

- When you know the shell meta characters well, you will appreciate that the * doesn‘t match filenames beginning with a dot. The dot is generally a safer but note that both commands change the permissions of directories also.

4. Directory Permissions

- It is possible that a file cannot be accessed even though it has read permission, and can be removed even when it is write protected. The default permissions of a directory are,

rwxr-xr-x (755)

- A directory must never be writable by group and others.
- Example:

$mkdir c_progs
$ls –ld c_progs

drwxr-xr-x 2 kumar metal 512 may 9 09:57 c_progs

- If a directory has write permission for group and others also, be assured that every user can remove every file in the directory. As a rule, you must not make directories universally writable unless you have definitereasons to do so.

Changing File Ownership

- Usually, on BSD and AT&T systems, there are two commands meant to change the ownership of a file ordirectory. Let kumar be the owner and metal be the group owner. If sharma copies a file of kumar, then sharma will become its owner and he can manipulate the attributes.
- chown changing file owner and chgrp changing group owner
- On BSD, only system administrator can use chown
- On other systems, only the owner can change both

chown

- Changing ownership requires super user permission, so use su command

$ls -l note
-rwxr ---x 1 kumar metal 347 may 10 20:30 note

$chown sharma note; ls -l note
-rwxr ---x 1 sharma metal 347 may 10 20:30 note

- Once ownership of the file has been given away to sharma, the user file permissions that previously applied to Kumar now apply to sharma. Thus, Kumar can no longer edit note since there is no write privilege for group and others. He cannot get back the ownership either. But he can copy the file to his own directory, in which case he becomes the owner of the copy.

chgrp

- This command changes the file‘s group owner. No super user permission is required.

#ls –l dept.lst
-rw-r—r-- 1 kumar metal 139 jun 8 16:43 dept.lst

#chgrp dba dept.lst; ls –l dept.lst
-rw-r—r-- 1 kumar dba 139 Jun 8 16:43 dept.lst

Chapter 2: The Shells Interpretive Cycle

1. Wild cards.
2. Removing the special meanings of wild cards.
3. Three standard files and redirection.
4. Connecting commands: Pipe.
5. Basic and Extended regular expressions.
6. The grep, egrep.
7. Typical examples involving different regular expressions.

THE SHELL

- Shell acts as both a command interpreter as well as a programming facility.

The shell and its interpretive cycle

- The shell sits between you and the operating system, acting as a command interpreter.
- It reads your terminal input and translates the commands into actions taken by the system.
- When you log into the system you are given a default shell.
- When the shell starts up it reads its startup files and may set environment variables, command search paths, and command aliases, and executes any commands specified in these files.
- The original shell was the Bourne shell, sh. Every Unix platform will either have the Bourneshell, or a Bourne compatible shell available.
- Even though the shell appears not to be doing anything meaningful when there is no activity at the terminal, it swings into action the moment you key in something.

The following activities are typically performed by the shell in its interpretive cycle:

- The shell issues the prompt and waits for you to enter a command.
- After a command is entered, the shell scans the command line for meta characters and expands abbreviations (like the * in rm *) to recreate a simplified command line.
- It then passes on the command line to the kernel for execution.
- The shell waits for the command to complete and normally can't do any work while thecommand is running.
- After the command execution is complete, the prompt reappears and the shell returns to its waiting role to start the next cycle. You are free to enter another

command.

1. Wild-Cards

- A pattern is framed using ordinary characters and a meta character (like *) using well-defined rules.
- The pattern can then be used as an argument to the command, and the shell will expand itsuitably before the command is executed.
- The meta characters that are used to construct the generalized pattern for matching filenamesbelong to a category called wild-cards.

The following table lists them:

Wild-card	Matches
*	Any number of characters including none
?	A single character
[ijk]	A single character – either an i, j or k
[x-z]	A single character that is within the ASCII range of characters x and z
[!ijk]	A single character that is not an i, j or k (Not in C shell)
[!x-z]	A single character that is not within the ASCII range of the characters x and z (Not in C Shell)
{pat1,pat2...}	Pat1, pat2, etc. (Not in Bourne shell)

The * and ?

- The metacharacter *, is one of the characters of the shell's wild card set.
- It matches any number of characters (including none).
- To list all files that begin with **chap**.

  $ ls chap*

  chap chap01 chap02 chap03 chap04 chap15 chap16 chap17 chapx chapy chapz
- chap* matches the string chap. When the shell encounters this command line, it identifies the * immediately as a wild card.
- It then looks in the current directory and recreates the command line as below from the filenames that match the pattern chap*:

  ls chap chap01 chap02 chap03 chap04 chap15 chap16 chap17 chapx chapy chapz

- ? matches a single character.
- To list all files whose filenames are six character long and start with chap.

  $ ls chap?

  chapx chapy chapz

  $ ls chap??

chap01 chap02 chap03 chap04 chap15 chap16 chap17

Matching the Dot

- Both * and ? operate with some restrictions. for example, the * doesn't match all filesbeginning with a . (dot) or the / of a pathname.
- If you wish to list all hidden filenames in your directory having at least three characters afterthe dot, the dot must be matched explicitly.

  $ ls .???*

  .bash_profile .exrc .netscape .profile
- However, if the filename contains a dot anywhere but at the beginning, it need not be matched explicitly.
- Similarly, these characters don't match the / in a pathname. So, you cannot use, **$cd /usr?local** to change to **/usr/local**.

The character class

- The character class comprises a set of characters enclosed by the rectangular brackets, [and], butit matches a single character in the class.
- The pattern [abd] is character class, and it matches a single character – an a,b or d.

Examples:

$ ls chap0[124]
chap01 chap02 chap04

$ls chap[x-z]
chapx chapy chapz

Negating the character class(!)

- You can negate a character class to reverse matching criteria. For example,

***.[!co] -** To match all filenames with a single-character extension but not the .c or .o files

[!a-zA-Z]* - To match all filenames that don't begin with an alphabetic character

2. Removing the special meanings of wild cards.

Escaping and Quoting

- Escaping is providing a \ (backslash) before the wild-card to remove (escape) its special meaning.
- For instance, if we have a file whose filename is **chap*** (Remember a file in UNIX can be names with virtually any character except the / and null), to remove the file, it is dangerous to give command as **rm chap***, as it will remove all files beginning with chap.
- Hence to suppress the special meaning of *, use the command **rm chap*.**

- To list the contents of the file **chap0[1-3]**, use ,
  $cat chap0\[1-3\]

- A filename can contain a whitespace character also. Hence to remove a file named **MyDocumend.doc**, which has a space embedded, a similar reasoning should be followed:
  $rm My\ Document.doc

- Quoting is enclosing the wild-card, or even the entire pattern, within quotes. Anythingwithin these quotes (barring a few exceptions) are left alone by the shell and not interpreted.
- When a command argument is enclosed in quotes, the meanings of all enclosed specialcharacters are turned off.

Examples:

$ echo '\'	*Displays a *
$ rm 'chap*'	*Removes file chap**
$ rm 'My Document.doc'	*Removes file My Document.doc*

3. Three standard files and redirection.

- The shell associates three files with the terminal – two for display and one for the keyboard.
- These files are streams of characters which many commands see as input and output.
- When a user logs in, the shell makes available three files representing three streams. Each stream is associated with a default device: -

Standard input: The file (stream) representing input, connected to the keyboard.
Standard output: The file (stream) representing output, connected to the display.
Standard error: The file (stream) representing error messages that emanate from the command or shell, connected to the display.

Standard input

The standard input can represent three input sources:

- The keyboard, the default source.
- A file using redirection with the < symbol.
- Another program using a pipeline.

How input redirection works:

$ wc < sample.txt

1. On seeing the <, the shell opens the disk file, sample.txt for reading
2. It unplugs the standard input file from its default source and assigns it to sample.txt
3. wc reads from standard input which has earlier been reassigned by the shell to sample.txt

Standard output

The standard output can represent three possible destinations:

- The terminal, the default destination.
- A file using the redirection symbols > and >>.
- As input to another program using a pipeline.

How output redirection works:

$ wc sample.txt > newfile

1. On seeing the >, the shell opens the disk file, newfile for writing
2. It unplugs the standard output file from its default destination and assigns it to newfile.
3. wc writes to standard output which has earlier been reassigned by the shell to newfile

Standard error:

- A file is opened by referring to its pathname, but subsequent read and write operationsidentify the file by a unique number called a file descriptor.
- The kernel maintains a table of file descriptors for every process running in the system.
- The first three slots are generally allocated to the three standard streams as,0 – Standard input

  1 – Standard output 2 – Standard error

These descriptors are implicitly prefixed to the redirection symbols.

4. Connecting commands: Pipe

- With piping, the output of a command can be used as input (piped) to a subsequent command.

  Syntax:

  $ command1 | command2
- Output from command1 is piped into input for command2. Thisis equivalent to, but more efficient than:

  $ command1 > temp

  $ command2 < temp

  $ rm temp
- To count the number of lines and redirect wc's input so that the filename doesn't appear in the output:

  $wc –l > user.txt

  5
- Using an intermediate file(user.txt), we counted the number of lines.

- Method of running two commands seperately has 2 disadvantages:

1. For long running commands, this process is slow. The second command cant act unless the first hascompleted its job.
2. You need an intermediate file that has to be removed after completion of the job. When handling largefiles, temporary files can build up easily and eat up disk space in no time.

- The shell can connect these streams using a special operator, the | (pipe) and avoid creation of the diskfile.
- The pipe is the third source and destination of standard input and standard outputExamples

$ ls -l | wc –l **Displays number of file in current directory**

$ who | wc –l **Displays number of currently logged in users**

- In a pipeline, all programs run simultaneously. A pipe has a built in mechanism to control the flow of thestream.
- Pipe is both being read and written, the reader and writer have to act in unison.
- If one operates faster than the other, then the appropriate driver has to readjust the flow.

5. The grep, egrep.

grep: Searching for a pattern

- grep scans its input for a pattern displays lines containing the pattern, the line numbers or filenameswhere the pattern occurs. The command uses the following syntax:
  $grep options pattern filename(s)

- grep searches for pattern in one or more filename(s), or the standard input if no filename is specified.

- The first argument (except the options) is the pattern and the remaining arguments are filenames.

Examples:

$ grep "sales" emp.lst

2233|a. k. Shukla |g. m. |sales |12/12/52|6000
1006|chanchal singhvi |director |sales |03/09/38|6700
1265|s. n. dasgupta |manager |sales |12/09/63|5600
2476|anil Aggarwal |manager |sales |01/05/59|5000
here, grep displays 4 lines containing pattern as "sales" from the file emp.lst.

- grep silently returns the prompt because no pattern as "president" found in file emp.lst.

$ grep president emp.lst #No quoting necessary here
$ _#No pattern (president) found

- when grep is used with multiple filenames, it displays the filenames along with the output.

$ grep "director" emp1.lst emp2.lst

emp1.lst: 9876|jai sharma |director |production
|12/03/50|7000
emp1.lst: 2365|barun sengupta |director |personnel
|11/05/47|7800 emp1.lst: 1006|chanchal singhvi |director |sales
|03/09/38|6700
emp2.lst: 6521|lalit chowdury |director |marketing
|26/09/45|8200

Here, first column shows file name.

grep options:

The below table shows all the options used by grep.

Option	Significance
-i	Ignores case for matching
-v	Doesn't display lines matching expression
-n	Displays line numbers along with lines
-c	Displays count of number of occurrences
-l	Displays list of filenames only
-e exp	Matches multiple patterns
-f filename	Takes patterns from file, one per line
-E	Treats patterns as an ERE
-F	Matches multiple fixed strings

- **Ignoring case (-i):**

When you look for a name but are not sure of the case, use the -i (ignore) option.

$ grep -i 'agarwal' emp.lst

3564|sudhir Agarwal |executive |personnel |06/07/47|7500
This locates the name Agarwal using the pattern agarwal.

- **Deleting Lines (-v):**

The -v option selects all the lines except those containing the pattern.

It can play an inverse role by selecting lines that does not containing the pattern.

$ grep -v 'director' empl.lst

```
2233|a. k. shukla        |g. m.     |sales      |12/12/52|6000
5678|sumit chakrobarty   |d. g. m.  |marketing  |19/04/43|6000
5423|n. k. gupta         |chairman  |admin      |30/08/56|5400
6213|karuna ganguly      |g. m.     |accounts   |05/06/62|6300
1265|s. n. dasgupta      |manager   |sales      |12/09/63|5600
4290|jayant choudhury    |executive |production |07/09/50|6000
2476|anil Aggarwal       |manager   |sales      |01/05/59|5000
3212|shyam saksena       | d. g. m. |accounts   |12/12/55|6000
3564|sudhir Agarwal      |executive |personnel  |06/07/47|7500
2345|j. b. saxena        |g. m.     |marketing  |12/03/45|8000
0110|v. k. Agrawal       |g. m.     |marketing  |31/12/40|9000
```

- **Displaying Line Numbers (-n):**

The -n(number) option displays the line numbers containing the pattern, along with thelines.

$ grep -n 'marketing' emp.lst

```
3: 5678 |sumit chakrobarty |d. g. m.  |marketing |19/04/43|6000
11: 6521|lalit chowdury    |director  |marketing |26/09/45|8200
14: 2345|j. b. saxena      |g. m.     |marketing |12/03/45|8000
15: 0110|v. k. Agrawal     |g. m.     |marketing |31/12/40|9000
```

here, first column displays the line number in emp.lst where pattern is found

- **Counting lines containing Pattern (-c):**

How many directors are there in the file emp.lst?
The -c(count) option counts the number of lines containing the pattern.

$ grep -c 'director' emp.lst
4

- **Matching Multiple Patterns (-e):**

With the -e option, you can match the three agarwals by using the grep like this:

$ grep -e "Agarwal" -e "aggarwal" -e "agrawal" emp.lst
2476|anil aggarwal |manager |sales |01/05/59|5000
3564|sudhir Agarwal |executive |personnel |06/07/47|7500
0110|v. k. agrawal |g. m. |marketing |31/12/40|9000

- **Taking patterns from a file (-f):**

You can place all the patterns in a separate file, one pattern per line.Grep uses -f option to take patterns from a file:

$ grep -f patterns.lst emp.lst

9876|jai sharma |director |production |12/03/50|7000
2365|barun sengupta |director |personnel |11/05/47|7800
5423|n. k. gupta |chairman |admin |30/08/56|5400
1006|chanchal singhvi |director |sales |03/09/38|6700
1265|s. n. dasgupta |manager |sales |12/09/63|5600
2476|anil aggarwal |manager |sales |01/05/59|5000
6521|lalit chowdury |director |marketing |26/09/45|8200

Basic Regular Expression (Bre)

- Like the shell's wild-cards which matches similar filenames with a single expression, grepuses an expression of a different type to match a group of similar patterns.
- Unlike shell's wild-cards, grep uses following set of meta-characters to design an expressionthat matches different patterns.
- If an expression uses any of these meta-characters, it is termed as **Regular Expression (RE).**

The below table shows the BASIC REGULAR EXPRESSION (BRE) character set-

Symbols or Expression	Matches
*	Zero or more occurrences of the previous character
g*	Nothing or g, gg, ggg, gggg, etc.
.	A single character
.*	Nothing or any number of characters
[pqr]	A single character p, q or r
[c1-c2]	A single character withing ASCII range shown by c1 and c2
[0-9]	A digit between 0 and 9
[^pqr]	A single character which is not a p, q or r
[^a-zA-Z]	A non-alphabetic character
^pat	Pattern pat at beginning of line
pat$	Pattern pat at end of line

^bash$	A bash as the only word in line
^$	Lines containing nothing

The character class

- A RE lets you specify a group of characters enclosed within a pair of rectangular brackets, [ ], in whichcase the match is performed for a single character in the group.

$ grep '[aA]g[ar][ar]wal' emp.lst

3564|sudhir Agarwal |executive |personnel |06/07/47|7500
0110|v. k. Agrawal |g. m. |marketing |31/12/40|9000

The *

- The * (asterisk) refers to the immediately preceding character.
- Here, it indicates that the previous character can occur many times, or not at all.

$ grep '[aA]gg*[ar][ar]wal' emp.lst

2476|anil Aggarwal |manager |sales |01/05/59|5000
3564|sudhir Agarwal |executive |personnel |06/07/47|7500
0110|v. k. Agrawal |g. m. |marketing |31/12/40|9000

The Dot

- A . matches a single character.
- The pattern 2... matches a four-character patten beginning with a 2.
- The pattern .* matches any number of characters, or none.

$ grep 'j.*saxena' emp.lst

2345|j. b. saxena |g. m. |marketing |12/03/45|8000

Specifying pattern locations (^ and $)

^ (carat) – For matching at the beginning of a line

$ (dollar) – For matching at the end of a line

$ grep '^2' emp.lst

2233|a. k. shukla |g. m. |sales |12/12/52|6000
2365|barun sengupta |director |personnel |11/05/47|7800
2476|anil Aggarwal |manager |sales |01/05/59|5000

2345|j. b. saxena |g. m. |marketing |12/03/45|8000

$ grep '7...$' emp.lst

9876|jai sharma |director |production |12/03/50|7000
2365|barun sengupta |director |personnel |11/05/47|7800
3564|sudhir Agarwal |executive |personnel |06/07/47|7500

6. Extended Regular Expression (Ere) and Egrep

- **ERE** make it possible to match dissimilar patterns with a single expression.
- **grep** uses ERE characters with -E option.
- **egrep** is another alternative to use all the ERE characters without -E option. This **ERE** uses someadditional characters set shown in below table-

Expression	Significance
ch+	Matches one or more occurrences of character ch
ch?	Matches zero or one occurrence of character ch
exp1 | exp2	Matches exp1 or exp2
GIF | JPEG	Matches GIF or JPEG
(x1|x2)x3	Matches x1x3 or x2x3
(hard|soft)ware	Matches hardware or software

The + and ?

+ - Matches one or more occurrences of the previous character
? - Matches zero or one occurrence of the previous character.

$ grep -E "[aA]gg?arwal" emp.lst

2476|anil aggarwal |manager |sales |01/05/59|5000
3564|sudhir Agarwal |executive |personnel |06/07/47|7500

Matching Multiple Patterns(|, (and))

$ grep -E 'sengupta|dasgupta' emp.lst

2365|barun sengupta |director |personnel |11/05/47|7800
1265|s. n. dasgupta |manager |sales |12/09/63|5600

$ grep -E '(sen|das)gupta' emp.lst

2365|barun sengupta |director |personnel |11/05/47|7800
1265|s. n. dasgupta |manager |sales |12/09/63|5600

Chapter 3: SHELL PROGRAMMING

1. Ordinary and environment variables.
2. The .profile.
3. Read and readonly commands
4. Command line arguments
5. . exit and exit status of a command.
6. Logical operators for conditional execution.
7. The test command and its shortcut.
8. The if, while, for and case control statements.
9. The set and shift commands and handling positional parameters.
10. The here (<<) document and trap command.
11. Simple shell program examples.

1. Ordinary and environment variables.

A local variable is a variable that is present within the current instance of the shell. It is not available to program that are started by the shell. They are set at command prompt.

VARIABLE NAMES:

- A variable is a character string to which we assign a value. The value assigned could be a number, text, filename, device, or any other type of data.
- A variable is nothing more than a pointer to the actual data. The shell enables you to create, assign, and delete variables.
- The name of a variable can contain only letters(a to z or A to Z),numbers(0 to 9) or the underscore character(_).
- By convention, Unix shell variables would have their names in UPPERCASE
- The following examples are valid variable names:-
  VAR_1
  VAR_2
  TOKEN_A

DEFINING VARIABLE:

- Variable are defiened as follows:
- Variable_name= variable_value
- For example
  NAME=”Sumithabha Das”

ACCESSING VARIABLES:

- To access the value stored in a variable, prefix its name with the dollar sign($).
- For example following script would access the value of defined variable NAME and would

print it on STDOUT
#!/bin/sh
NAME=”Sumitabha Das”
Echo $NAME

ENVIRONMENT VARIABLES:

An environment variables that is available to any child process of the shell. Some programs need environment variables in order to function correctly. Usually a shell script defines only those environment variables that are needed by the programs that it runs

- **SHELL:** points to the shell defined as default.
- **DISPLAY:** contains the identifier for the display that X11 programs should use by default.
- **HOME:** Indicates the home directory of the current user default argument for the cd built in command
- **IFS:** Indicates the Internal Field Separator that is used by the parser for word splitting after expansion.
- **PATH**: Indicates search path for commands .It is a colon separated list of directories in which shell looks the command.
- **PWD:** Indicates the current working directory as set by the cd command.
- **RANDOM**: Generates a random interger between 0 and 32767 each time it s referenced.
- **SHLVL**: Increments by one each time an instance of bash is created.
- **UID:** Expands to the numeric user ID of the current user initialized at shell prompt.

Following is the sample example showing few environment variables

```
$ echo $HOME
/root
]$ echo $DISPLAY

$ echo $TERM
xterm
$ ech $PATH
/usr/local/bin:/bin:/usr/bin:/home/amrood/bin:/usr/local/bin
$
```

PS1(Prompt String one) and Environment variables

The characters that the shell displays as your command prompt are stored in the variables PS1.You can change this variable to be anything you want .As soon as you change it it’ll be used by the shell from that point on.

For example, if you issued the command:

```
$PS1=’=>’
=>
=>
```

Your prompt would become=>

When you issue a command that is incomplete, the shell will display a secondary prompt and wait

for you to complete the command and hit Enter again.

The default secondary prompt is > (the greater than sign), but can be changed by re-defining the **PS2** shell variable:

Following is the example which uses the default secondary prompt:

```
$ echo "this is a
>test"
This is a
test
$
```

```
$ PS='-->'
$ echo "this is a
-->test"
```

2. The .profile File

- The file **/etc/profile** is maintained by the system administrator of your UNIX machine and contains shell initialization information required by all users on a system.
- The file **.profile** is under your control .you can add as much shell customization information as you want to this file. The minimum set fo information that you need to configure includes.
- The type of terminal you are using
  - A list of directories in which to locate commands
  - A list of variables effecting look and feel of you terminal.
- You can check your .profile available in your home directory. Open it using vieditor and check all the variable set for your environment

Shell scripts

- When a group of commands have to be executed regularly thry should be stored in a file and the file itself executed as a **shell script or shell program**.

Structure of shell script

```
#! /bin/sh
# script.sh: Sample shell script
echo "Todays date: 'date' "
echo "This month's calendar:"
cal 'date' "+%m  20%y"
 echo "My shell: $SHELL"
```

output:

```
$sh script.sh
Todays date : Mon Nov 7 10:03:42 IST 2016

This month's calendar:
```

November 2016

Su	Mo	Tu	We	Th	Fr	Sa
		1	2	3	4	5
6	7	8	9	10	11	12
13	14	15	16	17	18	19
20	21	22	23	24	25	26
27	28	29	30			

My shell: /bin/sh

read and readonly commands.

read:MAKING SRIPTS INTERACTIVE

- The read statement is the shell internal tool for taking the input from the user ie making scripts interactive.
- It is used with one or more variables. Input is supplied through the standard input is read into these variables.
- Use your vi editor to create the shell script script.sh.
- When you use statement like:
- The script runs three echo commands and shows the use of variable evaluation and command substitution. It also prints the calendar of the current month.

read name

- the script pauses at that point to take input from the keyboard. whatever you enter is stored in the variable name. since this is a form of assignment , no $ is used before the name.
- Note that the # is comment character, that can be placed anywhere in a line; the shell ignores all characters placed on its right.
- However this doesnot apply to the first line, which also begins with a #. This is interpreter line that was mentioned previously.
- It always begins with #! and is followed by pathname of the shell to be used for running the script. However this line specifies the bourne shell.
- To run the script, make it executable first and then invoke the script name

  $chmod a+x script.sh

  $sh script.sh

- Shell scripts are executed in a separate child shell process and this sub shell need not be of the same type as your login shell.By default child and parent shell belongs to the same type, but you can provide a interpreter line in the first line of the script to specify a different shell for your script.

- A single read statement can be used with one or more variables to let you enter multiple arguments.

  read pname flname

- The script asks for a pattern to be entered. Input the string director, which is assigned to the variable pname. Next the script asks for the filename enter the string emp.lst which is assigned to the variable flname.
- grep runs with these two variables as arguments

```
#!/bin/sh
#emp1.sh
#
echo "Enter the pattern to be searched : \c"
read pname
echo " Enter the file to be used : \c"
read flname
echo " Searching for $pname from file $flname"
grep "$pname"  $flname
echo "Selected rows shown above"
```

Output:

$sh emp1.sh

Enter the pattern to be searched:**director**

Enter the file to be used: **emp.lst**

Searching for **director** from file emp.lst

101 sharma|**director**|production|12/03/70|7000

102|barun|**director**|marketing|11/06/67|7800

selected rows shown above

COMMAND LINE ARGUMENTS

- When arguments are specified with a shell script they are assigned to certain special variables- positional parameters.
- $*→ store the complete set of positional parameters as a single string.
- $#--> It is set to the number of arguments specified.
- $0-→ holds the command name itself.
- When arguments are specified in this way the first word (the command itself) is assigned to $0, the second word(the first argument) to $1, the third word(the second argument) to $2.

```
#!/bin/sh
#emp2.sh
#
grep "$1" $2
echo "\n job over"
echo "Program:$0
The number of arguments specified is $#
The arguments are $*"
```

Output:

$ sh emp2.sh director emp.lst

Program: **emp2.sh**
The number of arguments specified is:**2**
The arguments are **director emp.lst**
101| sharma|**director**|production|12/03/70|7000
102|barun|**director**|marketing|11/06/67|7800
job over

SPECIAL PARAMETERS USED BY THE SHELL.

Shell Parameter	Significance
$1, $2...	Positional parameters representing command line arguments
$#	Number of arguments specified in command line
$0	Name of executed command
$*	Complete set of positional parameters as a single string
"$@"	Each quoted string treated as separate argument
$?	Exit status of last command
$$	PID of the current shell
$!	PID of the last background job

Exit and exit status of Command.

- The shells exit command
  exit 0 Used when everything went fine.
  exit 1 Used when something went wrong

Its through the exit command or function that every command returns an exit status to the caller. Further a command is said to return true exit status if it executes successfully and false if its fails.

- THE PARAMETER $? :It stores the exit status of the last command. It has the value 0 if the command succeeds and a non zero value if it fails. This parameter is set by exit's argument If no exit status is specified then $? is set to zero(true).

- Consider two files file1 which exist in current directory and file2 which does not exist

$ ls –l file1; echo $? **/*file1 attributes are listed**
Output :0 /*exit status $?=0, since cmd executed successfully

$ ls –l file2; echo $? **/*error since file2 doesnot exist**
Output: 1 /*exit status $?=1, since cmd execution failed.

THE LOGICAL OPERATORS && and || - CONDITIONAL EXECUTION

- The shell provides two operators that allow conditional execution.the && and||.
- The syntax:

cmd1 && cmd2

cmd1 || cmd2

- Consider a file emp.lst

$cat emp.lst

1066| sharma | **director** |sales |03/09/66 | 7000

1098| Kumar |**director**| production|0/08/67 | 8200

1082|sumith| **manager**|marketing|09/09/73| 7090

- The && delimits two commands ; the command cmd2 is executed only when cmd1 succeeds.

$ grep "director" emp.lst && echo "Pattern found in file"

Output:

1066| sharma | **director** |sales |03/09/66 | 7000

1098| Kumar |**director**| production|0/08/67 | 8200

Pattern found in file

- The || operator plays inverse role. The second command is executed only when the first fails.

$grep " deputy manager" emp.lst || echo "Pattern not found"

Output:

Pattern not found /* cmd1 **-deputy manager** is not found in emp.lst.
Hence cmd1 fails. Therefore cmd2 "**pattern not found**" executes.

$grep "manager" emp.lst || echo "Pattern not found"

Output

1082|sumith| **manager**|marketing|09/09/73| 709 /*Here cmd1 is executed successfully manager is found ,therefore cmd2 will not be executed.

CONDITIONAL STATEMENTS:

The if CONDITIONAL

If command is successful then execute commands else execute commands fi	If command is successful then execute commands fi	If command is successful then execute commands elif command is successful then .. else .. fi

The **if** statement makes two way decision making depending on the fulfillment of a certain condition.

- **If** also requires a **then.**
- It evaluates the success or failure of the command that is specified in its command line. If command succeeds the sequence of the commands following it is executed. If commands fails then the **else** statement is executed
- Every **if** is closed with corresponding with **fi.**

```
#!/bin/sh
a=10
b=20
if [ $a==$b ]
then
echo "a is equal to b"
elif [ $a –gt  $b  ]
then
echo " a is greater than b"
elif [ $a -lt $b  ]
then
echo " a is lesser than b"
else
echo " None of the conditions met"
fi
```

output:
a is lesser than b

The case CONDITIONAL

- The case statement is the second conditional offered by the shell
- The statement matches an expression for more than one alternative and uses a compact construct to permit multiway branching.

The general syntax is

case *expression* in

pattern1) commands1 ;;

pattern2) commands2 ;;

pattern3) commands3 ;;

……..

esac

case first matches expression with pattern1. If the match succeeds, then it executes commands1, which may be one or more commands. If the match fails, then pattern2 is matched and so on….Each command list is terminated with a pair of semicolons and the entire construct is closed with esac

```
#!/bin/sh
#menu.sh
# echo " MENU \n
1.List of files\n 2.Processes of user\n 3.Todays date\n
4.Users of system\n 5.Quit\n
Enter your option: \"

read choice
case "$choice" in
1 ) ls –l ;;
2 ) ps –f ;;
3 ) date  ;;
4 ) who  ;;
5 ) exit ;;
* ) echo "invalid option"
esac
```

To run the program:

$ sh menu.sh

Output:

MENU

1. List of files

2. Processes of user

```
3. Todays date
4.Users of system
5. Quit
Enter your option : 3
Sun Nov 6 18:03:06 IST 2016
```

Matching multiple patterns:

- case can also specify same action for more than one pattern.
- For example the expression y|Y can be used to match y in both upper and lower case letters.

```
echo "Do you wish to continue? : \c"
read answer
case "$answer" in
y|Y )       ;;
n|N ) exit ;;
esac
```

Wild cards: case uses them

- case has a string matching feature that uses wild cards.
- It uses the filename matching meta characters *, ? and the character class but only to match strings but not the files in the current directory.

```
case "$answer" in
[yY][eE] * )       ;;
[nN][oO]  ) exit  ;;
* ) echo "Invalid response"
esac
```

USING test command and its shortcut

USING test and [] to evaluate the expressions.

Test uses the certain operators to evaluate the condition on its right and returns either true or false exit status which is then used by if for making decision

Test works in 3 ways:

Compares two numbers (NUMERIC COMPARISION)

compares two strings or a single one for a null value.(STRING COMPARISION)

checks a file attributes. (FILE TEST)

NUMERIC COMPARISION:

The numeric comparison operators used by test are

Operator	Meaning
-eq	Equal to
-ne	Not equal to
-gt	Greater than
-ge	Greater than or equal to
-lt	Less than
-le	Less than or equal to

Numeric comparison in the shell is confined to integer values only , decimal values are simply

$ x=5 y=7 z=7.2

$ test $x –lt $y ; echo $?

Output: 0

$ test $x –eq $y ; echo $?

Output : 1

$ test $z –gt $y ; echo $?

Output: 1

test can be used to compare strings with yet another set of operators.

STRING COMPARISION

Test	True if
s1=s2	String s1=s2
s1!=s2	String s1 is not equal to s2
-n stg	String stg is not a null string
-z stg	String stg is null string
Stg	String stg is assigned and not null
s1==s2	String s1= s2(Korn and bash only)

Example:

```
#!/bin/sh

a=”abc”
b=”efg”
if [ $a = $b ]
then
echo “a is equal to b”
else
echo “a is not equal to b”
fi
```

output:
a is not equal to b

FILE TESTS

test can be used to test the various file attributes like its type(file, directory or symbolic link) or its permissions(read,write,execute)

Test	True if File
-f file	File exists and is regular file
-r file	File exist and is readable
-w file	File exists and is writable
-x file	File exists and is execuatble
-d file	File exists and is a directory
-s file	File exists and has a size greater than zero
-e file	File exists (Korn and bash only)
-L file	File exists and is symbolic link
f1 -nt f2	f1 is newer than f2(Korn and bash only)
f1 -ot f2	f1 is older than f2(Korn and bash only)
f1 -ef f2	f1 is linked to f2(Korn and bash only)

$ls –l emp.lst

-rw-rw-rw- 1 kumar group 870 Sep 8 1:5:52 emp.lst // **-f** (emp.lst file exist and its regular file)
0 // Yes
$ [-f emp.lst] ; echo $? (emp.lst file is executable or not)
1 //No
$ [-x emp.lst] ; echo $? // **-x**

---

while Looping

- The while statement repeatedly performs a set of instructions until the control command return a true exit status
- The general syntax is

  while *condition* is true

  do

  commands

  done

- The commands enclosed by do and done are executed repeatedly as long as condition remains true.

Using while to wait for a file

- There are situations when a program needs to read a file that is created by another program, but it has to wait until the file is created.
- The script, monitfile.sh periodically monitors the disk for the existence of the file. And then executes the program once the file has been located.
- It makes use of the external sleep command that makes the script pauses for the duration in seconds as specified in its arguments
- The loop executes repeatedly as long as the file invoice.lst cannot be read.
- If the file becomes readable the loop is terminated and the program alloc.pl is executed.
- We use the sleep command to check every 60 seconds for the existence of the file.

Setting up an infinite loop

Suppose you as system administrator want to see the free space available on your disk every five minutes

```
while true
do
df –t
sleep 300



done &
```

df reports free space on disk . sleep command is used to hek for every 300 seconds(5 minutes).

& after done runs loop in background

---

for : LOOPING WITH A LIST

The shells for loop differs in structure from the ones used in other programming languages.

There is no three part structure.

```
for variables in list
do
commands
done
```

The loop body also uses the keyword do and done. But the additional parameters here are variable and list. Each whitespace separated word in list is assigned to variable and commands are executed until list is executed

Ex:

```
$for file in chap20 chap21 chap22
 do
 cp $file  {$file}.bak
 echo $file copied to $file.bak
 done
```

Output:
chap20 copied to chap20.bak
chap21 copied to chap21.bak
chap22 copied to chap22.bak

set and shift: MANIPULATING THE POSITIONAL PARAMETERS

- set assigns its argument to positional parameters $1,$2 and so on.
  $set 989 878 779
  $_
  This assigns the value 989 to the positional parameter $1, 878 to the positional parameter $2 and 779 to $3

Ex:
$echo "\$1 is $1, \$2 is $2, \$3 is $3"
Output: $1 is 989, $2 is 878, $3 is 779
$echo "The $# arguments are $*"
Output: The 3 arguments are 989 878 779

Shift : Shifting Arguments left
Shift transfers the contents of a positional parameter to its immediate lower numbered one.
$ set `date'
$echo "$*
Output: Wed Nov 9 09:04:30 IST 2016
$shift
$ echo $1 $2 $3 $4 $5
Output: Nov 9 09:04:30 IST 2016

$shift 2
$echo $1 $2 $3

Output: 09:04:30 IST 2016

The HERE DOCUMENT (<<)

- The shell uses << symbols to read data from the same file containing the script.
- This is referred to as here document , signifying that the data is here rather than in a separate file

- If the message is short you can have both the command and message in the same script.

```
mail sharma << MARK
Your program for printing the invoices has been executed
on `date`. The updated file is $flname
MARK
```

- The here document symbol(<<) followed by three lines of data and a delimiter (the string MARK)
- The shell treats every line following the command and delimited by MARK as input to the command.
- Sharma at the other end will see the three lines of message text with the date inserted by command substitution and the evaluated filename.

trap: INTERRUPTING A PROGRAM

- By default shell scripts terminate whenever the interrupt key is pressed. It may leave a lot of temporary files on disk
- The trap statement lets you do things you want in case the script receives a signal. The statement is normally placed at the beginning of a shell script and uses two lists

trap 'command _list' signal_list

- When a script is sent any of the signals in signal_list, trap executes the commands in command_list
- The signal_list can contain the integer values or names of one or more signals.

trap **'rm $$* ; eho "Program Interrupted" ; exit '** HUP INT TERM

trap 'cmd_list' sig_list

- trap is a signal handler.
- Here it first removes all the files expanded from $$*, echoes a message and finally terminates the script when the signals SIGHUP(1), SIGINT(2), SIGTERM(15) are sent to the shell process running the script.

Chapter 1: UNIX File APIs

1. General File APIs
2. File and Record Locking
3. Directory File APIs
4. Device File APIs
5. FIFO File APIs
6. Symbolic Link File APIs

1. General file API's

Files in a UNIX and POSIX system may be any one of the following types:

- Regular file
- Directory file
- FIFO file
- Block device file
- Character device file
- Symbolic link file

There are special API's to create these types of files. There is a set of Generic API's that can be used to manipulate and create more than one type of files. These API's are:

open

- This is used to establish a connection between a process and a file i.e. it is used to open an existing file for data transfer function or else it may be also be used to create a new file.
- The returned value of the open system call is the file descriptor (row number of the file table), which contains the inode information.
- The prototype of open function is

```
#include<sys/types.h>
#include<sys/fcntl.h>
int open(const char *pathname, int accessmode, mode_t  permission);
```

- If successful, open returns a nonnegative integer representing the open file descriptor.
- If unsuccessful, open returns –1.
- The first argument is the name of the file to be created or opened. This may be an absolute pathname or relative pathname.
- If the given pathname is symbolic link, the open function will resolve the symbolic link reference to a non symbolic link file to which it refers.
- The second argument is access modes, which is an integer value that specifies how actually the file should be accessed by the calling process.
- Generally the access modes are specified in <fcntl.h>. Various access modes are:

O_RDONLY - open for reading file only **O_WRONLY** - open for writing file only **O_RDWR** - opens for reading and writing file.

There are other access modes, which are termed as access modifier flags, and one or more of the following can be specified by bitwise-ORing them with one of the above access mode flags to alter the access mechanism of the file.

O_APPEND	- Append data to the end of file.
O_CREAT	- Create the file if it doesn't exist
O_EXCL	- Generate an error if O_CREAT is also specified and the file already exists.
O_TRUNC	- If file exists discard the file content and set the file size to zero bytes.
O_NONBLOCK	- Specify subsequent read or write on the file should be non-blocking.
O_NOCTTY	- Specify not to use terminal device file as the calling process control terminal.

To illustrate the use of the above flags, the following example statement opens a file called /usr/divya/usp for read and write in append mode:

int fd=open("/usr/divya/usp",O_RDWR | O_APPEND,0);

- If the file is opened in read only, then no other modifier flags can be used.
- If a file is opened in write only or read write, then we are allowed to use any modifier flags along with them.
- The third argument is used only when a new file is being created. The symbolic names for file permission are given in the table in the previous page.

symbol	meaning
S_IRUSR	read by owner
S_IWUSR	write by owner
S_IXUSR	execute by owner
S_IRWXU	read, write, execute by owner
S_IRGRP	read by group
S_IWGRP	write by group
S_IXGRP	execute by group
S_IRWXG	read, write, execute by group
S_IROTH	read by others
S_IWOTH	write by others
S_IXOTH	execute by others
S_IRWXO	read, write, execute by others

creat

- This system call is used to create new regular files.
- The prototype of creat is

```
#include <sys/types.h>
#include<unistd.h>
int creat(const char *pathname, mode_t  mode);
```

- Returns: file descriptor opened for write-only if OK, -1 on error.
- The first argument pathname specifies name of the file to be created.
- The second argument mode_t, specifies permission of a file to be accessed by owner group and others.
- The creat function can be implemented using open function as:

#define creat(path_name, mode)
open (pathname, O_WRONLY | O_CREAT | O_TRUNC, mode);

read

- The read function fetches a fixed size of block of data from a file referenced by a given file descriptor.
- The prototype of read function is:

```
#include<sys/types.h>
#include<unistd.h>
size_t read(int fdesc, void *buf, size_t nbyte);
```

- If successful, read returns the number of bytes actually read.
- If unsuccessful, read returns –1.
- The first argument is an integer, fdesc that refers to an opened file.
- The second argument, buf is the address of a buffer holding any data read.
- The third argument specifies how many bytes of data are to be read from the file.
- The size_t data type is defined in the <sys/types.h> header and should be the same as unsigned int.
- There are several cases in which the number of bytes actually read is less than the amount requested:
  - When reading from a regular file, if the end of file is reached before the requested number of bytes has
  - been read. For example, if 30 bytes remain until the end of file and we try to read 100 bytes, read returns
  - 30. The next time we call read, it will return 0 (end of file).
  - When reading from a terminal device. Normally, up to one line is read at a time.
  - When reading from a network. Buffering within the network may cause less than the requested amount to be returned.
  - When reading from a pipe or FIFO. If the pipe contains fewer bytes than requested, read will return only what is available.

write

- The write system call is used to write data into a file.
- The write function puts data to a file in the form of fixed block size referred by a given file descriptor.
- The prototype of write function is:

```
#include<sys/types.h>
#include<unistd.h>
ssize_t write(int fdesc, const void *buf, size_t size);
```

- If successful, write returns the number of bytes actually written.
- If unsuccessful, write returns –1.
- The first argument, fdesc is an integer that refers to an opened file.
- The second argument, buf is the address of a buffer that contains data to be written.
- The third argument, size specifies how many bytes of data are in the buf argument.
- The return value is usually equal to the number of bytes of data successfully written to a file. (*size* value)

close

- The close system call is used to terminate the connection to a file from a process.
- The prototype of the close is

```
#include<unistd.h> int
close(int fdesc);
```

- If successful, close returns 0.
- If unsuccessful, close returns –1.
- The argument fdesc refers to an opened file.
- Close function frees the unused file descriptors so that they can be reused to reference other files. This is important because a process may open up to OPEN_MAX files at any time and the close function allows a process to reuse file descriptors to access more than OPEN_MAX files in the course of its execution.
- The close function de-allocates system resources like file table entry and memory buffer allocated to hold the read/write.

fcntl

- The fcntl function helps a user to query or set flags and the close-on-exec flag of any file descriptor.
- The prototype of fcntl is

```
#include<fcntl.h>
int fcntl(int fdesc, int cmd, …);
```

- The first argument is the file descriptor.
- The second argument cmd specifies what operation has to be performed.
- The third argument is dependent on the actual cmd value.
- The possible cmd values are defined in <fcntl.h> header.

cmd value	Use
F_GETFL	Returns the access control flags of a file descriptor fdesc
F_SETFL	Sets or clears access control flags that are specified in the third argument to fcntl. The allowed access control flags are O_APPEND & O_NONBLOCK
F_GETFD	Returns the close-on-exec flag of a file referenced by fdesc. If a return value is zero, the flag is off; otherwise on.
F_SETFD	Sets or clears the close-on-exec flag of a fdesc. The third argument to fcntl is an integer value, which is 0 to clear the flag, or 1 to set the flag
F_DUPFD	Duplicates file descriptor fdesc with another file descriptor. The third argument to fcntl is an integer value which specifies that the duplicated file descriptor must be greater than or equal to that value. The return value of fcntl is the duplicated file descriptor

- The fcntl function is useful in changing the access control flag of a file descriptor.
- For example: after a file is opened for blocking read-write access and the process needs to change the access to non-blocking and in write-append mode, it can call:

```
int cur_flags=fcntl(fdesc,F_GETFL);
int rc=fcntl(fdesc,F_SETFL,cur_flag | O_APPEND | O_NONBLOCK);
```

The following example reports the close-on-exec flag of fdesc, sets it to on afterwards:

```
cout<<fdesc<<"close-on-
exec"<<fcntl(fdesc,F_GETFD)<<endl;
(void)fcntl(fdesc,F_SETFD,1); //turn on close-on-exec flag
```

The following statements change the standard input og a process to a file called FOO:

```
int fdesc=open("FOO",O_RDONLY);          //open FOO for read
```

```
close(0);                                   //close standard  input
if(fcntl(fdesc,F_DUPFD,0)==-1)
perror("fcntl");                            //stdin from FOO  now
char buf[256];
int  rc=read(0,buf,256);                    //read data from  FOO
```

The dup and dup2 functions in UNIX perform the same file duplication function as fcntl. They can be implemented using fcntl as:

#define	**dup(fdesc)**	**fcntl(fdesc, F_DUPFD,0)**
#define	**dup2(fdesc1,fd2)**	**close(fd2),fcntl(fdesc,F_DUPFD,fd 2)**

lseek

- The lseek function is also used to change the file offset to a different value.
- Thus lseek allows a process to perform random access of data on any opened file.
- The prototype of lseek is

#include <sys/types.h> #include <unistd.h>

off_t ***lseek***(**int** fdesc, **off_t** pos, **int** whence);
On success it returns new file offset, and –1 on error.

- The first argument fdesc, is an integer file descriptor that refer to an opened file.
- The second argument pos, specifies a byte offset to be added to a reference location in deriving the new file offset value.
- The third argument whence, is the reference location.

Whence value	Reference location
SEEK_CUR	Current file pointer address
SEEK_SET	The beginning of a file
SEEK_END	The end of a file

- They are defined in the <unistd.h> header.
- If an lseek call will result in a new file offset that is beyond the current end-of-file, two outcomes possible are: o If a file is opened for read-only, lseek will fail.

- If a file is opened for write access, lseek will succeed.
- The data between the end-of-file and the new file offset address will be initialised with NULL characters.

link

- The link function creates a new link for the existing file.
- The prototype of the link function is

```
#include <unistd.h>
int link(const char *cur_link, const char *new_link);
```

- If successful, the link function returns 0.
- If unsuccessful, link returns –1.
- The first argument cur_link, is the pathname of existing file.
- The second argument new_link is a new pathname to be assigned to the same file.
- If this call succeeds, the hard link count will be increased by 1.
- The UNIX ln command is implemented using the link API.

```
#include<iostream.h>
#include<stdio.h>
#include<unistd.h>

int main(int argc, char* argv)
{
    if(argc!=3)
    {
        cerr<<"usage:"<<argv[0]<<"<src_file><dest_file>\n"; return 0;
    }
    if(link(argv[1],argv[2])==-1)
    {
        perror("link");
        return 1;
    }
    return 0;
}
```

unlink

- The unlink function deletes a link of an existing file.
- This function decreases the hard link count attributes of the named file, and removes the file name

entry of the link from directory file.

- A file is removed from the file system when its hard link count is zero and no process has any file descriptor referencing that file.
- The prototype of unlink is

```
#include  <unistd.h>
int unlink(const char * cur_link);
```

- If successful, the unlink function returns 0.
- If unsuccessful, unlink returns –1.
- The argument cur_link is a path name that references an existing file.
- ANSI C defines the rename function which does the similar unlink operation.
- The prototype of the rename function is:

```
#include<stdio.h>
int rename(const char * old_path_name,const char *  new_path_name);
```

- The UNIX mv command can be implemented using the link and unlink APIs as shown:

```
#include <iostream.h>
#include
<unistd.h>
#include<string.h
>
int main ( int argc, char *argv[ ])
{
        if (argc != 3 || strcmp(argv[1],argcv[2]))
                cerr<<"usage:"<<argv[0]<<""<old_link><new_link>\
                n";
        else if(link(argv[1],argv[2]) == 0)
                return unlink(argv[1]);
        return 1;
}
```

stat, fstat

- The stat and fstat function retrieves the file attributes of a given file.
- The only difference between stat and fstat is that the first argument of a stat is a file pathname, where as the first argument of fstat is file descriptor.
- The prototypes of these functions are

```
#include<sys/stat.h>
#include<unistd.h>

int stat(const char *pathname, struct stat *statv);
int fstat(const int fdesc, struct stat *statv);
```

- The second argument to stat and fstat is the address of a struct stat-typed variable which is defined in the <sys/stat.h> header.
- Its declaration is as follows:

```
struct stat
{
dev_t      st_dev;     /* file system ID */
ino_t      st_ino;     /* file inode number */
mode_t     st_mode;    /* contains file type and permission
                          */
nlink_t    st_nlink;   /* hard link count */
uid_t      st_uid;     /* file user ID */
gid_t      st_gid;     /* file group ID */
dev_t      st_rdev;    /*contains major and minor
                          device#*/
off_t      st_size;    /* file size in bytes */
time_t     st_atime;   /* last access time */
time_t     st_mtime;   /* last modification time */
time_t     st_ctime;   /* last status change time */
};
```

- The return value of both functions is
  - 0 if they succeed
  - 1 if they fail
  - *errno* contains an error status code

- The lstat function prototype is the same as that of stat:

```
int lstat(const char * path_name, struct stat*  statv);
```

- We can determine the file type with the macros as shown.

macro	Type of file
S_ISREG()	regular file
S_ISDIR()	directory file
S_ISCHR()	character special file
S_ISBLK()	block special file
S_ISFIFO()	pipe or FIFO
S_ISLNK()	symbolic link
S_ISSOCK()	socket

access

- The access system call checks the existence and access permission of user to a named file.
- The prototype of access function is:

```
#include<unistd.h>
int access(const char *path_name, int  flag);
```

- On success access returns 0, on failure it returns –1.
- The first argument is the pathname of a file.
- The second argument flag, contains one or more of the following bit flag .

Bit flag	Uses
F_OK	Checks whether a named file exist
R_OK	Test for read permission
W_OK	Test for write permission
X_OK	Test for execute permission

- The flag argument value to an access call is composed by bitwise-ORing one or more of the above bit flags as shown:

int rc=access("/usr/divya/usp.txt",R_OK | W_OK);

- example to check whether a file exists:

```
if(access("/usr/divya/usp.txt",
F_OK)==-1) printf("file does not
exists");
else
printf("file exists");
```

chmod, fchmod

- The chmod and fchmod functions change file access permissions for owner, group & others as well as the set_UID, set_GID and sticky flags.
- A process must have the effective UID of either the super-user/owner of the file.

```
#include<sys/types.h>
#include<sys/stat.h>
#include<unistd.h>
int chmod(const char *pathname, mode_t flag); int fchmod(int fdesc, mode_t flag);
```

- The prototypes of these functions are

- The pathname argument of chmod is the path name of a file whereas the fdesc argument of fchmod is the file descriptor of a file.
- The chmod function operates on the specified file, whereas the fchmod function operates on a file that has already been opened.
- To change the permission bits of a file, the effective user ID of the process must be equal to the owner ID of the file, or the process must have super-user permissions. The mode is specified as the bitwise OR of the constants shown below.

Mode	Description
S_ISUID	set-user-ID on execution
S_ISGID	set-group-ID on execution
S_ISVTX	saved-text (sticky bit)
S_IRWXU	read, write, and execute by user (owner)

S_IRUSR	read by user (owner)
S_IWUSR	write by user (owner)
S_IXUSR	execute by user (owner)
S_IRWXG	read, write, and execute by group
S_IRGRP	read by group
S_IWGRP	write by group
S_IXGRP	execute by group
S_IRWXO	read, write, and execute by other (world)
S_IROTH	read by other (world)
S_IWOTH	write by other (world)
S_IXOTH	execute by other (world)

chown, fchown, lchown

- The chown functions changes the user ID and group ID of files.

The prototypes of these functions are

```
#include<unistd.h>
#include<sys/types.h>

int chown(const char *path_name, uid_t uid, gid_t gid); int fchown(int fdesc, uid_t uid, gid_t gid);
int lchown(const char *path_name, uid_t uid, gid_t  gid);
```

- The path_name argument is the path name of a file.
- The uid argument specifies the new user ID to be assigned to the file.
- The gid argument specifies the new group ID to be assigned to the file.

/* Program to illustrate chown function */

```
#include<iostream.h>
#include<sys/types.h>
#include<sys/stat.h>
#include<unistd.h>
#include<pwd.h>
int main(int argc, char *argv[ ])
{
        if(argc>3)
        {
                cerr<<"usage:"<<argv[0]<<"<usr_name><file>    \n";
                return 1;
```

```
        }

        struct passwd *pwd = getpwuid(argv[1]) ;
        uid_t           UID = pwd ? pwd -> pw_uid : -1 ;
        struct          stat    statv;


        if (UID == (uid_t)-1)
                cerr <<"Invalid user name"; else for (int i
        = 2; i < argc ; i++)
                if (stat(argv[i], &statv)==0)
                {
                        if (chown(argv[i], UID,statv.st_gid)) perror
                                ("chown");
                        else
                                perror ("stat");
                }
         return 0;
}
```

- The above program takes at least two command line arguments:
  - The first one is the user name to be assigned to files
  - The second and any subsequent arguments are file path names.
- The program first converts a given user name to a user ID via *getpwuid* function. If that succeeds, the program processes each named file as follows: it calls *stat* to get the file group ID, then it calls *chown* to change the file user ID. If either the stat or chown fails, error is displayed.

utime Function

- The utime function modifies the access time and the modification time stamps of a file.
- The prototype of utime function is

```
#include<sys/types.h>
#include<unistd.h>
#include<utime.h>

int utime(const char *path_name, struct utimbuf  *times);
```

- On success it returns 0, on failure it returns –1.
- The path_name argument specifies the path name of a file.
- The times argument specifies the new access time and modification time for the file.
- The struct utimbuf is defined in the <utime.h> header as:

```
struct utimbuf
{
        time_t          actime;         /* access time */
        time_t          modtime;        /* modification time */
}
```

- The time_t datatype is an unsigned long and its data is the number of the seconds elapsed since the birthday of UNIX : 12 AM , Jan 1 of 1970.
- If the times (variable) is specified as NULL, the function will set the named file access and modification time to the current time.
- If the times (variable) is an address of the variable of the type struct utimbuf, the function will set the file access time and modification time to the value specified by the variable.

2. File and Record Locking

- Multiple processes performs read and write operation on the same file concurrently.
- This provides a means for data sharing among processes, but it also renders difficulty for any process in determining when the other process can override data in a file.
- So, in order to overcome this drawback UNIX and POSIX standard support file locking mechanism.
- File locking is applicable for regular files.
- Only a process can impose a write lock or read lock on either a portion of a file or on the entire file.
- The differences between the read lock and the write lock is that when write lock is set, it prevents the other process from setting any over-lapping read or write lock on the locked file.
- Similarly when a read lock is set, it prevents other processes from setting any overlapping write locks on the locked region.
- The intension of the write lock is to prevent other processes from both reading and writing the locked region while the process that sets the lock is modifying the region, so write lock is termed as "**Exclusive lock**".
- The use of read lock is to prevent other processes from writing to the locked region while the process that sets the lock is reading data from the region.
- Other processes are allowed to lock and read data from the locked regions. Hence a read lock is also called as "**shared lock** ".
- File lock may be **mandatory** if they are enforced by an operating system kernel.
- If a mandatory exclusive lock is set on a file, no process can use the read or write system calls to access the data on the locked region.
- These mechanisms can be used to synchronize reading and writing of shared files by multiple processes.
- If a process locks up a file, other processes that attempt to write to the locked regions are blocked until the former process releases its lock.
- Problem with mandatory lock is – if a runaway process sets a mandatory exclusive lock on a file and never unlocks it, then, no other process can access the locked region of the file until the runway process is killed or the system has to be rebooted.
- If locks are not mandatory, then it has to be **advisory** lock.
- A kernel at the system call level does not enforce advisory locks.

- This means that even though a lock may be set on a file, no other processes can still use the read and write functions to access the file.
- To make use of advisory locks, process that manipulate the same file must co-operate such that they follow the given below procedure for every read or write operation to the file.
    1. Try to set a lock at the region to be accesses. If this fails, a process can either wait for the lock request to become successful.
    2. After a lock is acquired successfully, read or write the locked region.
    3. Release the lock.
- If a process sets a read lock on a file, for example from address 0 to 256, then sets a write lock on the file from address 0 to 512, the process will own only one write lock on the file from 0 to 512, the previous read lock from 0 to 256 is now covered by the write lock and the process does not own two locks on the region from 0 to 256. This process is called "**Lock Promotion**".
- Furthermore, if a process now unblocks the file from 128 to 480, it will own two write locks on the file: one from 0 to 127 and the other from 481 to 512. This process is called "**Lock Splitting**".
- UNIX systems provide fcntl function to support file locking. By using fcntl it is possible to impose read or write locks on either a region or an entire file.
- The prototype of fcntl is

```
#include<fcntl.h>
int fcntl(int fdesc, int cmd_flag, ..... );
```

- The first argument specifies the file descriptor.
- The second argument cmd_flag specifies what operation has to be performed.
- If fcntl is used for file locking then it can values as

F_SETLK	sets a file lock, do not block if this cannot succeed immediately.
F_SETLKW	sets a file lock and blocks the process until the lock is acquired.
F_GETLK	queries as to which process locked a specified region of file.

- For file locking purpose, the third argument to fctnl is an address of a *struct flock* type variable.
- This variable specifies a region of a file where lock is to be set, unset or queried.

```
struct flock
{
short    l_type;     /* what lock to be set or to unlock file */
short    l_whence;     /* Reference address for the next field */
off_t    l_start ;     /*offset from the l_whence reference
                       addr*/
off_t    l_len ;     /*how many bytes in the locked region */
pid_t    l_pid ;      /*pid of a process which has locked the
                      file */
};
```

- The l_type field specifies the lock type to be set or unset.
- The possible values, which are defined in the <fcntl.h> header, and their uses are:

l_type value	Use
F_RDLCK	Set a read lock on a specified region
F_WRLCK	Set a write lock on a specified region
F_UNLCK	Unlock a specified region

- The l_whence, l_start, and l_len define a region of a file to be locked or unlocked.
- The possible values of l_whence and their uses are:

l_whence value	Use
SEEK_CUR	The l_start value is added to current file pointer address
SEEK_SET	The l_start value is added to byte 0 of the file
SEEK_END	The l_start value is added to the end of the file

- A lock set by the fcntl API is an advisory lock but we can also use fcntl for mandatory locking purpose with the following attributes set before using fcntl
    1. Turn on the set-GID flag of the file.
    2. Turn off the group execute right permission of the file.

- In the given example program we have performed a read lock on a file "divya" from the 10th byte to 25th byte.

Example Program

```
#include <unistd.h>
#include<fcntl.h> int main ()
{
        int fd;
        struct flock lock;
        fd=open("divya",O_RDONLY);
        lock.l_type=F_RDLCK;
        lock.l_whence=0; lock.l_start=10;
        lock.l_len=15;
        fcntl(fd,F_SETLK,&lock);
}
```

3. Directory File API's

- A Directory file is a record-oriented file, where each record stores a file name and the inode number of a file that resides in that directory.
- Directories are created with the mkdir API and deleted with the rmdir API.
- The prototype of mkdir is

```
#include<sys/stat.h>
#include<unistd.h>

int mkdir(const char *path_name, mode_t  mode);
```

- The first argument is the path name of a directory file to be created.
- The second argument mode, specifies the access permission for the owner, groups and others to be assigned to the file. This function creates a new empty directory.
- The entries for "." and ".." are automatically created. The specified file access permission, mode, are modified by the file mode creation mask of the process.
- To allow a process to scan directories in a file system independent manner, a directory record is defined as ***struct dirent*** in the <dirent.h> header for UNIX.
- Some of the functions that are defined for directory file operations in the above header are

```
#include<sys/types.h>
#if defined (BSD)&&!_POSIX_SOURCE
    #include<sys/dir.h>
    typedef struct direct Dirent;
  #else
    #include<dirent.h>
    typedef struct direct Dirent;
  #endif
```

DIR ***opendir**(**const char ***path_name);

Dirent ***readdir**(**DIR ***dir_fdesc);

int ***closedir**(**DIR ***dir_fdesc);

void ***rewinddir**(**DIR ***dir_fdsec);

The uses of these functions are

Function	Use
opendir	Opens a directory file for read-only. Returns a file handle dir * for future reference of the file.
readdir	Reads a record from a directory file referenced by dir-fdesc and returns that record information.
rewinddir	Resets the file pointer to the beginning of the directory file referenced by dir-fdesc. The next call to readdir will read the first record from the file.
closedir	closes a directory file referenced by dir-fdesc.

- An empty directory is deleted with the rmdir API.
- The prototype of rmdir is

```
#include<unistd.h>
int rmdir (const char * path_name);
```

- UNIX systems have defined additional functions for random access of directory file records.

Function	Use
telldir	Returns the file pointer of a given dir_fdesc
seekdir	Changes the file pointer of a given dir_fdesc to a specified address

The following list_dir.C program illustrates the uses of the mkdir, opendir, readdir, closedir and rmdir APIs:

```
#include<iostream.
h>
#include<stdio.h>
#include<sys/types.h
>
#include<unistd.h>
#include<string.h>
#include<sys/stat.h>
#if defined(BSD) && !_POSIX_SOURCE
        #include<sys/dir.h>
        typedef struct dirent Dirent;

#else
#endif
#include<dirent.h>
typedef struct dirent Dir
int main(int agc, char* argv[])
{
Dirent* dp; DIR*
dir_fdesc; while(--argc>0)
{
if(!(dir_fdesc=opendir(*++argv)))
{
if(mkdir(*argv,S_IRWXU | S_IRWXG |
S_IRWXO)==-1) perror("opendir");
continue;
}
for(int i=0;i<2;i++)
for(int cnt=0;dp=readdir(dir_fdesc);)
{
if(i) cout<<dp->d_name<<endl;
if(strcmp(dp->d_name,".") && strcmp(dp->d_name,".")) cnt++;
}
if(!cnt)
{
```

```
rmdir(*argv); break;
}
rewinddir(dir_fdesc);
}
closedir(dir_fdesc);
}
}
```

4. Device file APIs

- Device files are used to interface physical device with application programs.
- A process with superuser privileges to create a device file must call the mknod API.
- The user ID and group ID attributes of a device file are assigned in the same manner as for regular files.
- When a process reads or writes to a device file, the kernel uses the major and minor device numbers of a file to select a device driver function to carry out the actual data transfer.
- Device file support is implementation dependent. UNIX System defines the mknod API to create device files.
- The prototype of mknod is

```
#include<sys/stat.h>
#include<unistd.h>
int mknod(const char* path_name, mode_t mode, int device_id);
```

- The first argument pathname is the pathname of a device file to be created.
- The second argument mode specifies the access permission, for the owner, group and others, also S_IFCHR or S_IBLK flag to be assigned to the file.
- The third argument device_id contains the major and minor device number.
- **Example**

  mknod("SCSI5",S_IFBLK | S_IRWXU | S_IRWXG | S_IRWXO,(15<<8) | 3);

- The above function creates a block device file "divya", to which all the three i.e. read, write and execute permission is granted for user, group and others with major number as 8 and minor number 3.
- On success mknod API returns 0 , else it returns -1

The following test_mknod.C program illustrates the use of the mknod, open, read, write and close APIs on a block device file.

```
#include<iostream.
h>
```

```
#include<stdio.h>
#include<stdlib.h>
#include<sys/types.h
>
#include<unistd.h>
#include<fcntl.h>
#include<sys/stat.h
>

int main(int argc, char* argv[])
{

        if(argc!=4)
        {
                cout<<"usage:"<<argv[0]<<"<file><major_no><minor_no>"; return 0;
        }
        int major=atoi(argv[2],minor=atoi(argv[3]);
        (void) mknod(argv[1], S_IFCHR | S_IRWXU | S_IRWXG | S_IRWXO, (major<<8) | minor);

        int rc=1,fd=open(argv[1],O_RDW | O_NONBLOCK | O_NOCTTY); char buf[256];
        while(rc && fd!=-1) if((rc=read(fd,buf,sizeof(buf)))<0)
                perror("read");
        else if(rc) cout<<buf<<endl;
        close(fd);
}
```

5. FIFO file API's

- FIFO files are sometimes called named pipes.
- Pipes can be used only between related processes when a common ancestor has created the pipe.
- Creating a FIFO is similar to creating a file.
- Indeed the pathname for a FIFO exists in the file system.
- The prototype of mkfifo is

```
#include<sys/types.h>
#include<sys/stat.h>
#include<unistd.h>

int mkfifo(const char *path_name, mode_t  mode);
```

- The first argument pathname is the pathname(filename) of a FIFO file to be created.
- The second argument mode specifies the access permission for user, group and others and as well as the S_IFIFO flag to indicate that it is a FIFO file.
- On success it returns 0 and on failure it returns –1.
- **Example**

mkfifo("FIFO5",S_IFIFO | S_IRWXU | S_IRGRP | S_ROTH);

- The above statement creates a FIFO file "divya" with read-write-execute permission for user and only read permission for group and others.
- Once we have created a FIFO using mkfifo, we open it using open.
- Indeed, the normal file I/O functions (read, write, unlink etc) all work with FIFOs.
- When a process opens a FIFO file for reading, the kernel will block the process until there is another process that opens the same file for writing.
- Similarly whenever a process opens a FIFO file write, the kernel will block the process until another process opens the same FIFO for reading.
- This provides a means for synchronization in order to undergo inter-process communication.
- If a particular process tries to write something to a FIFO file that is full, then that process will be blocked until another process has read data from the FIFO to make space for the process to write.
- Similarly, if a process attempts to read data from an empty FIFO, the process will be blocked until another process writes data to the FIFO.
- From any of the above condition if the process doesn't want to get blocked then we should specify O_NONBLOCK in the open call to the FIFO file.
- If the data is not ready for read/write then open returns –1 instead of process getting blocked.
- If a process writes to a FIFO file that has no other process attached to it for read, the kernel will send SIGPIPE signal to the process to notify that it is an illegal operation.
- Another method to create FIFO files (not exactly) for inter-process communication is to use the pipe system call.
- The prototype of pipe is

```
#include<unistd.h>
int pipe(int fds[2]);
```

- Returns 0 on success and –1 on failure.
- If the pipe call executes successfully, the process can read from fd[0] and write to fd[1]. A single process with a pipe is not very useful. Usually a parent process uses pipes to communicate with its children.

The following test_fifo.C example illustrates the use of mkfifo, open, read, write and close APIs for a FIFO file:

```
#include<iostream.h>
#include<stdio.h>
#include<sys/types.h
>
#include<unistd.h>
#include<fcntl.h>
#include<sys/stat.h>
#include<string.h>
#include<errno.h>
int main(int argc,char* argv[])
{
        if(argc!=2 && argc!=3)
        {
                cout<<"usage:"<<argv[0]<<"<file> [<arg>]"; return 0;
        }
        int fd;
        char buf[256];
        (void) mkfifo(argv[1], S_IFIFO | S_IRWXU | S_IRWXG | S_IRWXO ); if(argc==2)
        {
                fd=open(argv[1],O_RDONLY | O_NONBLOCK);
                while(read(fd,buf,sizeof(buf))==-1 && errno==EAGAIN)
                        sleep(1); while(read(fd,buf,sizeof(buf))>0)
                        cout<<buf<<endl;
        }
        else
        {
                fd=open(argv[1],O_WRONLY);
                write(fd,argv[2],strlen(argv[2]));
        }
        close(fd
        );
}
```

6. Symbolic Link File API's

- A symbolic link is an indirect pointer to a file, unlike the hard links which pointed directly to the inode of the file.
- Symbolic links are developed to get around the limitations of hard links.
- Symbolic links can link files across file systems.
  - Symbolic links can link directory files
  - Symbolic links always reference the latest version of the files to which they link
  - There are no file system limitations on a symbolic link and what it points to and anyone can create a symbolic link to a directory.
  - Symbolic links are typically used to move a file or an entire directory hierarchy to some other location on a system.
  - A symbolic link is created with the symlink.
  - The prototype is

```
#include<unistd.h>
#include<sys/types.h>
#include<sys/stat.h>

int symlink(const char *org_link, const char *sym_link); int readlink(const char*
sym_link,char* buf,int size); int lstat(const char * sym_link, struct stat* statv);
```

- The org_link and sym_link arguments to a sym_link call specify the original file path name and the symbolic link path name to be created.

/* Program to illustrate symlink function */

```
#include<unistd.h>
#include<sys/types.h
> #include<string.h>

int main(int argc, char *argv[])
{
```

```
        char *buf [256], tname [256]; if (argc ==4)
        return symlink(argv[2], argv[3]); /* create a symbolic link
        */ else return link(argv[1], argv[2]); /* creates a hard link
        */
}
```

Chapter 2: The Environment of a UNIX Process

1. Introduction
2. main function,
3. Process Termination
4. Command-Line Arguments
5. Environment List
6. Memory Layout of a C Program
7. Shared Libraries
8. Memory Allocation
9. Environment Variables
10. setjmp and longjmp Functions
11. getrlimit, setrlimit Functions
12. UNIX Kernel Support for Processes.

1. INTRODUCTION

A Process is a program under execution in a UNIX or POSIX system.

2. main FUNCTION

- A C program starts execution with a function called main.
- The prototype for the main function is

  int main(int argc, char *argv[]);

  where argc is the number of command-line arguments,
  and argv is an array of pointers to the arguments.
- When a C program is executed by the kernel by one of the exec functions, a special start-up routine is called before the main function is called.
- The executable program file specifies this routine as the starting address for the program; this is set up by the link editor when it is invoked by the C compiler. This start-up routine takes values from

the kernel, the command-line arguments and the environment and sets things up so that the main function is called.

3. PROCESS TERMINATION

- There are eight ways for a process to terminate. Normal termination occurs in five ways:
  1. Return from main
  2. Calling exit
  3. Calling _exit or _Exit
  4. Return of the last thread from its start routine
  5. Calling pthread_exit from the last thread Abnormal termination occurs in three ways:
     a. Calling abort
     b. Receipt of a signal
     c. Response of the last thread to a cancellation request

Exit Functions

- Three functions terminate a program normally: _exit and _Exit, which return to the kernel immediately, and exit, which performs certain cleanup processing and then returns to the kernel.

```
#include <stdlib.h>
void exit(int status);
void _Exit(int status);
#include <unistd.h>
void _exit(int status);
```

- All three exit functions expect a single integer argument, called the exit status. Returning an integer value from the main function is equivalent to calling exit with the same value.
- Thus **exit(0);** is the same as **return(0);** from the main function.
- In the following situations the exit status of the process is undefined.
  1. any of these functions is called without an exit status.

2. main does a return without a return value
3. main "falls off the end", i.e if the exit status of the process is undefined. Example:

```
#include <stdio.h>
main()
{
    printf("hello, world\n");
}
```

```
$ cc hello.c
$ ./a.out
hello, world
$ echo $? // print the exit status
13
```

atexit Function

- With ISO C, a process can register up to 32 functions that are automatically called by exit. These are called exit handlers and are registered by calling the atexit function.

```
#include <stdlib.h>
int atexit(void (*func)(void));
```

Returns: 0 if OK, nonzero on error

- This declaration says that we pass the address of a function as the argument to atexit. When this function is called, it is not passed any arguments and is not expected to return a value.

- The exit function calls these functions in reverse order of their registration. Each function is called as many times as it was registered.

Example of exit handlers

```
#include "apue.h"
static void my_exit1(void);
static void my_exit2(void);
int main(void)
{
if (atexit(my_exit2) != 0)
err_sys("can't register my_exit2");
if (atexit(my_exit1) != 0)
err_sys("can't register my_exit1");
if (atexit(my_exit1) != 0)
err_sys("can't register my_exit1");
printf("main is done\n");
return(0);
}
static void
my_exit1(void)
{
printf("first exit handler\n");
}
static void
my_exit2(void)
{
printf("second exit handler\n");
}
```

Output:

```
$ ./a.out
main is done
first exit handler
first exit handler
second exit handler
```

The below figure summarizes how a C program is started and the various ways it can terminate.

4. COMMAND-LINE ARGUMENTS

- When a program is executed, the process that does the exec can pass command-line arguments to the new program.

Example: Echo all command-line arguments to standard output

```
#include "apue.h"
int main(int argc, char *argv[])
{
int i;
for (i = 0; i < argc; i++) /* echo all command-line args */
printf("argv[%d]: %s\n", i, argv[i]);
exit(0);
}
```

Output:

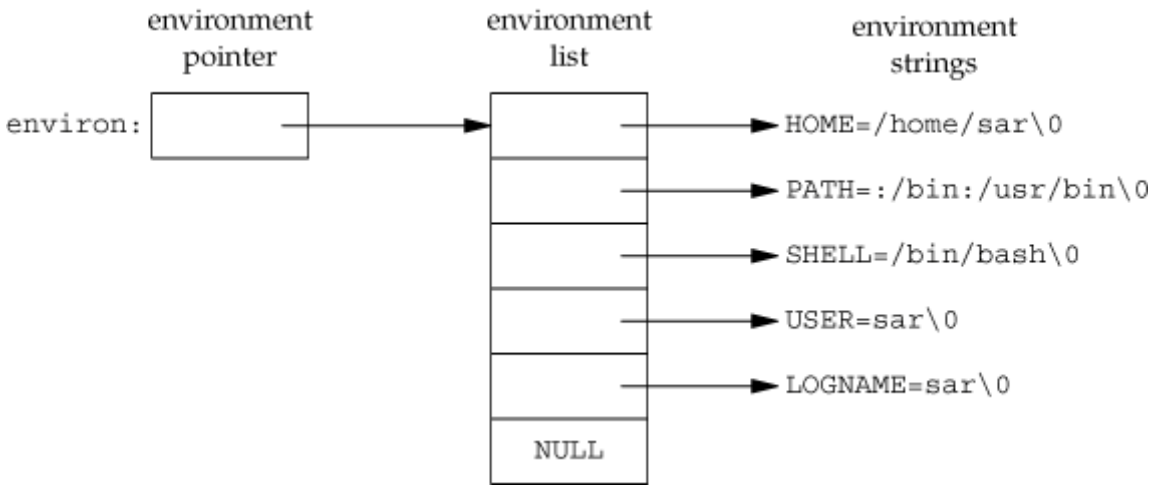
```
$ ./echoarg arg1 TEST foo
argv[0]: ./echoarg
argv[1]: arg1
argv[2]: TEST
argv[3]: foo
```

5. ENVIRONMENT LIST

- Each program is also passed an environment list. Like the argument list, the environment list is an array of character pointers, with each pointer containing the address of a null-terminated C string.

- The address of the array of pointers is contained in the global variable environ:
  extern char **environ;

Figure : Environment consisting of five C character strings



Generally any environmental variable is of the form: ***name = value.***

6. MEMORY LAYOUT OF A C PROGRAM

Historically, a C program has been composed of the following pieces:

- **Text segment**, the machine instructions that the CPU executes. Usually, the text segment is sharable so that only a single copy needs to be in memory for frequently executed programs, such as text editors, the C compiler, the shells, and so on. Also, the text segment is often read-only, to prevent a program from accidentally modifying its instructions.
- **Initialized data segment**, usually called simply the data segment, containing variables that are specifically initialized in the program. For example, the C declaration

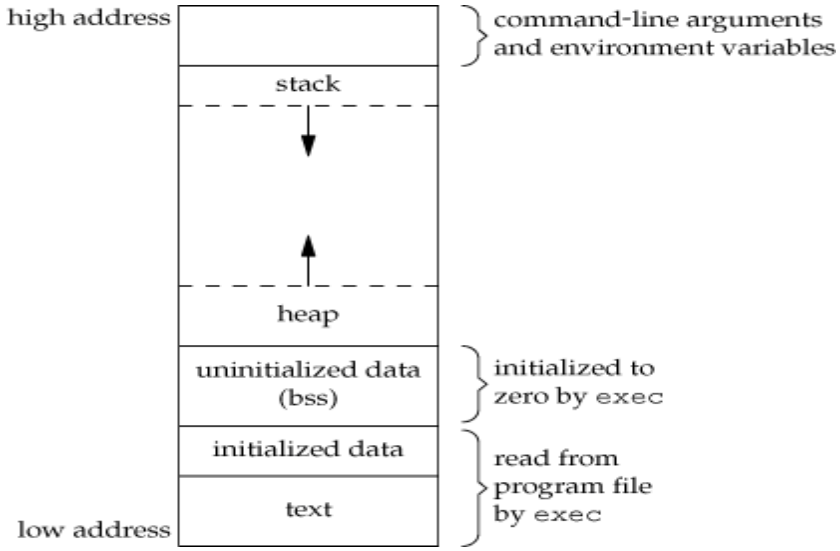
  int maxcount = 99;

  appearing outside any function causes this variable to be stored in the initialized data segment with its initial value.
- **Uninitialized data segment**, often called the "bss" segment, named after an ancient assembler operator that stood for "block started by symbol." Data in this segment is initialized by the kernel to arithmetic 0 or null pointers before the program starts executing. The C declaration

  long sum[1000];

  appearing outside any function causes this variable to be stored in the uninitialized data segment.
- **Stack**, where automatic variables are stored, along with information that is saved each time a function is called. Each time a function is called, the address of where to return to and certain information about the caller's environment, such as some of the machine registers, are saved on the stack. The newly called function then allocates room on the stack for its automatic and temporary variables. This is how recursive functions in C can work. Each time a recursive function calls itself, a new stack frame is used, so one set of variables doesn't interfere with the variables from another instance of the function.

- **Heap**, where dynamic memory allocation usually takes place. Historically, the heap has been located between the uninitialized data and the stack.



7. SHARED LIBRARIES

- Nowadays most UNIX systems support shared libraries. Shared libraries remove the common library routines from the executable file, instead maintaining a single copy of the library routine somewhere in memory that all processes reference.
- This reduces the size of each executable file but may add some runtime overhead, either when the program is first executed or the first time each shared library function is called. Another advantage of shared libraries is that, library functions can be replaced with new versions without having to re-link, edit every program that uses the library. With cc compiler we can use the option –g to indicate that we are using shared library.

8. MEMORY ALLOCATION

ISO C specifies three functions for memory allocation:

- malloc, which allocates a specified number of bytes of memory. The initial value of the memory is indeterminate.
- calloc, which allocates space for a specified number of objects of a specified size. The space is initialized to all 0 bits.

 r

ealloc, which increases or decreases the size of a previously allocated area. When the size increases, it may involve moving the previously allocated area somewhere else, to provide the additional room at the end. Also, when the size increases, the initial value of the space between the old contents and the end of the new area is indeterminate.

```
#include <stdlib.h>
void *malloc(size_t size);
void *calloc(size_t nobj, size_t size);
void *realloc(void *ptr, size_t
newsize);
All three return: non-null pointer if OK, NULL on error
void free(void *ptr);
```

- The pointer returned by the three allocation functions is guaranteed to be suitably aligned so that it

can be used for any data object.

- Because the three alloc functions return a generic void * pointer, if we #include <stdlib.h> (to obtain the function prototypes), we do not explicitly have to cast the pointer returned by these functions when we assign it to a pointer of a different type.
- The function free causes the space pointed to by ptr to be deallocated. This freed space is usually put into a pool of available memory and can be allocated in a later call to one of the three alloc functions.
- The realloc function lets us increase or decrease the size of a previously allocated area. For example, if we allocate room for 512 elements in an array that we fill in at runtime but find that we need room for more than 512 elements, we can call realloc. If there is room beyond the end of the existing region for the requested space, then realloc doesn't have to move anything; it simply allocates the additional area at the end and returns the same pointer that we passed it. But if there isn't room at the end of the existing region, realloc allocates another area that is large enough, copies the existing 512-element array to the new area, frees the old area, and returns the pointer to the new area.
- The allocation routines are usually implemented with the sbrk(2) system call. Although sbrk can expand or contract the memory of a process, most versions of malloc and free never decrease their memory size.
- The space that we free is available for a later allocation, but the freed space is not usually returned to the kernel; that space is kept in the malloc pool.
- It is important to realize that most implementations allocate a little more space than is requested and use the additional space for record keeping the size of the allocated block, a pointer to the next allocated block, and the like. This means that writing past the end of an allocated area could overwrite this record-keeping information in a later block. These types of errors are often catastrophic, but difficult to find, because the error may not show up until much later. Also, it is possible to overwrite this record keeping by writing before the start of the allocated area.
- Because memory allocation errors are difficult to track down, some systems provide versions of these functions that do additional error checking every time one of the three alloc functions or free is called. These versions of the functions are often specified by including a special library for the link editor. There are also publicly available sources that you can compile with special flags to enable additional runtime checking.

Alternate Memory Allocators

Many replacements for malloc and free are available.

- **libmalloc**

  SVR4-based systems, such as Solaris, include the libmalloc library, which provides a set of interfaces matching the ISO C memory allocation functions. The libmalloc library includes mallopt, a function that allows a process to set certain variables that control the operation of the storage allocator. A function called mallinfo is also available to provide statistics on the memory allocator.

- **vmalloc**

  Vo describes a memory allocator that allows processes to allocate memory using different techniques for different regions of memory. In addition to the functions specific to vmalloc, the library also provides emulations of the ISO C memory allocation functions.

- **quick-fit**

  Historically, the standard malloc algorithm used either a best-fit or a first-fit memory allocation strategy. Quick-fit is faster than either, but tends to use more memory. Free implementations of malloc and free based on quick-fit are readily available from several FTP sites.

- **alloca Function**

  The function alloca has the same calling sequence as malloc; however, instead of allocating memory from the heap, the memory is allocated from the stack frame of the current function. The advantage is that we don't have to free the space; it goes away automatically when the function returns. The alloca function increases the size of the stack frame. The disadvantage is that some systems can't support alloca, if it's impossible to increase the size of the stack frame after the function has been called.

9. ENVIRONMENT VARIABLES

- The environment strings are usually of the form: ***name=value.***
- The UNIX kernel never looks at these strings; their interpretation is up to the various applications
- . The shells, for example, use numerous environment variables.
- Some, such as HOME and USER, are set automatically at login, and others are for us to set. We normally set environment variables in a shell start-up file to control the shell's actions.

- The functions that we can use to set and fetch values from the variables are setenv, putenv, and getenv functions.
- The prototype of these functions are

```
#include <stdlib.h>
char *getenv(const char *name);
                                    Returns: pointer to value associated with name, NULL if not found.
```

- Note that this function returns a pointer to the value of a **name=value** string. We should always use getenv to fetch a specific value from the environment, instead of accessing environ directly.

- In addition to fetching the value of an environment variable, sometimes we may want to set an environment variable. We may want to change the value of an existing variable or add a new variable to the environment. The prototypes of these functions are

```
#include <stdlib.h>
int putenv(char
*str);
int setenv(const char *name, const char *value, int
rewrite); int unsetenv(const char *name);
                                    All return: 0 if OK, nonzero on error.
```

- The **putenv** function takes a string of the form **name=value** and places it in the environment list. If name already exists, its old definition is first removed.
- The **setenv** function sets name to value. If name already exists in the environment, then
  (a) if rewrite is nonzero, the existing definition for name is first removed;
  (b) if rewrite is 0, an existing definition for name is not removed, name is not set to the new value, and no error occurs.
- The **unsetenv** function removes any definition of name. It is not an error if such a definition does not exist.

  - ✓ Note the difference between **putenv** and **setenv**. Whereas **setenv** must allocate memory to create the **name=value** string from its arguments, **putenv** is free to place the string passed to

it directly into the environment.

NOTE:

1. If we're modifying an existing name:
   a) If the size of the new value is less than or equal to the size of the existing value, we can just copy the new string over the old string.
   b) If the size of the new value is larger than the old one, however, we must malloc to obtain room for the new string, copy the new string to this area, and then replace the old pointer in the environment list for name with the pointer to this allocated area.
2. If we're adding a new name, it's more complicated. First, we have to call malloc to allocate room for the name=value string and copy the string to this area.
   a) Then, if it's the first time we've added a new name, we have to call malloc to obtain room for a new list of pointers. We copy the old environment list to this new area and store a pointer to the name=value string at the end of this list of pointers. We also store a null pointer at the end of this list, of course. Finally, we set environ to point to this new list of pointers.
   b) If this isn't the first time we've added new strings to the environment list, then we know that we've already allocated room for the list on the heap, so we just call realloc to allocate room for one more pointer. The pointer to the new name=value string is stored at the end of the list (on top of the previous null pointer), followed by a null pointer.

10. setjmp AND longjmp FUNCTIONS

- In C, we can't goto a label that's in another function. Instead, we must use the setjmp and longjmp functions to perform this type of branching. As we'll see, these two functions are useful for handling error conditions that occur in a deeply nested function call.

```
#include <setjmp.h>
int setjmp(jmp_buf env);
Returns: 0 if called directly, nonzero if returning from a call to longjmp
void longjmp(jmp_buf env, int val);
```

- The setjmp function records or marks a location in a program code so that later when the longjmp function is called from some other function, the execution continues from the location onwards.
- The env variable (the first argument) records the necessary information needed to continue execution.
- The env is of the jmp_buf defined in <setjmp.h> file, it contains the task.

Example of setjmp and longjmp

```
#include "apue.h"
#include <setjmp.h>
#define TOK_ADD 5
jmp_buf jmpbuffer;
int main(void)
{
char line[MAXLINE];
if (setjmp(jmpbuffer) != 0)
printf("error");
while (fgets(line, MAXLINE, stdin) != NULL)
do_line(line);
exit(0);
}
...
void cmd_add(void)
{
int token;
token = get_token();
if (token < 0) /* an error has occurred */
longjmp(jmpbuffer, 1);
/* rest of processing for this command */
}
```

- The setjmp function always returns '0' on its success when it is called directly in a process (for the first time).
- The longjmp function is called to transfer a program flow to a location that was stored in the env argument.
- The program code marked by the env must be in a function that is among the callers of the current function.
- When the process is jumping to the target function, all the stack space used in the current function and it callers, upto the target function are discarded by the longjmp function.
- The process resumes execution by re-executing the setjmp statement in the target function that is marked by env. The return value of setjmp function is the value(val), as specified in the longjmp function call.
- The 'val' should be nonzero, so that it can be used to indicate where and why the longjmp function was invoked and process can do error handling accordingly.

Note: The values of ***automatic*** and ***register*** variables are indeterminate when the longjmp is called but static and global variable are unaltered. The variables that we don't want to roll back after longjmp are declared with keyword 'volatile'.

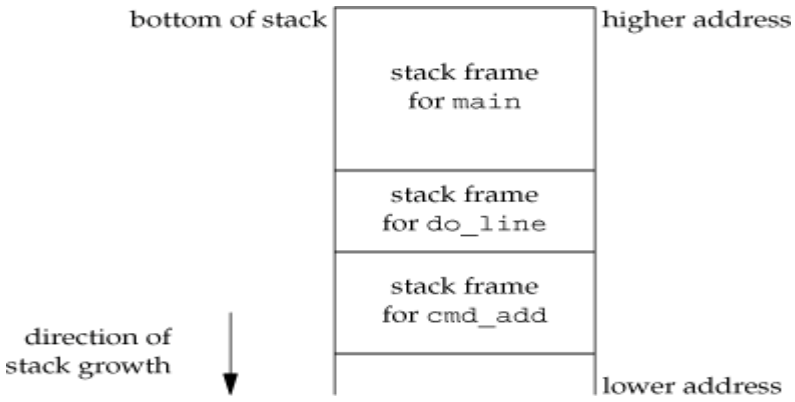


Figure: Stack frames after cmd_add has been called

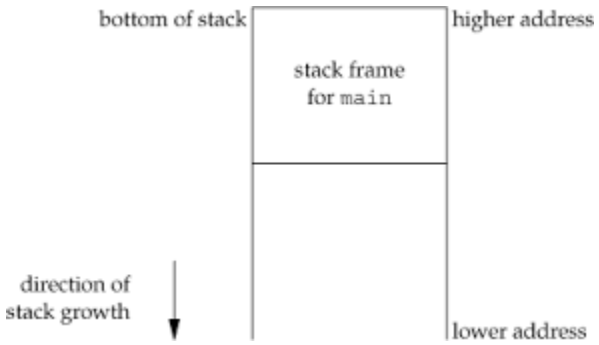


Figure: Stack frames after longjmp has been called

11.getrlimit AND setrlimit FUNCTIONS

- Every process has a set of resource limits, some of which can be queried and changed by the getrlimit and setrlimit functions.

```
#include <sys/resource.h>
int getrlimit(int resource, struct rlimit *rlptr);
int setrlimit(int resource, const struct rlimit *rlptr);
                                    Both return: 0 if OK, nonzero on error
```

Each call to these two functions specifies a single resource and a pointer to the following structure:

```
struct rlimit
{
rlim_t rlim_cur; /* soft limit: current limit */
rlim_t rlim_max; /* hard limit: maximum value for rlim_cur */
};
```

Three rules govern the changing of the resource limits.

1. A process can change its soft limit to a value less than or equal to its hard limit.
2. A process can lower its hard limit to a value greater than or equal to its soft limit. This lowering of the hard limit is irreversible for normal users.
3. Only a superuser process can raise a hard limit.

An infinite limit is specified by the constant RLIM_INFINITY. The resource argument takes on one of the following values:

RLIMIT_AS The maximum size in bytes of a process's total available memory.

RLIMIT_CORE The maximum size in bytes of a core file. A limit of 0 prevents the creation of a core file.

RLIMIT_CPU The maximum amount of CPU time in seconds. When the soft limit is exceeded, the SIGXCPU signal is sent to the process.

RLIMIT_DATA The maximum size in bytes of the data segment: the sum of the initialized data, uninitialized data, and heap.

RLIMIT_FSIZE The maximum size in bytes of a file that may be created. When the soft limit is exceeded, the process is sent the SIGXFSZ signal.

RLIMIT_LOCKS The maximum number of file locks a process can hold.

RLIMIT_MEMLOCK The maximum amount of memory in bytes that a process can lock into memory using mlock(2).

RLIMIT_NOFILE The maximum number of open files per process. Changing this limit affects the value returned by the sysconf function for its _SC_OPEN_MAX argument

RLIMIT_NPROC The maximum number of child processes per real user ID. Changing this limit affects the value returned for _SC_CHILD_MAX by the sysconf function

RLIMIT_RSS Maximum resident set size (RSS) in bytes. If available physical memory is low, the kernel takes memory from processes that exceed their RSS.

RLIMIT_SBSIZE The maximum size in bytes of socket buffers that a user can consume at any given time.

RLIMIT_STACK The maximum size in bytes of the stack.

RLIMIT_VMEM This is a synonym for RLIMIT_AS. The resource limits affect the calling process and are inherited by any of its children. This means that the setting of resource limits needs to be built into the shells to affect all our future processes.

Example: Print the current resource limits

```
#include "apue.h"
#if defined(BSD) || defined(MACOS)
#include <sys/time.h>
#define FMT "%10lld "
```

```
#else
#define FMT "%10ld "
#endif
#include <sys/resource.h>
#define doit(name) pr_limits(#name, name)
static void pr_limits(char *, int);
int main(void)
{
#ifdef RLIMIT_AS
doit(RLIMIT_AS);
#endif
doit(RLIMIT_CORE);
doit(RLIMIT_CPU);
doit(RLIMIT_DATA);
doit(RLIMIT_FSIZE);
#ifdef RLIMIT_LOCKS
doit(RLIMIT_LOCKS);
#endif
#ifdef RLIMIT_MEMLOCK
doit(RLIMIT_MEMLOCK);
#endif
doit(RLIMIT_NOFILE);
#ifdef RLIMIT_NPROC
doit(RLIMIT_NPROC);
#endif
#ifdef RLIMIT_RSS
doit(RLIMIT_RSS);
#endif
#ifdef RLIMIT_SBSIZE
doit(RLIMIT_SBSIZE);
#endif
```

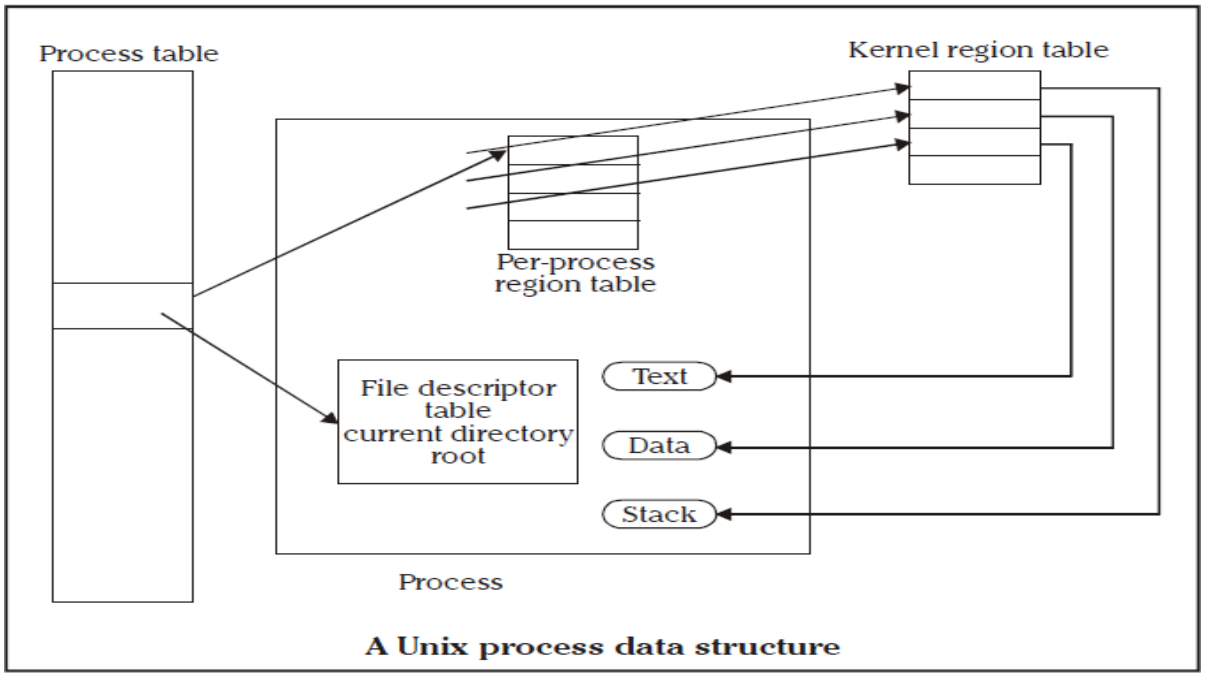
```
doit(RLIMIT_STACK);
#ifdef RLIMIT_VMEM
doit(RLIMIT_VMEM);
#endif
exit(0);
}
static void pr_limits(char *name, int resource)
```

```
{
struct rlimit limit;
if (getrlimit(resource, &limit) < 0)
err_sys("getrlimit error for %s", name);
printf("%-14s ", name);
if (limit.rlim_cur == RLIM_INFINITY)
printf("(infinite) ");
else
printf(FMT, limit.rlim_cur);
if (limit.rlim_max == RLIM_INFINITY)
printf("(infinite)");
else
printf(FMT, limit.rlim_max);
putchar((int)'\n');
}
```

12. UNIX KERNEL SUPPORT FOR PROCESS

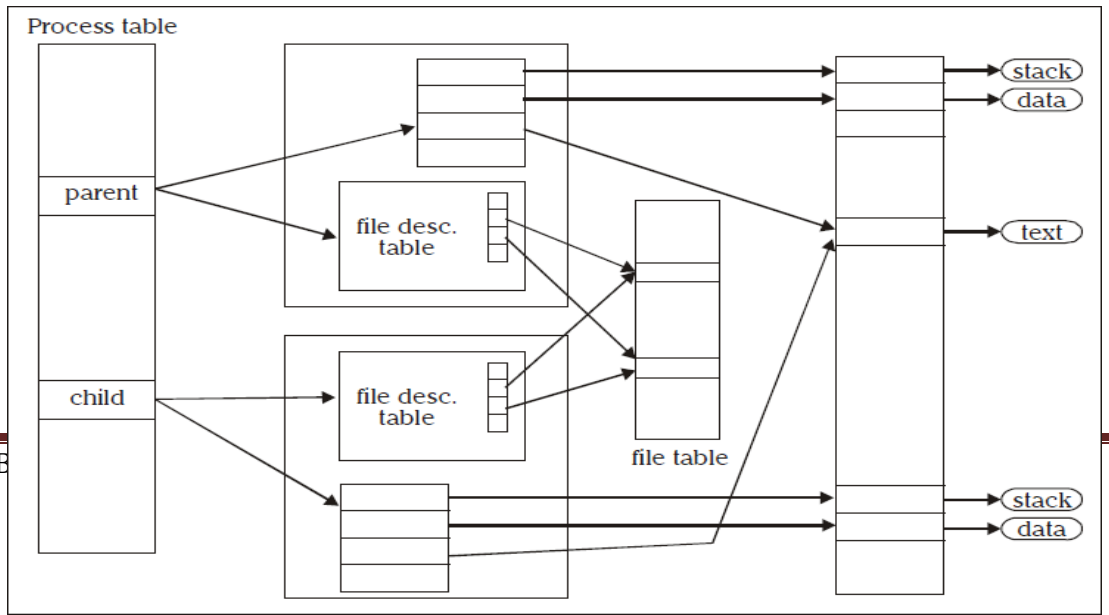
- The data structure and execution of processes are dependent on operating system implementation.
- A UNIX process consists minimally of a text segment, a data segment and a stack segment. A segment is an area of memory that is managed by the system as a unit.
  - ✓ A text segment consists of the program text in machine executable instruction code format.
  - ✓ The data segment contains static and global variables and their corresponding data.
  - ✓ A stack segment contains runtime variables and the return addresses of all active functions for a process.
- UNIX kernel has a process table that keeps track of all active process present in the system. Some of these processes belongs to the kernel and are called as “system process”.

- Every entry in the process table contains pointers to the text, data and the stack segments and also to U-area of a process.
- U-area of a process is an extension of the process table entry and contains other process specific data such as the file descriptor table, current root and working directory inode numbers and set of system imposed process limits.



A Unix process data structure

- All processes in UNIX system expect the process that is created by the system boot code, are created by the fork system call. After the fork system call, once the child process is created, both the parent and child processes resumes execution. When a process is created by fork, it contains duplicated



copies of the text, data and stack segments of its parent as shown in the Figure below. Also it has a file descriptor table, which contains reference to the same opened files as the parent, such that they both share the same file pointer to each opened files.

Figure: Parent & child relationship after fork

The process will be assigned with attributes, which are either inherited from its parent or will be set by the kernel.

- ✓ **A real user identification number (rUID)**: the user ID of a user who created the parent process.
- ✓ **A real group identification number (rGID)**: the group ID of a user who created that parent process.
- ✓ **An effective user identification number (eUID)**: this allows the process to access and create files with the same privileges as the program file owner.
- ✓ **An effective group identification number (eGID)**: this allows the process to access and create files with the same privileges as the group to which the program file belongs.
- ✓ **Saved set-UID and saved set-GID**: these are the assigned eUID and eGID of the process respectively.
- ✓ **Process group identification number (PGID) and session identification number (SID):** these identify the process group and session of which the process is member.
- ✓ **Supplementary group identification numbers:** this is a set of additional group IDs for a user who created the process.
- ✓ **Current directory:** this is the reference (inode number) to a working directory file.
- ✓ **Root directory**: this is the reference to a root directory.
- ✓ **Signal handling**: the signal handling settings.
- ✓ **Signal mask**: a signal mask that specifies which signals are to be blocked.
- ✓ **Unmask**: a file mode mask that is used in creation of files to specify which accession rights should be taken out.
- ✓ **Nice value**: the process scheduling priority value.
- ✓ **Controlling terminal**: the controlling terminal of the process.

- In addition to the above attributes, the following attributes are different between the parent and child processes:

Process identification number (PID): an integer identification number that is unique per process in an entire operating system.

Parent process identification number (PPID): the parent process PID.

Pending signals: he set of signals that are pending delivery to the parent process.

Alarm clock time: the process alarm clock time is reset to zero in the child process.

File locks: the set of file locks owned by the parent process is not inherited by the chid process.

fork and ***exec*** are commonly used together to spawn a sub-process to execute a different program. The advantages of this method are:

- ✓ A process can create multiple processes to execute multiple programs concurrently.
- ✓ Because each child process executes in its own virtual address space, the parent process is not affected by the execution status of its child process.

Chapter 3: Process Control

1. Introduction
2. Process Identifiers
3. fork
4. vfork
5. exit
6. wait
7. waitpid
8. wait3
9. wait4 Functions
10. Race Conditions
11. exec

1. INTRODUCTION

Process control is concerned about creation of new processes, program execution, and process termination.

2. PROCESS IDENTIFIERS

- Every process has a unique process ID, anon negative integer.
- There are some processes:
  Process ID 0 is usually the scheduler process known as swapper
  Process ID 1 is usually the init process and is invoked by the kernel at the end of the bootstrap procedure
  Process ID 2 is the page deamon responsible for supporting the paging of the virtual memory system.
- In addition to the process ID, there are other identifier's for every process. The following function retien these identifiers.

#include <unistd.h>
pid_t getpid(void);
Returns: process ID of calling process
pid_t getppid(void);
Returns: parent process ID of calling process
uid_t getuid(void);
Returns: real user ID of calling process
uid_t geteuid(void);
Returns: effective user ID of calling process
gid_t getgid(void);
Returns: real group ID of calling process
gid_t getegid(void);
Returns: effective group ID of calling process

3. fork FUNCTION

- An existing process can create a new one by calling the fork function.

```
#include <unistd.h>
pid_t fork(void);
                                    Returns: 0 in child, process ID of child in parent, 1 on error.
```

- The new process created by fork is called the child process.
- This function is called once but returns twice.
- The only difference in the returns is that the return value in the child is 0, whereas the return value in the parent is the process ID of the new child.
- The reason the child's process ID is returned to the parent is that a process can have more than one child, and there is no function that allows a process to obtain the process IDs of its children.
- The reason fork returns 0 to the child is that a process can have only a single parent, and the child can always call getppid to obtain the process ID of its parent. (Process ID 0 is reserved for use by the kernel, so it's not possible for 0 to be the process ID of a child.)
- Both the child and the parent continue executing with the instruction that follows the call to fork.
- The child is a copy of the parent.
- For example, the child gets a copy of the parent's data space, heap, and stack.
- Note that this is a copy for the child; the parent and the child do not share these portions of memory.
- The parent and the child share the text segment.

Example programs:

Program 1

```
/* Program to demonstrate fork function
Program name – fork1.c */
#include<sys/types.h>
#include<unistd.h>
int main( )
{
fork( );
printf("\n hello USP");
}
```

Output :

```
$ cc fork1.c
$ ./a.out
hello USP
hello USP
```

Program 2

```
/* Program name – fork2.c */
#include<sys/types.h>
#include<unistd.h>
int main( )
{
printf("\n 6 sem ");
fork( );
printf("\n hello USP");
}
```

Output :

```
$ cc fork1.c
$ ./a.out
6 sem
hello USP
hello USP
```

File Sharing

Consider a process that has three different files opened for standard input, standard output, and standard error. On return from fork, we have the arrangement shown in Figure :

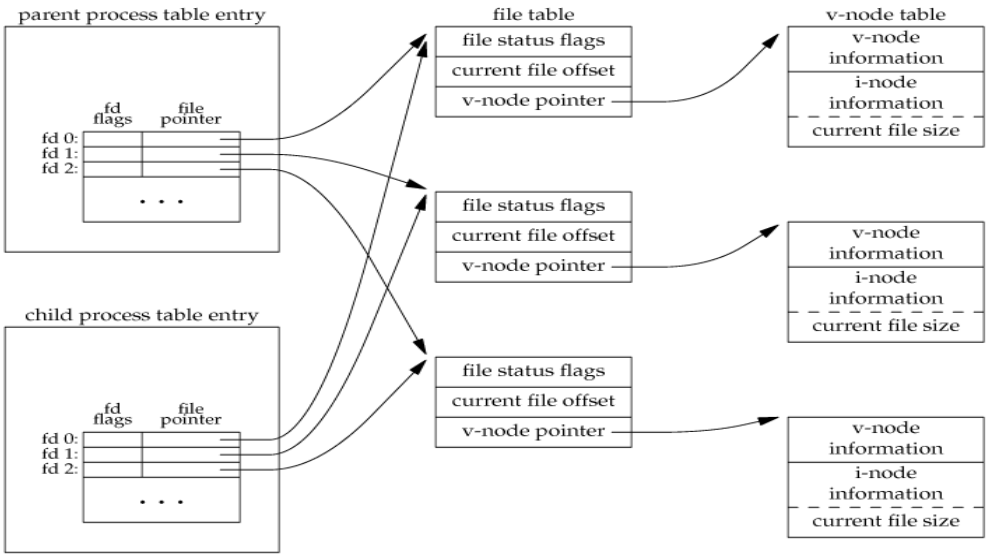


Figure:Sharing of open files between parent and child after fork

- It is important that the parent and the child share the same file offset.
- Consider a process that forks a child, then waits for the child to complete.
- Assume that both processes write to standard output as part of their normal processing.
- If the parent has its standard output redirected (by a shell, perhaps) it is essential that the parent's file offset be updated by the child when the child writes to standard output.
- In this case, the child can write to standard output while the parent is waiting for it; on completion of the child, the parent can continue writing to standard output, knowing that its output will be appended to whatever the child wrote.
- If the parent and the child did not share the same file offset, this type of interaction would be more difficult to accomplish and would require explicit actions by the parent.

There are two normal cases for handling the descriptors after a fork.

1. The parent waits for the child to complete. In this case, the parent does not need to do anything with its descriptors. When the child terminates, any of the shared descriptors that the child read from or wrote to will have their file offsets updated accordingly.
2. Both the parent and the child go their own ways. Here, after the fork, the parent closes the descriptors that it doesn't need, and the child does the same thing. This way, neither interferes with the other's open descriptors. This scenario is often the case with network servers.

There are numerous other properties of the parent that are inherited by the child:

- Real user ID, real group ID, effective user ID, effective group ID
- Supplementary group IDs
- Process group ID
- Session ID
- Controlling terminal
- The set-user-ID and set-group-ID flags
- Current working directory
- Root directory
- File mode creation mask
- Signal mask and dispositions
- The close-on-exec flag for any open file descriptors
- Environment
- Attached shared memory segments
- Memory mappings
- Resource limits

The differences between the parent and child are

- The return value from fork
- The process IDs are different
- The two processes have different parent process IDs: the parent process ID of the child is the parent; the

parent process ID of the parent doesn't change.

- The child's tms_utime, tms_stime, tms_cutime, and tms_cstime values are set to 0
- File locks set by the parent are not inherited by the child
- Pending alarms are cleared for the child
- The set of pending signals for the child is set to the empty set

The two main reasons for fork to fail are

(a) if too many processes are already in the system, which usually means that something else is wrong, or

(b) if the total number of processes for this real user ID exceeds the system's limit.

There are two uses for fork:

- When a process wants to duplicate itself so that the parent and child can each execute different sections of code at the same time. This is common for network servers, the parent waits for a service request from a client. When the request arrives, the parent calls fork and lets the child handle the request. The parent goes back to waiting for the next service request to arrive.
- When a process wants to execute a different program. This is common for shells. In this case, the child does an exec right after it returns from the fork.

4. vfork FUNCTION

- The function vfork has the same calling sequence and same return values as fork.
- The vfork function is intended to create a new process when the purpose of the new process is to exec a new program.
- The vfork function creates the new process, just like fork, without copying the address space of the parent into the child, as the child won't reference that address space; the child simply calls exec (or exit) right after the vfork.
- Instead, while the child is running and until it calls either exec or exit, the child runs in the address space of the parent. This optimization provides an efficiency gain on some paged virtual-memor implementations of the UNIX System.

- Another difference between the two functions is that vfork guarantees that the child runs first, until the child calls exec or exit. When the child calls either of these functions, the parent resumes.

Example of vfork function

```
#include "apue.h"
int glob = 6; /* external variable in initialized data */
int main(void)
{
int var; /* automatic variable on the stack */
pid_t pid;
var = 88;
printf("before vfork\n"); /* we don't flush stdio */
if ((pid = vfork()) < 0) {
err_sys("vfork error");
} else if (pid == 0) { /* child */
glob++; /* modify parent's variables */
var++;
_exit(0); /* child terminates */
}
/*
* Parent continues here.
*/
printf("pid = %d, glob = %d, var = %d\n", getpid(), glob, var);
exit(0);
}
```

Output:

```
$ ./a.out
before vfork
pid = 29039, glob = 7, var = 89
```

5. exit FUNCTIONS

- A process can terminate normally in five ways:

1. Executing a return from the main function.
2. Calling the exit function.
3. Calling the _exit or _Exit function.
   In most UNIX system implementations, exit(3) is a function in the standard C library, whereas _exit(2) is a system call.
4. Executing a return from the start routine of the last thread in the process. When the last thread returns from its start routine, the process exits with a termination status of 0.
5. Calling the pthread_exit function from the last thread in the process.

The three forms of abnormal termination are as follows:

1. Calling abort. This is a special case of the next item, as it generates the SIGABRT signal.
2. When the process receives certain signals. Examples of signals generated by the kernel include the process referencing a memory location not within its address space or trying to divide by 0.
3. The last thread responds to a cancellation request. By default, cancellation occurs in a deferred manner: one thread requests that another be canceled, and sometime later, the target thread terminates.

6. wait AND waitpid FUNCTIONS

- When a process terminates, either normally or abnormally, the kernel notifies the parent by sending the SIGCHLD signal to the parent. Because the termination of a child is an asynchronous event - it can happen at any time while the parent is running - this signal is the asynchronous notification from the kernel to the parent.
- The parent can choose to ignore this signal, or it can provide a function that is called when the signal occurs: a signal handler.
- A process that calls wait or waitpid can:
    - Block, if all of its children are still running
    - Return immediately with the termination status of a child, if a child has terminated and is

waiting for its termination status to be fetched

- Return immediately with an error, if it doesn't have any child processes.

```
#include <sys/wait.h>
pid_t wait(int *statloc);
pid_t waitpid(pid_t pid, int *statloc, int options);
                                    Both return: process ID if OK, 0 (see later), or 1 on error.
```

The differences between these two functions are as follows.

- ✓ The wait function can block the caller until a child process terminates, whereas waitpid has an option that prevents it from blocking.
- ✓ The waitpid function doesn't wait for the child that terminates first; it has a number of options that control which process it waits for.

- ➢ If a child has already terminated and is a zombie, wait returns immediately with that child's status. Otherwise, it blocks the caller until a child terminates. If the caller blocks and has multiple children, wait returns when one terminates.
- ➢ For both functions, the argument statloc is a pointer to an integer. If this argument is not a null pointer, the termination status of the terminated process is stored in the location pointed to by the argument.

Print a description of the exit status

```
#include "apue.h"
#include <sys/wait.h>
Void pr_exit(int status)
{
if (WIFEXITED(status))
        printf("normal termination, exit status = %d\n", WEXITSTATUS(status));
else if (WIFSIGNALED(status))
        printf("abnormal termination, signal number = %d%s\n",WTERMSIG(status),
#ifdef WCOREDUMP
                        WCOREDUMP(status) ? " (core file generated)" : "");
#else
                        "");
#endif
      else if (WIFSTOPPED(status))
           printf("child stopped, signal number = %d\n", WSTOPSIG(status));
}
```

Program to Demonstrate various exit statuses

```
#include "apue.h"
#include <sys/wait.h>
Int main(void)
{
pid_t pid;
int status;
if ((pid = fork()) < 0)
err_sys("fork error");
else if (pid == 0) /* child */
exit(7);
if (wait(&status) != pid) /* wait for child */
err_sys("wait error");
pr_exit(status); /* and print its status */
if ((pid = fork()) < 0)
err_sys("fork error");
else if (pid == 0) /* child */
abort(); /* generates SIGABRT */
if (wait(&status) != pid) /* wait for child */
err_sys("wait error");
pr_exit(status); /* and print its status */
```

```
if ((pid = fork()) < 0)
err_sys("fork error");
else if (pid == 0) /* child */
status /= 0; /* divide by 0 generates SIGFPE */
if (wait(&status) != pid) /* wait for child */
err_sys("wait error");
pr_exit(status); /* and print its status */
exit(0);
}
```

The interpretation of the pid argument for waitpid depends on its value:

pid == 1 Waits for any child process. In this respect, waitpid is equivalent to wait.

pid > 0 Waits for the child whose process ID equals pid.

pid == 0 Waits for any child whose process group ID equals that of the calling process.

pid < 1 Waits for any child whose process group ID equals the absolute value of pid.

Macros to examine the termination status returned by wait **and** waitpid

Macro	**Description**
WIFEXITED(status)	True if status was returned for a child that terminated normally. In this case, we can execute WEXITSTATUS (status) to fetch the low-order 8 bits of the argument that the child passed to exit, _exit,or _Exit.
WIFSIGNALED (status)	True if status was returned for a child that terminated abnormally, by receipt of a signal that it didn't catch. In this case, we can execute WTERMSIG (status) to fetch the signal number that caused the termination. Additionally, some implementations (but not the Single UNIX Specification) define the macro WCOREDUMP (status) that returns true if a core file of the terminated process was generated.
WIFSTOPPED (status)	True if status was returned for a child that is currently stopped. In this case, we can execute WSTOPSIG (status) to fetch the signal number that caused the child to stop.
WIFCONTINUED (status)	True if status was returned for a child that has been continued after a job control stop

The options constants for waitpid

Constant	Description
WCONTINUED	If the implementation supports job control, the status of any child specified by pid that has been continued after being stopped, but whose status has not yet been reported, is returned.
WNOHANG	The waitpid function will not block if a child specified by pid is not immediately available. In this case, the return value is 0.
WUNTRACED	If the implementation supports job control, the status of any child specified by pid that has stopped, and whose status has not been reported since it has stopped, is returned. The WIFSTOPPED macro determines whether the return value corresponds to a stopped child process.

- ✓ The waitpid function provides three features that aren't provided by the wait function.
- ✓ The waitpid function lets us wait for one particular process, whereas the wait function returns the status of any terminated child. We'll return to this feature when we discuss the popen function.
- ✓ The waitpid function provides a nonblocking version of wait. There are times when we want to fetch a child's status, but we don't want to block.
- ✓ The waitpid function provides support for job control with the WUNTRACED and WCONTINUED options.

Program to Avoid zombie processes by calling fork twice

```
#include "apue.h"
#include <sys/wait.h>
Int main(void)
{
pid_t pid;
if ((pid = fork()) < 0) {
err_sys("fork error");
} else if (pid == 0) { /* first child */
if ((pid = fork()) < 0)
err_sys("fork error");
else if (pid > 0)
exit(0); /* parent from second fork == first child */
/*
* We're the second child; our parent becomes init as soon
* as our real parent calls exit() in the statement above.
* Here's where we'd continue executing, knowing that when
* we're done, init will reap our status.
*/
```

```
sleep(2);
printf("second child, parent pid = %d\n", getppid());
exit(0);
}
if (waitpid(pid, NULL, 0) != pid) /* wait for first child */
err_sys("waitpid error");
/*
* We're the parent (the original process); we continue executing,
* knowing that we're not the parent of the second child.
*/
exit(0);
}
```

Output:

```
$ ./a.out
$ second child, parent pid = 1
```

7. waitid FUNCTION

The waitid function is similar to waitpid, but provides extra flexibility.

```
#include <sys/wait.h>
int waitid (idtype_t idtype, id_t id, siginfo_t *infop, int options);
                                                    Returns: 0 if OK, -1 on error
```

The *idtype* constants for waited are as follows:

Constant	Description
P_PID	Wait for a particular process: id contains the process ID of the child to wait for.
P_PGID	Wait for any child process in a particular process group: id contains the process group ID of the children to wait for.
P_ALL	Wait for any child process: id is ignored.

The options argument is a bitwise OR of the flags as shown below: these flags indicate which state changes the caller is interested in.

Constant	Description
WCONTINUED	Wait for a process that has previously stopped and has been continued, and whose status has not yet been reported.
WEXITED	Wait for processes that have exited.
WNOHANG	Return immediately instead of blocking if there is no child exit status available.
WNOWAIT	Don't destroy the child exit status. The child's exit status can be retrieved by a subsequent call to wait, waitid,or waitpid.
WSTOPPED	Wait for a process that has stopped and whose status has not yet been reported.

8. wait3 AND wait4 FUNCTIONS

- The only feature provided by these two functions that isn't provided by the wait, waitid, and waitpid functions is an additional argument that allows the kernel to return a summary of the resources used by the terminated process and all its child processes.
- The prototypes of these functions are:

```
#include <sys/types.h>
#include <sys/wait.h>
#include <sys/time.h>
#include <sys/resource.h>

pid_t wait3(int *statloc, int options, struct rusage *rusage);
pid_t wait4(pid_t pid, int *statloc, int options, struct rusage *rusage);

                                        Both return: process ID if OK,-1 on error
```

- The resource information includes such statistics as the amount of user CPU time, the amount of system

CPU time, number of page faults, number of signals received etc. the resource information is available only for terminated child process not for the process that were stopped due to job control.

9. RACE CONDITIONS

- A race condition occurs when multiple processes are trying to do something with shared data and the final outcome depends on the order in which the processes run.
- **Example**: The program below outputs two strings: one from the child and one from the parent.
- The program contains a race condition because the output depends on the order in which the processes are run by the kernel and for how long each process runs.

```
#include "apue.h"
static void charatatime(char *);
int main(void)
{
pid_t pid;
if ((pid = fork()) < 0) {
err_sys("fork error");
} else if (pid == 0) {
charatatime("output from child\n");
} else {
charatatime("output from parent\n");
}
exit(0);
}
static void
charatatime(char *str)
{
char *ptr;
int c;
setbuf(stdout, NULL); /* set unbuffered */
for (ptr = str; (c = *ptr++) != 0; )
putc(c, stdout);
}
```

Output:

```
$ ./a.out
ooutput from child
utput from parent
$ ./a.out
ooutput from child
utput from parent
$ ./a.out
output from child
output from parent
```

program modification to avoid race condition

Program to use TELL and WAIT functions

```
#include "apue.h"
static void charatatime(char *);
int main(void)
{
pid_t pid;
+ TELL_WAIT();
+
if ((pid = fork()) < 0) {
```

```
err_sys("fork error");
} else if (pid == 0) {
+ WAIT_PARENT(); /* parent goes first */
charatatime("output from child\n");
} else {
charatatime("output from parent\n");
+ TELL_CHILD(pid);
}
exit(0);
}
static void
charatatime(char *str)
{
char *ptr;
int c;
setbuf(stdout, NULL); /* set unbuffered */
for (ptr = str; (c = *ptr++) != 0; )
putc(c, stdout);
}
```

- When we run this program, the output is as we expect; there is no intermixing of output from the two processes.

10. exec FUNCTIONS

- When a process calls one of the exec functions, that process is completely replaced by the new program, and the new program starts executing at its main function.
- The process ID does not change across an exec, because a new process is not created; exec merely replaces the current process - its text, data, heap, and stack segments - with a brand new program from disk.
- There are 6 exec functions:

```
#include <unistd.h>

int execl(const char *pathname, const char *arg0,... /* (char *)0 */ );
int execv(const char *pathname, char *const argv []);
int execle(const char *pathname, const char *arg0,... /*(char *)0, char *const envp */ );
int execve(const char *pathname, char *const argv[], char *const envp[]);
int execlp(const char *filename, const char *arg0, ... /* (char *)0 */ );
int execvp(const char *filename, char *const argv []);
```

All six return: -1 on error, no return on success.

- The first difference in these functions is that the first four take a pathname argument, whereas the last two take a filename argument. When a filename argument is specified
  - ✓ If filename contains a slash, it is taken as a pathname.
  - ✓ Otherwise, the executable file is searched for in the directories specified by the PATH environment variable.
- The next difference concerns the passing of the argument list (l stands for list and v stands for vector).
- The functions execl, execlp, and execle require each of the command-line arguments to the new program to be specified as separate arguments. For the other three functions (execv, execvp, and execve), we have to build an array of pointers to the arguments, and the address of this array is the argument to these three functions.
- The final difference is the passing of the environment list to the new program. The two functions whose names end in an e (execle and execve) allow us to pass a pointer to an array of pointers to the environment strings. The other four functions, however, use the environ variable in the calling process to copy the existing environment for the new program.

Function	*pathname*	*filename*	Arg list	*argv*[]	environ	*envp*[]
execl	•		•		•	
execlp		•	•		•	
execle	•		•			•
execv	•			•	•	
execvp		•		•	•	
execve	•			•		•
(letter in name)		p	l	v		e

The above table shows the differences among the 6 exec functions.

- We've mentioned that the process ID does not change after an exec, but the new program inherits additional properties from the calling process:

  Process ID and parent process ID
  Real user ID and real group ID
  Supplementary group IDs
  Process group ID
  Session ID
  Controlling terminal
  Time left until alarm clock
  Current working directory
  Root directory
  File mode creation mask
  File locks
  Process signal mask
  Pending signals
  Resource limits
  Values for tms_utime, tms_stime, tms_cutime, and tms_cstime.

Example of exec functions

```
#include "apue.h"
#include <sys/wait.h>
char *env_init[] = { "USER=unknown", "PATH=/tmp", NULL };
int main(void)
{
pid_t pid;
if ((pid = fork()) < 0) {
err_sys("fork error");
} else if (pid == 0) { /* specify pathname, specify environment */
if (execle("/home/sar/bin/echoall", "echoall", "myarg1",
"MY ARG2", (char *)0, env_init) < 0)
err_sys("execle error");
}
if (waitpid(pid, NULL, 0) < 0)
err_sys("wait error");
if ((pid = fork()) < 0) {
err_sys("fork error");
} else if (pid == 0) { /* specify filename, inherit environment */
if (execlp("echoall", "echoall", "only 1 arg", (char *)0) < 0)
err_sys("execlp error");
}
exit(0);
}
```

Output:

```
$ ./a.out
argv[0]: echoall
argv[1]: myarg1
argv[2]: MY ARG2
USER=unknown
PATH=/tmp
$ argv[0]: echoall
argv[1]: only 1 arg
USER=sar
LOGNAME=sar
SHELL=/bin/bash
47 more lines that aren't shown
HOME=/home/sar
```

- Note that the shell prompt appeared before the printing of argv[0] from the second exec. This is because the parent did not wait for this child process to finish.

Echo all command-line arguments and all environment strings

```
#include "apue.h"
int main(int argc, char *argv[])
{
int i;
char **ptr;
extern char **environ;
for (i = 0; i < argc; i++) /* echo all command-line args */
printf("argv[%d]: %s\n", i, argv[i]);
for (ptr = environ; *ptr != 0; ptr++) /* and all env strings */
printf("%s\n", *ptr);
exit(0);
}
```

Chapter 1: Process Control

1. Changing User IDs and Group IDs
2. Interpreter Files
3. System Function
4. Process Accounting
5. User Identification
6. Process Times
7. I/O Redirection

1. CHANGING USER IDs AND GROUP IDs

- When our programs need additional privileges or need to gain access to resources that they currently aren't allowed to access, they need to change their user or group ID to an ID that has the appropriate privilege or access.
- Similarly, when our programs need to lower their privileges or prevent access to certain resources, they do so by changing either their user ID or group ID to an ID without the privilege or ability access to the resource.

```
#include <unistd.h>

int setuid(uid_t uid);
int setgid(gid_t gid);
```

Both return: 0 if OK, 1 on error

There are rules for who can change the IDs. Let's consider only the user ID for now. (Everything we describe for the user ID also applies to the group ID.)

- If the process has superuser privileges, the setuid function sets the real user ID, effective user ID, and saved set-user-ID to uid.
- If the process does not have superuser privileges, but uid equals either the real user ID or the saved set-user- ID, setuid sets only the effective user ID to uid. The real user ID and the saved set-user-ID are not changed.
- If neither of these two conditions is true, errnois set to EPERM, and 1 is returned.

We can make a few statements about the three user IDs that the kernel maintains.

- Only a superuser process can change the real user ID. Normally, the real user ID is set by the login(1) program when we log in and never changes. Because login is a superuser process, it sets all three user IDs when it calls setuid.
- The effective user ID is set by the exec functions only if the set-user-ID bit is set for the program file. If the set-user-ID bit is not set, the exec functions leave the effective user ID as its current

value. We can call setuid at any time to set the effective user ID to either the real user ID or the saved set-user-ID. Naturally, we can't set the effective user ID to any random value.

- The saved set-user-ID is copied from the effective user ID by exec. If the file's set-user-ID bit is set, this copy is saved after execstores the effective user ID from the file's user ID.

setreuidand setregidFunctions

- ✓ Swapping of the real user ID and the effective user ID with the setreuid function.

```
#include <unistd.h>

int setreuid(uid_t ruid, uid_t euid);

int setregid(gid t rgid, gid t egid);
```

Both return : 0 if OK, -1 on error

- ✓ We can supply a value of 1 for any of the arguments to indicate that the corresponding ID should remain unchanged.
- ✓ The rule is simple: an unprivileged user can always swap between the real user ID and the effective user ID.
- ✓ This allows a set-user-ID program to swap to the user's normal permissions and swap back again later for set-user- ID operations.

seteuid and setegid functions :

- POSIX.1 includes the two functions seteuid and setegid. These functions are similar to setuid and setgid, but only the effective user ID or effective group ID is changed.

```
#include  <unistd.h>

int seteuid(uid_t uid);

int setegid(gid_t gid);
```

Both return : 0 if OK, 1 on error

- An unprivileged user can set its effective user ID to either its real user ID or its saved set-user-ID.
- For a privileged user, only the effective user ID is set to uid. (This differs from the setuid function, which changes all three userIDs)

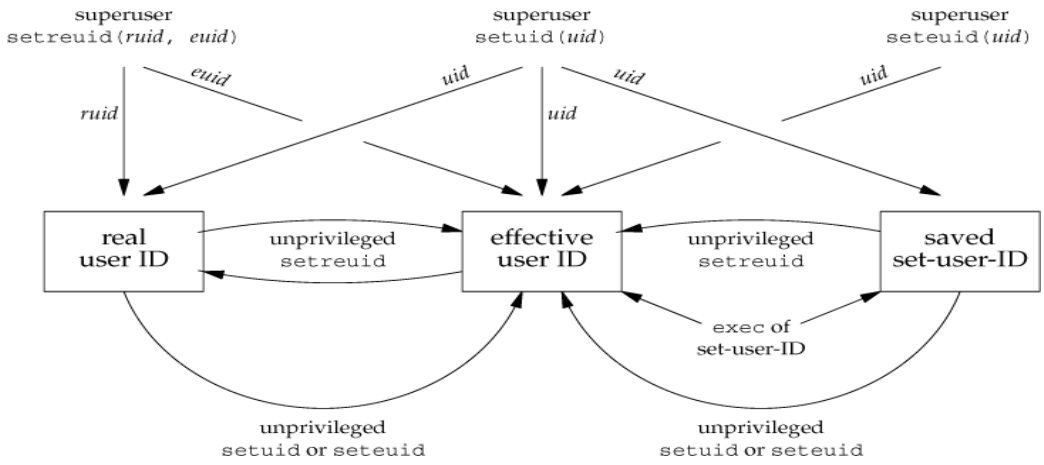


Figure: Summary of all the functions that set the various user Ids

2. INTERPRETER FILES

- These files are text files that begin with a line of the form
- #! pathname [optional-argument]
- The space between the exclamation point and the pathname is optional. The most common of these interpreter files begin with the line
- #!/bin/sh
- The pathname is normally an absolute pathname, since no special operations are performed on it (i.e., PATH is not used).
- The recognition of these files is done within the kernel as part of processing the exec system call.
- The actual file that gets executed by the kernel is not the interpreter file, but the file specified by the pathname on the first line of the interpreter file.
- Be sure to differentiate between the interpreter filea text file that begins with #!and the interpreter, which is specified by the pathname on the first line of the interpreter file.
- Be aware that systems place a size limit on the first line of an interpreter file. This limit includes the #!, the pathname, the optional argument, the terminating newline, and any spaces.

A program that execs an interpreter file

```
#include "apue.h" #include<sys/wait.h>
int main(void)
{
    pid_t   pid;
    if ((pid = fork()) < 0) {
        err_sys("fork error");
    } else if (pid == 0) {          /* child */
```

```
        if (execl("/home/sar/bin/testinterp","testinterp", "myarg1", "MY ARG2", (char *)0)< 0)
         err_sys("execl error");
    }
    if (waitpid(pid, NULL, 0) < 0) /* parent
        */ err_sys("waitpid error");
    exit(0);
}
```

Output:

```
$ cat
/home/sar/bin/testinterp
#!/home/sar/bin/echoarg foo
$ ./a.out
argv[0]:
/home/sar/bin/echoarg
argv[1]:  foo
argv[2]:
/home/sar/bin/testinterp
argv[3]: myarg1
argv[4]: MY ARG2
```

system FUNCTION

```
#include <stdlib.h>
int system(const char *cmdstring);
```

- If cmdstring is a null pointer, system returns nonzero only if a command processor is available. This feature determines whether the system function is supported on a given operating system.
- Under the UNIX System, system is always available.

Because system is implemented by calling fork, exec, and waitpid, there are three types of return values.

1. If either the forkfails or waitpid returns an error other than EINTR, systemreturns 1 with errnoset toindicate the error.
2. If the execfails, implying that the shell can't be executed, the return value is as if the shell had executedexit(127).
3. Otherwise, all three functions fork, exec, and waitpid succeed, and the return value from system is thetermination status of the shell, in the format specified for waitpid.

Program: The systemfunction, without signal handling

```
#include<sys/wait.h>
#include<errno.h>
#include<unistd.h>
int system(const char *cmdstring)    /* version without signal handling */
{
```

```
    if (cmdstring == NULL)
        return(1);      /* always a command processor with UNIX */

    if ((pid = fork()) < 0)
    {
        status = -1;    /* probably out of processes */
    }
    else if (pid == 0)
     {  execl("/bin/sh", "sh", "-c", cmdstring, (char *)0);
        _exit(127);     /* execl error */
     }
    else {                                /* parent */
        while (waitpid(pid, &status, 0) < 0)
        { if (errno != EINTR) {
                status = -1;
            }
        }
    }

    return(status);
}
```

Program: Calling the systemfunction

```
#include "apue.h"
#include <sys/wait.h>
int main(void)
{
    int status;
    if ((status = system("date")) < 0)
        err_sys("system() error");
    pr_exit(status);

   if ((status = system("nosuchcommand")) < 0)
       err_sys("system() error");
   pr_exit(status);

   if ((status = system("who; exit 44")) < 0)
       err_sys("system() error");
   pr_exit(status);

   exit(0);
}
```

Program: Execute the command-line argument using system

```
#include "apue.h"

int main(int argc, char *argv[])
{
```

```
    int status;

    if (argc < 2)
        err_quit("command-line argument required");

    if ((status = system(argv[1])) < 0)
        err_sys("system() error");
    pr_exit(status);

    exit(0);
}
```

Program: Print real and effective user IDs

```
#include "apue.h"

int main(void)
{
printf("real uid = %d, effective uid = %d\n", getuid(), geteuid());
exit(0);
}
```

3. PROCESS ACCOUNTING

- Most UNIX systems provide an option to do process accounting. When enabled, the kernel writes anaccounting record each time a process terminates.
- These accounting records are typically a small amount of binary data with the name of the command, theamount of CPU time used, the user ID and group ID, the starting time, and so on.
- A super user executes accton with a pathname argument to enable accounting.
- The accounting records are written to the specified file, which is usually /var/account/acct. Accounting is turned off by executing accton without any arguments.
- The data required for the accounting record, such as CPU times and number of characters transferred, is kept by the kernel in the process table and initialized whenever a new process is created, as in the child after a fork.
- Each accounting record is written when the process terminates.
- This means that the order of the records in the accounting file corresponds to the termination order of the processes, not the order in which they were started.
- The accounting records correspond to processes, not programs.
- A new record is initialized by the kernel for the child after a fork, not when a new program is executed. The structure of the accounting records is defined in the header <sys/acct.h> and looks something like

```
typedef  u_short comp_t;   /* 3-bit base 8 exponent; 13-bit fraction */

struct  acct
{
  char   ac_flag;            /* flag  */
  char   ac_stat;            /* termination status (signal & core flag only) */
```

```
                                /* (Solaris only) */
    uid_t  ac_uid;              /* real user ID */
    gid_t  ac_gid;              /* real group ID */
    dev_t  ac_tty;              /* controlling terminal */
    time_t ac_btime;            /* starting calendar time */
    comp_t ac_utime;            /* user CPU time (clock ticks) */
    comp_t ac_stime;            /* system CPU time (clock ticks) */
    comp_t ac_etime;            /* elapsed time (clock ticks) */
    comp_t ac_mem;              /* average memory usage */
    comp_t ac_io;               /* bytes transferred (by read and write) */
                                /* "blocks" on BSD systems */
    comp_t ac_rw;               /* blocks read or written */
                                /* (not present on BSD systems) */
    char   ac_comm[8];          /* command name: [8] for Solaris, */
                                /* [10] for Mac OS X, [16] for FreeBSD, and */
                                /* [17] for Linux */
};
```

The ac_flag member records certain events during the execution of the process.

ac_flag	Description
AFORK	process is the result of `fork`, but never called `exec`
ASU	process used superuser privileges
ACOMPAT	process used compatibility mode
ACORE	process dumped core
AXSIG	process was killed by a signal
AEXPND	expanded accounting entry

Process structure for accounting example

4. USER IDENTIFICATION

- Any process can find out its real and effective user ID and group ID.
- Sometimes, however, we want to find out the login name of the user who's running the program.
- We could call getpwuid(getuid()), but what if a single user has multiple login names, each with the same user ID? (A person might have multiple entries in the password file with the same user ID to have a different login shell for each entry).
- The system normally keeps track of the name we log in and the getlogin function provides a way to fetch that login name.

```
#include <unistd.h>
char *getlogin(void);
```

Returns : pointer to string giving login name if OK, NULL on error

- This function can fail if the process is not attached to a terminal that a user logged in to.

5. PROCESS TIMES

- ✓ We describe three times that we can measure: wall clock time, user CPU time, and system CPU time. Any process can call the times function to obtain these values for itself and any terminated children.

```
#include<sys/times.h>
Clock_t times(struct tms * buf)
```

Returns: elapsed wall clock time in clock ticks if OK, 1 on error

- ✓ This function fills in the tms structure pointed to by buf:

```
struct tms
  {
  clock_t       tms_utime;      /* user CPU time */
  clock_t       tms_stime;            /* system CPU time */
  clock_t       tms_cutime; /* user CPU time, terminated children */
  clock_t tms_cstime;                 /* system CPU time, terminated children */
  };
```

- ✓ Note that the structure does not contain any measurement for the wall clock time.
- ✓ Instead, the function returns the wall clock time as the value of the function, each time it's called.
- ✓ This value is measured from some arbitrary point in the past, so we can't use its absolute value; instead, we use its relative value.

6. I/O Redirection

- It scans the command line for the occurrence of the special redirection characters <,>,or >>
- Unix provides the capability to change where standard input comes from or where ouput goesusing a concept called Input/Output(I/O) redirection.
- I/O redirection is accomplished using a redirection operator which allows the user to specify theinput or output data be directed to a file.
- The output redirection operator is the >(greater than) symbol and general syntax:
  - **command > output_file_spec**
- Spaces around the redirection is not mandatory, but to add readability to the command.

Eg: **$ls > my_files [Enter]** **$ cat my_files [Enter]** **foo** **bar** **fred** **dino** **$**	**$ echo "Hello World!" > my_files [Enter]** **$ cat my_files [Enter]** **Hello World!**

- The append operator is the >>

```
$ ls > my_files [Enter]
$ echo "Hello World!" >>  my_files [Enter]
$ cat my_files [Enter]
        foo
        bar
        fred
        dino
        Hello World!
```

- The first output redirection creates the file if it does not exist or overwrites its content if it does and thesecond redirection appends the string "Hello World!" to the end of the file.
- When using the append redirection operator, if the file does not exist, >> will cause its creation andappend the output (to the empty file).
- The ability also exists to redirect the standard input using the input redirection operator, the < (less than)symbol
- The general syntax of input redirection:

  command < input_file_spec

Chapter 2: INTERPROCESS COMMUNICATION

Overview of IPC Methods

1. Pipes
2. Popen and pclose Functions
3. Coprocesses
4. FIFOs
5. System V IPC
6. Message Queues
7. Semaphores

INTRODUCTION

- ✓ IPC enables one application to control another application, and for several applications to share the same data without interfering with one another. IPC is required in all multiprocessing systems, but it is not generally supported by single-process operating systems.
- ✓ The various forms of IPC that are supported on a UNIX system are as follows :

1) Half duplex Pipes
2) FIFO's
3) Full duplex Pipes
4) Named full duplex Pipes
5) Message queues
6) Shared memory
7) Semaphores
8) Sockets
9) STREAMS

- ✓ The first seven forms of IPC are usually restricted to IPC between processes on the same host.
- ✓ The final two i.e. Sockets and STREAMS are the only two that are generally supported for IPC between processes on different hosts.

1. PIPES

- ✓ Pipes are the oldest form of UNIX System IPC. Pipes have two limitations.
- ✓ Historically, they have been half duplex (i.e., data flows in only one direction).
- ✓ Pipes can be used only between processes that have a common ancestor.
- ✓ Normally, a pipe is created by a process, that process calls fork, and the pipe is used between the parentand the child.

- ✓ A pipe is created by calling the pipe function.

```
#include <unistd.h>
int pipe(int filedes[2]);
                                Returns: 0 if OK, 1 on error.
```

- ✓ Two file descriptors are returned through the filedes argument: filedes[0] is open for reading, andfiledes[1] is open for writing.
- ✓ The output of filedes[1] is the input for filedes[0].

- ✓ Two ways to picture a half-duplex pipe are shown in Figure 1.
- ✓ The left half of the figure shows the two ends of the pipe connected in a single process. The right half ofthe figure emphasizes that the data in the pipe flows through the kernel.

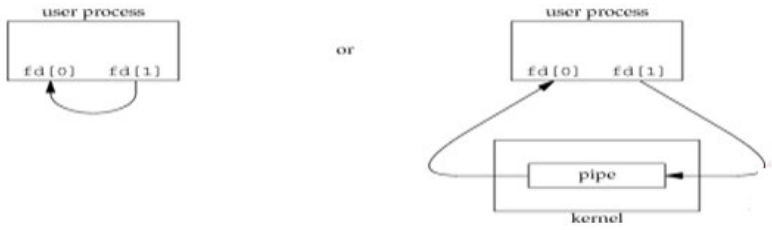


Figure 1. Two ways to view a half-duplex pipe

- ✓ A pipe in a single process is next to useless.
- ✓ Normally, the process that calls pipe then calls fork, creating an IPC channel from the parent to the child orvice versa. Figure 2 shows this scenario.

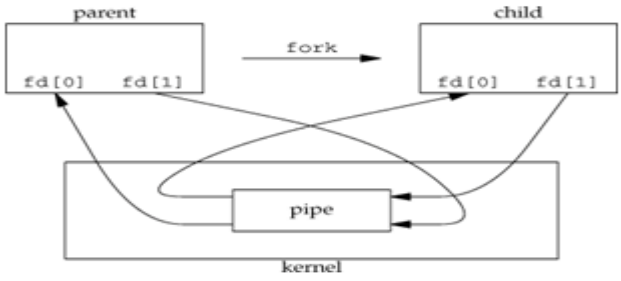


Figure 2 Half-duplex pipe after a fork

- ✓ What happens after the fork depends on which direction of data flow we want.
- ✓ For a pipe from the parent to the child, the parent closes the read end of the pipe (fd[0]), and the child closes the write end (fd[1]). Figure 3 shows the resulting arrangement of descriptors.

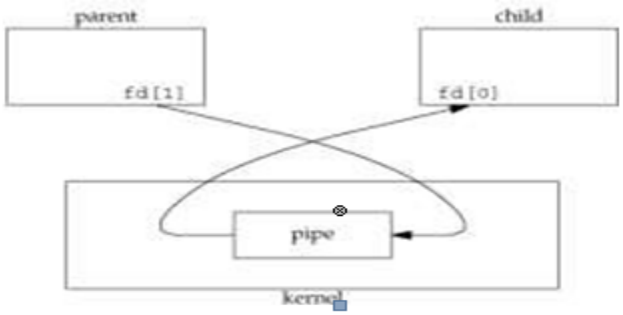


Figure 3 Pipe from parent to child

- ✓ For a pipe from the child to the parent, the parent closes fd[1], and the child closes fd[0]. When one end ofa pipe is closed, the following two rules apply.
  - ➢ If we read from a pipe whose write end has been closed, read returns 0 to indicate an end of file after all the data has been read.
  - ➢ If we write to a pipe whose read end has been closed, the signal SIGPIPE is generated. If we either ignore the signal or catch it and return from the signal handler, write returns 1 with errno set to EPIPE.

PROGRAM: shows the code to create a pipe between a parent and its child and to send data down the pipe.

```
#
i
n
c
l
u
d
e

"
a
p
u
e
.
h
"

i
n
t

m
a
i
n
(
v
o
i
d
)
{
```

```
int     n;
int     fd[2];
pid_t   pid;
char    line[MAXLINE];

i
    f

    (
    p
    i
    p
    e
    (
    f
    d
    )

    <

    0
    )

    e
    r
    r
    _
    s
    y
    s
    (
    "
    p
    i
    p
    e

    e
    r
    r
    o
    r
    "
    )
    ;
i
    f

    (
    (
    p
    i
    d

    =

    f
    o
    r
    k
    (
    )
```

```
    )

    <

    0
    )

    {

    e
    r
    r

    _
    s
    y
    s
    (
    "
    f
    o
    r
    k

    e
    r
    r
    o
    r
    "
    )
    ;
}
    e
    l
    s
    e

    i
    f

    (
    p
    i
    d

    >

    0
    )

    {
                            /*parent */ close(fd[0]);
    write(fd[1], "hello world\n", 12);
} else {                /*
    c
    h
    i
    l
    d

    *
    /
```

```
close(fd[1]);
n = read(fd[0], line, MAXLINE);
write(STDOUT_FILEN
```

```
            o
            ,
            l
            i
            n
            e
            ,
            n
            )
            ;
        }
        exit(0);
}
```

2. popen AND pcloseFUNCTIONS

- ✓ Since a common operation is to create a pipe to another process, to either read its output or send it input, the standard I/O library has historically provided the popen and pclose functions.
- ✓ These two functions handle all the dirty work that we've been doing ourselves: creating a pipe, forking a child, closing the unused ends of the pipe, executing a shell to run the command, and waiting for the command to terminate.

```
#include <stdio.h>

FILE *popen(const char *cmdstring, const char *type);

                                    Returns: file pointer if OK, NULL on error
```

```
int pclose(FILE *fp);

                        Returns: termination status of cmdstring, or 1 on error
```

- ✓ The function popen does a fork and exec to execute the cmdstring, and returns a standard I/Ofile pointer.

  If type is "r", the file pointer is connected to the standard output of cmdstring

Figure 4 Result of fp = popen(cmdstring, "r")

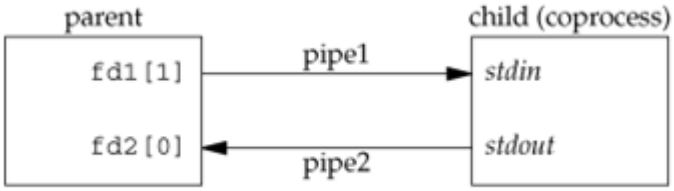
If type is "w", the file pointer is connected to the standard input of cmdstring, as shown:

Figure 5 Result of fp = popen(cmdstring, "w")

3. COPROCESSES

- ✓ A UNIX system filter is a program that reads from standard input and writes to standard output.
- ✓ Filters are normally connected linearly in shell pipelines.
- ✓ A filter becomes a coprocess when the same program generates the filter's input and reads the filter'soutput.
- ✓ A coprocess normally runs in the background from a shell, and its standard input and standard output are connected to another program using a pipe.
- ✓ The process creates two pipes: one is the standard input of the coprocess, and the other is the standard output of the coprocess. Figure 6 shows this arrangement.

Figure 6. Driving a coprocess by writing its standard input and reading its standard output



Program: Simple filter to add two numbers

```
#
i
n
c
l
u
d
e

"
a
p
u
e
.
h
"

i
n
```

```
t

m
a
i
n
(
v
o
i
d
)
{
    i
    n
    t

    n
    ,

    i
    n
    t
    1
    ,

    i
    n
    t
    2
    ;

    c
    h
    a
    r

    l
    i
    n
    e
    [
    M
    A
    X
    L
    I
    N
    E
    ]
    ;
    while ((n = read(STDIN_FILENO, line, MAXLINE)) > 0)
    {    line[n] = 0;       /* null terminate */

        if (sscanf(line, "%d%d", &int1, &int2) == 2)

        {
            sprintf
            (line,
            "%d\n",
            int1 +
            int2);n
```

```
            =
            strlen(
            line);
            if
                  (write(S
                  TDOUT_FI
                  LENO,
                  line, n)
                  != n)
                  err_sys(
                  "write
                  error");
        } else {
            if (write(STDOUT_FILENO,
                "invalid args\n", 13)
                != 13)err_sys("write
                error");
        }
    }
    exit(0);
}
```

4. FIFOs

- ✓ FIFOs are sometimes called named pipes. Pipes can be used only between related processes when a common ancestor has created the pipe.

```
#include <sys/stat.h>
int mkfifo(const char *pathname, mode_t mode);
                                                    Returns: 0 if OK, -1 on error
```

Once we have used mkfifo to create a FIFO, we open it using open. When we open a FIFO, the nonblocking flag (O_NONBLOCK) affects what happens.

In the normal case (O_NONBLOCK not specified), an open for read-only blocks until some other process opens the FIFO for writing. Similarly, an open for write-only blocks until some other process opens the FIFO for reading.

If O_NONBLOCK is specified, an open for read-only returns immediately. But an open for write-only returns 1 with errno set to ENXIOif no process has the FIFO open for reading.

There are two uses for FIFOs.

- ✓ FIFOs are used by shell commands to pass data from one shell pipeline to another without creating intermediate temporary files.
- ✓ FIFOs are used as rendezvous points in client-server applications to pass data between the clients andthe servers.

Example Using FIFOs to Duplicate Output Streams

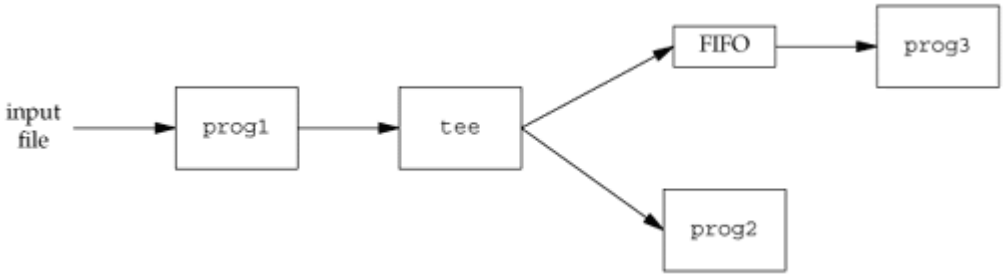
- ✓ FIFOs can be used to duplicate an output stream in a series of shell commands.
- ✓ This prevents writing the data to an intermediate disk file. Consider a procedure that needs toprocess a filtered input stream twice. Figure shows this arrangement.

FIGURE : Procedure that processes a filtered input stream twice

- ✓ With a FIFO and the UNIX program tee(1), we can accomplish this procedure without using a temporary file. (The tee program copies its standard input to both its standard output and to the file named on its command line.)

```
m
k
f
i
f
o

f
i
f
o
1

p
r
o
g
3
< fifo1 &

prog1 < infile | tee fifo1 | prog2
```

- ✓ We create the FIFO and then start prog3 in the background, reading from the FIFO. We then start prog1 and use tee to send its input to both the FIFO and prog2. Figure shows the process arrangement.



- ✓ FIGURE : Using a FIFO and tee to send a stream to two different processes
- ✓ Example Client-Server Communication Using a FIFO
- ✓ FIFO's can be used to send data between a client and a server. If we have a server that is contacted by numerous clients, each client can write its request to a well-known FIFO that the server creates. Since there are multiple writers for the FIFO, the requests sent by the clients to the server need to be less than PIPE_BUF bytes in size.
- ✓ This prevents any interleaving of the client writes. The problem in using FIFOs for this type of client server communication is how to send replies back from the server to each client.
- ✓ A single FIFO can't be used, as the clients would never know when to read their response versus responses for other clients. One solution is for each client to send its process ID with the request. The server then creates a unique FIFO for each client, using a pathname based on the client'sprocess ID.
- ✓ For example, the server can create a FIFO with the name /vtu/ ser.XXXXX, where XXXXX is replaced with the client's process ID. This arrangement works, although it is impossible for the server to tell whether a client crashes. This causes the client-specific FIFOs to be left in the file system.
- ✓ The server also must catch SIGPIPE, since it's possible for a client to send a request and terminate before reading the response, leaving the client-specific FIFO with one writer (the server) and no reader.

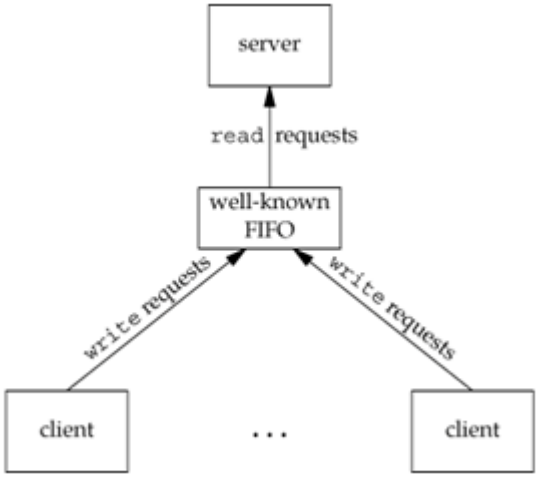


Figure : Clients sending requests to a server using a FIFO

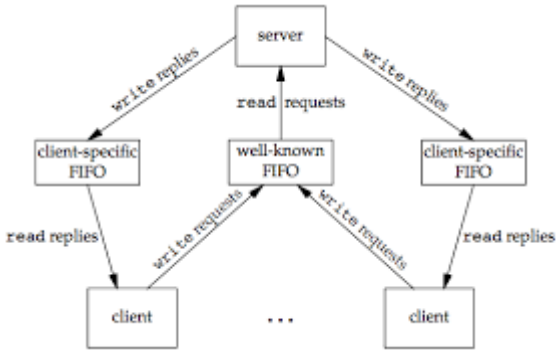


Figure: Client-server communication using FIFOs

5. System V IPC

❖ Identifiers and Keys

Each IPC structure (message queue, semaphore, or shared memory segment) in the kernel is referred to by a non- negative integer identifier. The identifier is an internal name for an IPC object. Cooperating processes need an external naming scheme to be able to rendezvous using the same IPC object. For this purpose, an IPC object is associated with a key that acts as an external name.

Whenever an IPC structure is being created, a key must be specified. The data type of this key is the primitive system data type key_t, which is often defined as a long integer in the header <sys/types.h>. This key is converted into an identifier by the kernel.

There are various ways for a client and a server to rendezvous at the same IPC structure.

- ✓ The server can create a new IPC structure by specifying a key of IPC_PRIVATE and store the returned

  identifier somewhere (such as a file) for the client to obtain. The key IPC_PRIVATE guarantees that the server creates a new IPC structure. The disadvantage to this technique is that file system operations are required for the server to write the integer identifier to a file, and then for the clients to retrieve this identifier later.

  The IPC_PRIVATE key is also used in a parent-child relationship. The parent creates a new IPC structure specifying IPC_PRIVATE, and the resulting identifier is then available to the child after the fork. The child can pass the identifier to a new program as an argument to one of the execfunctions.

- ✓ The client and the server can agree on a key by defining the key in

a common header, for example. The server then creates a new IPC structure specifying this key. The problem with this approach is that it's possible for the key to already be associated with an IPC structure, in which case the get function (msgget, semget, or shmget) returns an error. The server must handle this error, deleting the existing IPC structure, and try to create it again.

- ✓ The client and the server can agree on a pathname and project ID (the project ID is a character value between 0 and 255) and call the function ftok to convert these two values into a key. This key is then used in step 2. The only service provided by ftok is a way of generating a key from a pathname and project ID.

```
#include <sys/ipc.h>
key_t ftok(const char *path, int id);
```

Returns: key if OK, `(key_t)`-1 on error

- ✓ The path argument must refer to an existing file. Only the lower 8 bits of id are used when generating the key.
- ✓ The key created by ftok is usually formed by taking parts of the st_dev and st_ino fields in the stat structure corresponding to the given pathname and combining them with the project ID.
- ✓ If two pathnames refer to two different files, then ftok usually returns two different keys for the two pathnames. However, because both i-node numbers and keys are often stored in long integers, there can be information loss creating a key. This means that two different pathnames to different files can generate the same key if the same project ID isused.

❖ **Permission Structure**

- ✓ XSI IPC associates an ipc_perm structure with each IPC structure. This structure defines the permissions and owner and includes at least the following members:

```
struct ipc_perm
{
      uid_t uid;  /* owner's
      effective user id */
      gid_t gid; /* owner's
      effective group id */
      uid_t cuid; /*
      creator's effective
      user id */ gid_t cgid;
      /* creator's effective
      group id */mode_t mode;
      /* access modes */
      .
      .
       };
```

- ✓ All the fields are initialized when the IPC structure is created. At a

later time, we can modify the uid, gid, and mode fields by calling msgctl, semctl, or shmctl. To change these values, the calling process must be either the creator of the IPC structure or the superuser. Changing these fields is similar to calling chown or chmod for a file.

Permission	Bit
user-read	0400
user-write (alter)	0200
group-read	0040
group-write (alter)	0020
other-read	0004
other-write (alter)	0002

XSI IPC permissions

❖ **Advantages and Disadvantages**

✓ A fundamental problem with XSI IPC(System V IPC) is that the IPC structures are system wide and do not have a reference count. For example, if we create a message queue, place some messages on the queue, and then terminate, the message queue and its contents are not deleted. They remain in the system until specifically read or deleted by some process calling msgrcv or msgctl, by someone executing the ipcrm(1) command, or by the system being rebooted. Compare this with a pipe, which is completely removed when the last process to reference it terminates. With a FIFO, although the name stays in the file system until explicitly removed, any data left in a FIFO is removed when the last process to reference the FIFO terminates.

✓ Another problem with XSI IPC (System V IPC) is that these IPC structures are not known by names in the file system. We can't access them and modify their properties with the functions. Almost a dozen new system calls (msgget, semop, shmat, and so on) were added to the kernel to support these IPC objects. We can't see the IPC objects with an ls command, we can't remove them with the rm command, and we can't change their permissions with the chmod command. Instead, two new commands ipcs(1) and ipcrm(1)were added.

✓ Since these forms of IPC don't use file descriptors, we can't use the multiplexed I/O functions (select and poll) with them. This makes it harder to use more than one of these IPC structures at a time or to use any of these IPC structures with file or device I/O. For example, we can't have a server wait for a message to be placed on one of two message queues without some form of busy wait loop.

6. MESSAGE QUEUES

✓ A message queue is a linked list of messages stored within the

kernel and identified by a message queue identifier. We'll call the message queue just a queue and its identifier a queue ID.

- ✓ 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.

Each queue has the following msqid_dsstructure associated with it:

```
struct msqid_ds
{
        Struct ipc_perm  msg_perm;
        msgqnum_t  msg_qnum;          /*  #
        of  messages  on  queue  */ msglen_t
        msg_qbytes;                   /*
        max  #  of  bytes  on  queue  */ pid_t
                msg_lspid;            /* pid
        of last msgsnd() */
        pid_t      msg_lspid;         /*
        pid of last msgrcv() */time_t
                msg_stime;            /*
        last-msgsnd() time */ time_t
                msg_rtime;            /*
        last-msgrcv() time */ time_t
                msg_ctime;            /*
        last-change time */
        .
        .
};
```

This structure defines the current status of the queue.

msgget

- ✓ The first function normally called is msgget to either open an existing queue or create a new queue.

```
#include <sys/msg.h>
int msgget(key_t key, int flag);
                                        Returns: message queue ID if OK, 1 on error
```

When a new queue is created, the following members of the msqid_dsstructure are initialized.

- ✓ The ipc_perm structure is initialized. The mode member of this structure is set to the correspondingpermission bits of flag.
- ✓ msg_qnum, msg_lspid, msg_lrpid, msg_stime, and msg_rtimeare all set to 0.
- ✓ msg_ctimeis set to the current time.
- ✓ msg_qbytesis set to the system limit.

On success, msgget returns the non-negative queue ID. This value is then used with the other threemessage queue functions.

Msgctl

- ✓ The msgctlfunction performs various operations on a queue.

```
#include <sys/msg.h>
int msgctl(int msqid, int cmd, struct msqid_ds *buf );
                                                    Returns: 0 if OK, -1 c
```

The cmd argument specifies the command to be performed on the queue specified by msqid.

Msgsnd

- ✓ Data is placed onto a message queue by calling msgsnd.

```
#include <sys/msg.h>
int msgsnd(int msqid, const void *ptr, size_t nbytes, int flag);
                                                    Returns: 0 if OK, -1 on error
```

- ✓ Each message is composed of a positive long integer type field, a non-negative length (nbytes), and the actual data bytes (corresponding to the length). Messages are always placed at the end of the queue.
- ✓ The ptr argument points to a long integer that contains the positive integer message type, and it is immediately followed by the message data. (There is no message data if nbytes is 0.) If the largest message we send is 512 bytes, we can define the following structure:

```
struct mymesg
{
  long  mtype;      /* positive message type */
  char  mtext[512]; /* message data, of length nbytes */
};
```

- ✓ The ptr argument is then a pointer to a mymesg structure. The message type can be used by the receiver tofetch messages in an order other than first in, first out.

msgrcv

✓ Messages are retrieved from a queue by msgrcv

```
#include <sys/msg.h>
ssize_t msgrcv(int msqid, void *ptr, size_t nbytes, long type, int flag);
                                    Returns: size of data portion of message if OK, -1 on error.
```

The type argument lets us specify which message we want.

type == 0	The first message on the queue is returned.
type > 0	The first message on the queue whose message type equals type is returned.
type < 0	The first message on the queue whose message type is the lowest value less than or equal to the absolute value of type is returned.

7. SEMAPHORES

A semaphore is a counter used to provide access to a shared data object for multiple processes.

To obtain a shared resource, a process needs to do the following:

1. Test the semaphore that controls the resource.
2. If the value of the semaphore is positive, the process can use the resource. In this case, the process

   decrements the semaphore value by 1, indicating that it has used one unit of the resource.
3. Otherwise, if the value of the semaphore is 0, the process goes to sleep until the semaphore value is greater than 0. When the process wakes up, it returns to step 1.

When a process is done with a shared resource that is controlled by a semaphore, the semaphore value is incremented by 1. If any other processes are asleep, waiting for the semaphore, they are awakened.
A common form of semaphore is called a ***binary semaphore***. It controls a single resource, and its value is initialized to 1. In general, however, a semaphore can be initialized to any positive value, with the value indicating how many units of the shared resource are available for sharing.
XSI semaphores are, unfortunately, more complicated than this. Three features contribute to this unnecessary complication.

1. A semaphore is not simply a single non-negative value. Instead, we have to define a semaphore as a set of one or more semaphore values. When we create a semaphore, we specify the number of values in the set.
2. The creation of a semaphore (semget) is independent of its initialization (semctl). This is a fatal flaw, since we cannot atomically create a new semaphore set and initialize all the values in the set.

3. Since all forms of XSI IPC remain in existence even when no process is using them, we have to worry about a program that terminates without releasing the semaphores it has been allocated. The undo feature that we describe later is supposed to handle this.

The kernel maintains a semid_dsstructure for each semaphore set:

```
struct semid_ds
    {
        struct ipc_perm   sem_perm;
        unsigned short    sem_nsems; /* #
        of semaphores in set */time_t
                  sem_otime; /* last-semop()
        time */
        time_t   sem_ctime; /* last-change time */
.
.
.
}
;
```

Each semaphore is represented by an anonymous structure containing at least the following members:

```
    struct
    {
    unsigned short semval; /*
    semaphore value, always >= 0 */
    pid_t          sempid;  /* pid for
    last operation */
    unsigned short semncnt; /* # processes
    awaiting semval>curval */unsigned short
    semzcnt; /* # processes awaiting
    semval==0 */
    .
    .
    .
  };
```

semget

The first function to call is semget to obtain a semaphore ID.

```
#include <sys/sem.h>

int semget(key_t key, int nsems, int flag);
```

Returns: semaphore ID if OK, 1 c

When a new set is created, the following members of the semid_ds structure are initialized.

- The ipc_perm structure is initialized. The mode member of this structure is set to the correspondingpermission bits of flag.
- sem_otime is set to 0.

- sem_ctime is set to the current time.
- sem_nsems is set to nsems.
- The number of semaphores in the set is nsems. If a new set is being created (typically in the server), we must specify nsems. If we are referencing an existing set (a client), we can specify nsems as 0.

Semctl

- ✓ The semctl function is the catchall for various semaphore operations.

```
#include <sys/sem.h>
int semctl(int semid, int semnum, int cmd,... /* union semun arg */);
```

- ✓ The fourth argument is optional, depending on the command requested, and if present, is of type semun, a union of various command-specific arguments:

```
union semun
{
    intval;                          /* for SETVAL */
    struct semid_ds *buf;            /*
    for IPC_STAT and IPC_SET */unsigned
    short  *array;                   /*
    for GETALL and SETALL */
};
```

- ✓ The cmd argument specifies one of the above ten commands to be performed on the set specified by semid.

semop

- ✓ The function semop atomically performs an array of operations on a

```
#include <sys/sem.h>
int semop(int semid, struct sembuf semoparray[ ], size_t nops);
                                                    Returns: 0 if OK, -1 on error.
```

semaphore set.

- ✓ The semoparray argument is a pointer to an array of semaphore operations, represented by sembuf structures:

```
struct sembuf
{
  unsigned short sem_num; /* member # in
  set (0, 1, ..., nsems-1) */short sem_op;
```

```
    /* operation (negative, 0, or positive)
    */ short     sem_flg; /* IPC_NOWAIT,
    SEM_UNDO */
};
```

- The nops argument specifies the number of operations (elements) in the array.
- The sem_op element operations are values specifying the amount by which the semaphore value isto be changed.
  - If sem_op is an integer ***greater than zero***, semop adds the value to the corresponding semaphore element value and awakens all processes that are waiting for the element to increase.
  - If sem_op is ***0*** and the semaphore element value is not 0, semop blocks the calling process (waiting for 0) and increments the count of processes waiting for a zero value of that element.
  - If sem_op is a ***negative*** number, semop adds the sem_op value to the corresponding semaphore element value provided that the result would not be negative. If the operation would make the element value negative, semop blocks the process on the event that the semaphore element value increases. If the resulting value is 0, semop wakes the processes waiting for 0.

7.16 Client-Server Properties

- Following are the some of the properties of client - server architecture :

1. Clients and servers are separate processes.
2. They may run on the same or different machines.
3. Each process can hide internal information.
4. Each process can implement its own set of business rules.
5. They communicate by peer to peer protocols.

- The simplest example of this type is client's "fork" and "exec" of server. Two half-duplex pipes can be created before the fork to allow data to be transferred in both directions. Open server is created using this type of arrangement.
- **With FIFOs,** an individual per client FIFO is also required if the server is to send data back to the client. If the client server application sends data only from the client to the server, a single well-known FIFO suffices.
- Multiple possibilities exist with **message queues.**

1. A single queue can be used between the server and all the clients, using the type field of each message to indicate the message recipient.
2. An individual message queue can be used for each client. Before sending the first request to a server, each client creates its own message queue with a key of IPC_PRIVATE. The server also has its own queue, with a key or identifier known to all clients.

- One problem with this technique is that each client-specific queue usually has only a single message on it : A request for the server or a response for a client. Either of these two techniques using message queues can be implemented using shared memory segments and a synchronization method.

7.17 Stream Pipes

VTU : July-14, 17, Jan.-15, 17, 18

- A stream pipe is a bidirectional pipe. For bidirectional data flow between a parent and child, only a single stream pipe is required.
  Fig. 7.17.1 shows stream pipe.

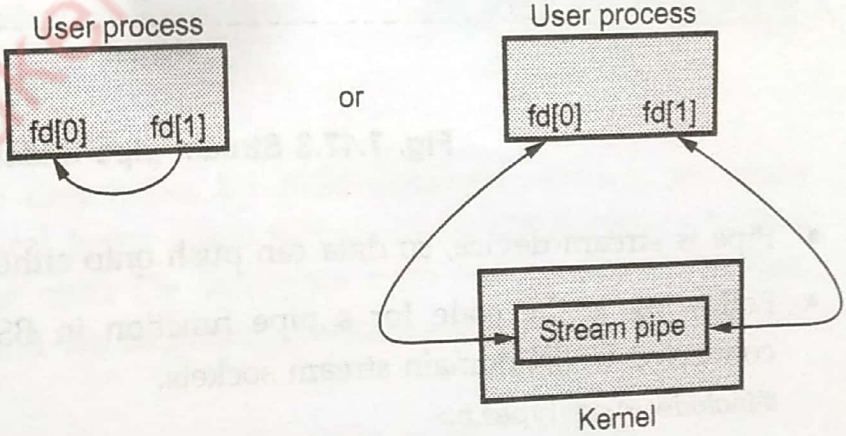


Fig. 7.17.1

- Stream pipe is full duplex. The fd[0] is used by parent process and the child uses only fd[1]. Each end of the stream pipe is full duplex, the parent reads and writes fd[0] and the child duplicates fd[1] to both standard input and standard output. Fig. 7.17.2 shows the

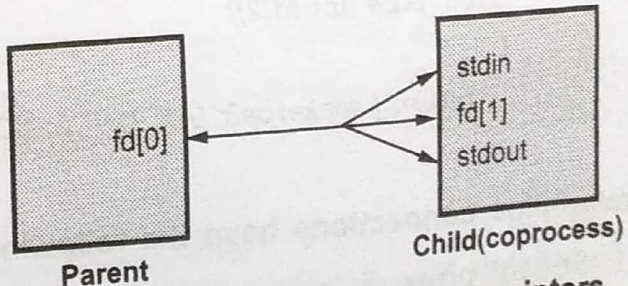


Fig. 7.17.2 Arrangement of descriptors for processes

- The function s_pipe is similar to the standard pipe function. It takes the same argument as pipe, but the returned descriptors are open for reading and writing.
- s_pipe function under SVR4 :

```
#include  "ourhdr.h"
int s_pipe (int fd[2])
{
   return ( pipe (fd) );
}
```

- Fig. 7.17.3 shows the stream pipe under SVR4.

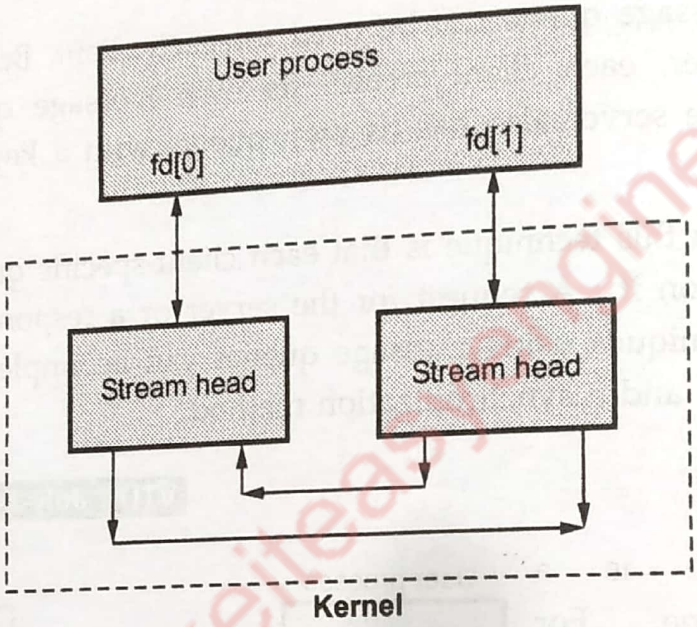


Fig. 7.17.3 Stream pipe under SVR4

- Pipe is stream device, so data can push onto either end of the pipe.
- Following is the code for s_pipe function in BSD version. It creates a pair of connected UNIX domain stream sockets.

```
#include  <sys/types.h>
#include <sys/socket.h>
int s_pipe (int fd[2])
{
   return ( socketpair (AF_UNIX , SOCK_STREAM, 0, fd );
}
```

Stream-pipe connections have the following advantages :

1. Stream pipes do not pose the security risk of being overwritten or read by other programs that explicitly access the same portion of shared memory.

2. Stream-pipe connections allow distributed transactions between database servers that are on the same computer.

Stream-pipe connections have the following disadvantages :

1. Stream-pipe connections might be slower than shared-memory connections on some computers.
2. Stream pipes are not available on all platforms.
3. When you use stream pipes for client/server communications, the hostname entry is ignored.

University Questions

1. *What are pipes ? Explain the different ways to view a half duplex pipe. Write a program to create a pipe between a parent and its child and to send data down the pipe.* **VTU : July-14, Marks 10**
2. *Explain with a neat diagram, how STREAM PIPES can be used to implement client server model.* **VTU : Jan.-15, Marks 10**
3. *What are pipes ? Write a C/C++ program to send data from parent to child over a pipe.* **VTU : Jan.-17, Marks 10**
4. *Explain STREAMs - based pipes. Write a C function that is used by a server to wait for a client's connect request to arrive.* **VTU : July-17, Marks 10**
5. *Write short notes on any two of the following : Stream pipes* **VTU : Jan.-18, Marks 10**

7.18 Passing File Descriptors

- The normal inheritance of file descriptors by child processes suffices for many purposes, one of the most typical being when servers use child processes to deal with clients. When we pass an open file descriptor from one process to another, we want the passing process and the receiving process to also share the same file table entry.
- Fig. 7.18.1 shows passing open file from one process to another process.
(See Fig. 7.18.1 on next page)
- When a descriptor is passed from one process to another, sending process closes the descriptor after passing the descriptor. Closing the descriptor by the sender does not really close the file or device, since the descriptor is still considered open by the receiving process.
- In order for a file descriptor to be passed, the sender and receiver must already share a pipe or pump stream for its conveyance. Given a writable file descriptor pd1 for such an open stream and a file descriptor fd1 to be passed, the sender executes the command.

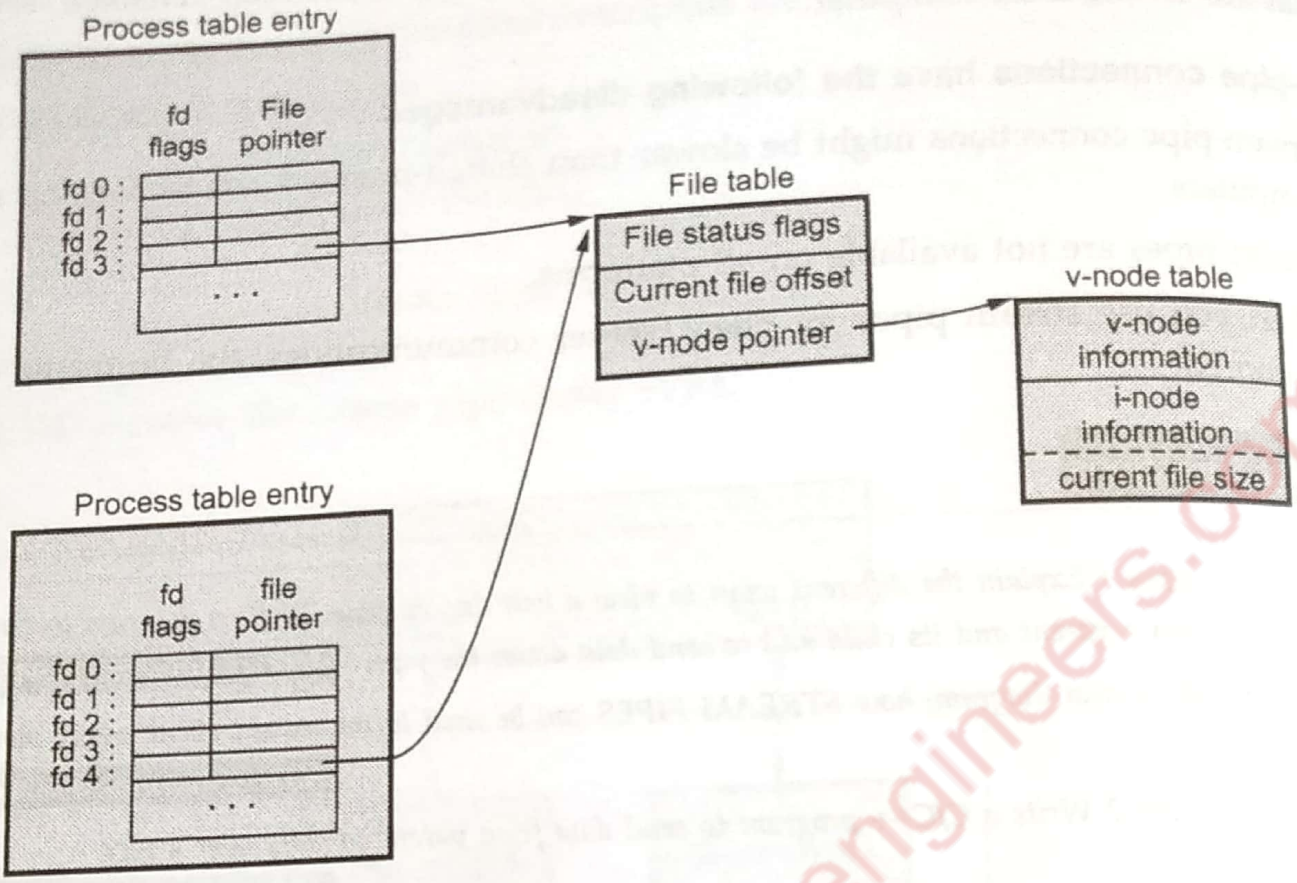


Fig. 7.18.1 Passing an open file

```
#include "ourhdr.h"
int send_fd (int spipefd,  int filedes);
int send_err (int spipefd,  int status,  const char *errmsg);
int recv_fd (int spipefd, ssize_t (*userfun ) (int, const viod *, size_t));
```

- When a process wants to pass a descriptor to another process it calls either send_fd or send_err. The process waiting to receive the descriptor calls recv_fd.
- The function send_err calls the send_fd function after writing the error message to the s_pipe.

```
int send_err(int fd, int errcode, const char *msg)
{
   int    n;
   if ((n = strlen(msg)) > 0)
      if (writen(fd, msg, n) != n)    /* send the error message */
         return(-1);
   if (errcode >= 0)
      errcode = -1;   /* must be negative */
   if (send_fd(fd, errcode) < 0)
      return(-1);
   return(0);
}
```

- File descriptors are exchanged using *ioctl* command *I_SENDFD* and *I_RECVED* in SRV4 on stream pipe. Third argument is used for send descriptor. Following is the program for these purposes.
- The send_fd function for STREAMS pipes :

```
#include "apue.h"
#include <stropts.h>

int  send_fd(int fd, int fd_to_send)
{
   char   buf[2];    /* send_fd()/recv_fd( ) 2-byte protocol */
     buf[0] = 0;        /* null byte flag to recv_fd( ) */
     if (fd_to_send < 0)
      {
        buf[1] = -fd_to_send;   /* nonzero status means error */
         if (buf[1] == 0)
           buf[1] = 1; /* -256, etc. would screw up protocol */
      }
       else
      {
           buf[1] = 0;     /* zero status means OK */
      }

        if (write(fd, buf, 2) != 2)
          return(-1);
        if (fd_to_send >= 0)
          if (ioctl(fd, I_SENDFD, fd_to_send) < 0)
            return(-1);
           return(0);
}
```

The strrecvfd structure :

```
struct strrecvfd
 {
    int    fd;       /* new descriptor */
    uid_t  uid;      /* effective user ID of sender */
    gid_t  gid;      /* effective group ID of sender */
    char   fill[8];
 };
```

- The *recv_fd* function reads the STREAMS pipe until the first byte of the 2-byte protocol is received. When we issue the *I_RECVFD* ioctl command, the next message on the stream head's read queue must be a descriptor from an I_SENDFD call, or we get an error.

7.19 An Open Server-Version 1

- To develop a open server program i.e. a program that is executed by a process to open one or more files. But instead of sending the contents of the file back to the calling process, the server sends back an open file descriptor.
- Following is the advantages for designing separate program on server are as follows :

1. The server can easily be contacted by any client, similar to the client calling a library function.
2. If we need to change the server, only a single program is affected.
3. The server can be a set-user-ID program, providing it with additional permissions that the client does not have.

- The client process creates an s_pipe and then calls fork and exec to invoke the server. The client sends requests across the s_pipe, and the server sends back responses across the s_pipe. Following are the application protocol between the client and the server.

1. The client sends a request of the form "open <pathname> <openmode>\0" across the s-pipe to the server.
2. The server sends back an open descriptor or an error by calling either send_fd or send_err.

The client "main" function, version 1 is as follows :

```
#include   "open.h"
#include   <fcntl.h>

#define BUFFSIZE   8192
int main(int argc, char *argv[ ])
{
   int     n, fd;
   char    buf[BUFFSIZE], line[MAXLINE];

   /* read filename to cat from stdin */
   while (fgets(line, MAXLINE, stdin) != NULL)
     {
```

```
        if (line[strlen(line) - 1] == '\n')
        line[strlen(line) - 1] = 0; /* replace newline with null */

      /* open the file */
        if ((fd = csopen(line, O_RDONLY)) < 0)
          continue;   /* csopen() prints error from server */

      /* and cat to stdout */
        while ((n = read(fd, buf, BUFFSIZE)) > 0)
         if (write(STDOUT_FILENO, buf, n) != n)
          err_sys("write error");
          if (n < 0)
          err_sys("read error");
         close(fd);
        }
      exit(0);
     }
```

The server main function, version 1

```
#include   "opend.h"

char    errmsg[MAXLINE];
int     oflag;
char    *pathname;

int main(void)
{
  int    nread;
  char   buf[MAXLINE];

  for ( ; ; )
    {  /* read arg buffer from client, process request */
       if ((nread = read(STDIN_FILENO, buf, MAXLINE)) < 0)
       err_sys("read error on stream pipe");
     else if (nread == 0)
       break;     /* client has closed the stream pipe */
     request(buf, nread, STDOUT_FILENO);
    }
  exit(0);
}
```

7.20 Client-Server Connection Functions

VTU : Jan.-16, 17

- Here we will discuss three functions that can be used by a client server to establish the functions per client connections.

```
#include "ourhdr.h"
int serv_listen (const char *name);
```

- Server listens for client connection on a well known name by calling *serv_listen*. Here name is the well known name of the server. When client wants to connect with server then client used this name. The return value is the file descriptor for the server's end of the named stream pipe.
- Once a server has called serv_listen, it calls serv_accept to wait for a client connection to arrive.

```
#include "ourhdr.h"
int serv_accept (int listenfd, uid_t  *uidptr);
```

where *listenfd* is a descriptor from *serv_listen*.

- When the client does connect to the server, a new stream pipe is automatically created, and the new descriptor is returned as the value of the function. The effective user ID of the client is stored through the pointer *uidptr*.
- A client just calls *cli_conn* to connect to a server.

```
#include "ourhdr.h"
int cli_conn (const char *name);
```

- where name specified by the client must be the same name that was advertised by the server's call to *serv_listen*.
- All above three functions are used for writing server daemons to handle multiple clients. The number of descriptor available to a single process is only limit to handle the multiple clients. Server requires one descriptor for each client connection. The client-server connections are all stream pipes, open descriptors can be passed across the connections.

7.20.1 System V Release 4

- SVR4 provides mounted streams and a streams processing module named *connld* that we can use to provide a named stream pipe with unique connections for the server. Initially server creates an unnamed stream pipe and pushes the stream processing module *connld* on one end. Fig. 7.20.1 shows SVR4 pipe after pushing "connld".
- SVR4 uses *fattach* function to attach a pathname to the end of the pipe that has the *connld* pushed onto it.

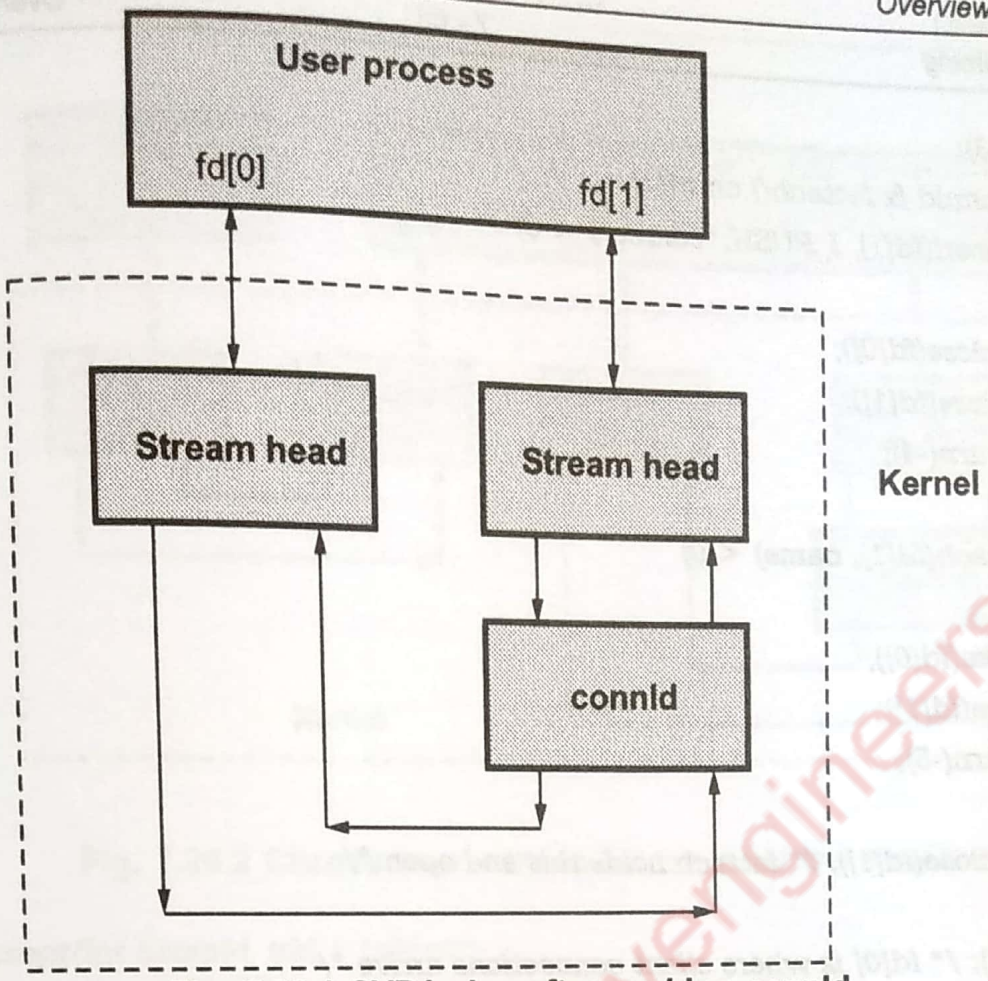


Fig. 7.20.1 SVR4 pipe after pushing connld

- The serv_listen function using STREAMS pipes is as follows :

```
#include "apue.h"
#include <fcntl.h>
#include <stropts.h>
/* pipe permissions: user rw, group rw, others rw */
#define FIFO_MODE (S_IRUSR|S_IWUSR|S_IRGRP|S_IWGRP|S_IROTH|S_IWOTH)

  /* Establish an endpoint to listen for connect requests.*/
int serv_listen(const char *name)
 {
    int    tempfd;
    int    fd[2];

  /*  Create a file: mount point for fattach( ).    */
  unlink(name);
  if ((tempfd = creat(name, FIFO_MODE)) < 0)
     return(-1);
  if (close(tempfd) < 0)
     return(-2);
  if (pipe(fd) < 0)
```

```
        return(-3);
    /*  Push connld & fattach() on fd[1].    */
            if (ioctl(fd[1], I_PUSH, "connld") < 0)
              {
               close(fd[0]);
              close(fd[1]);
             return(-4);
            }
          if (fattach(fd[1], name) < 0)
           {
            close(fd[0]);
           close(fd[1]);
           return(-5);
          }
              close(fd[1]); /* fattach holds this end open */

    return(fd[0]); /* fd[0] is where client connections arrive */
}
```

- When another process call a open for the named end of the pipe, the following occurs :

1. A new pipe is created.
2. One descriptor for new pipe is passed back to the client as the return value from open.
3. The other descriptor is passed to the server on the other end of the named pipe.

- Fig. 7.20.2 shows client-server connection on a named stream pipe. The pipe between the client and server is created by the open. The file descriptor in the client (fd) is returned by the open. The new file descriptor in the server is received by the server using an *ioctl* of *I_RECVFD* on the descriptor fd[0]. Once the server has pushed *connld* onto fd[1] and attached a name to fd[1], it never specifically uses fd[1] again.
- The server waits for a client connection to arrive by calling the serv_accept function. It shown in the following program.

```
#include    <sys/types.h>
#include    <sys/stat.h>
#include    <stropts.h>
#include    "ourhdr.h"

/* Wait for a client connection to arrive, and accept it.  */
```

Chapter 1: SIGNALS AND DAEMON PROCESSES

1. Signals: The UNIX Kernel Support for Signals
2. Signal Mask
3. Sigaction
4. The SIGCHLD Signal and the waitpid Function
5. The sigsetjmp and siglongjmp Function
6. Kill
7. Alarm
8. Interval Timers
9. POSIX.lb Timers
10. Daemon Processes: Introduction
11. Daemon Characteristics
12. Coding Rules
13. Error Logging
14. Client-Server Model.

- ✓ Signals are software interrupts.
- ✓ Signals provide a way of handling asynchronous events: a user at a terminal typing the interrupt keyto stop a program or the next program in a pipeline terminating prematurely.

Name	Description	Default action
SIGABRT	abnormal termination (`abort`)	terminate+core
SIGALRM	timer expired (`alarm`)	terminate
SIGBUS	hardware fault	terminate+core
SIGCANCEL	threads library internal use	ignore
SIGCHLD	change in status of child	ignore
SIGCONT	continue stopped process	continue/ignore
SIGEMT	hardware fault	terminate+core
SIGFPE	arithmetic exception	terminate+core
SIGFREEZE	checkpoint freeze	ignore
SIGHUP	hangup	terminate
SIGILL	illegal instruction	terminate+core
SIGINFO	status request from keyboard	ignore
SIGINT	terminal interrupt character	terminate
SIGIO	asynchronous I/O	terminate/ignore
SIGIOT	hardware fault	terminate+core
SIGKILL	termination	terminate
SIGLWP	threads library internal use	ignore
SIGPIPE	write to pipe with no readers	terminate

SIGPOLL	pollable event (`poll`)	terminate
SIGPROF	profiling time alarm (`setitimer`)	terminate
SIGPWR	power fail/restart	terminate/ignore
SIGQUIT	terminal quit character	terminate+core
SIGSEGV	invalid memory reference	terminate+core
SIGSTKFLT	coprocessor stack fault	terminate
SIGSTOP	stop	stop process
SIGSYS	invalid system call	terminate+core
SIGTERM	termination	terminate
SIGTHAW	checkpoint thaw	ignore
SIGTRAP	hardware fault	terminate+core
SIGTSTP	terminal stop character	stop process
SIGTTIN	background read from control tty	stop process
SIGTTOU	background write to control tty	stop process
SIGURG	urgent condition (sockets)	ignore
SIGUSR1	user-defined signal	terminate
SIGUSR2	user-defined signal	terminate
SIGVTALRM	virtual time alarm (`setitimer`)	terminate
SIGWAITING	threads library internal use	ignore
SIGWINCH	terminal window size change	ignore
SIGXCPU	CPU limit exceeded (`setrlimit`)	terminate+core/ignore
SIGXFSZ	file size limit exceeded (`setrlimit`)	terminate+core/ignore
SIGXRES	resource control exceeded	Ignore

- When a signal is sent to a process, it is pending on the process to handle it. The process can react to pendingsignals in one of three ways:

- Accept the **default action** of the signal, which for most signals will terminate the process.
- **Ignore** the signal. The signal will be discarded and it has no affect whatsoever on the recipient
- process. Invoke a **user-defined** function. The function is known as a signal handler routine and the signal is said to be***caught*** when this function is called.

1. THE UNIX KERNEL SUPPORT OF SIGNALS

- When a signal is generated for a process, the kernel will set the corresponding signal flag in the process tableslot of the recipient process.
- If the recipient process is asleep, the kernel will awaken the process by scheduling it.
- When the recipient process runs, the kernel will check the process U-area that contains an array of signalhandling specifications.
- If array entry contains a zero value, the process will accept the default action of the signal.

- If array entry contains a 1 value, the process will ignore the signal and kernel will discard it.
- If array entry contains any other value, it is used as the function pointer for a user-defined signal handlerroutine.

2. SIGNAL

- The function prototype of the signal API is:

```
#include <signal.h>
void (*signal(int sig_no, void (*handler)(int)))(int);
```

- The formal arguments of the API are: sig_no is a signal identifier like SIGINT or SIGTERM. The handler argument is the function pointer of a user-defined signal handler function.
- The following example attempts to catch the SIGTERM signal, ignores the SIGINT signal, and accepts the default action of the SIGSEGV signal. The pause API suspends the calling process until it is interrupted by a signal and the corresponding signal handler does a return:

```
#include<iostream.h>
#include<signal.h>
/*signal handler function*/
void catch_sig(int sig_num)
{
      signal (sig_num,catch_sig);

      cout<<"catch_sig:"<<sig_num<<endl;
}


/*main function*/
int main()
{
      signal(SIGTERM,catch_sig);
      signal(SIGINT,SIG_IGN);
      signal(SIGSEGV,SIG_DFL);
      pause( );               /*wait for a signal interruption*/
}
```

- The SIG_IGN specifies a signal is to be ignored, which means that if the signal is generated to the process, it willbe discarded without any interruption of the process.
- The SIG_DFL specifies to accept the default action of a signal.

3. SIGNAL MASK

- A process initially inherits the parent's signal mask when it is created, but any pending signals for the parent process are not passed on. A process may query or set its signal mask via the sigprocmask API:

```
#include <signal.h>
int sigprocmask(int cmd, const sigset_t *new_mask, sigset_t *old_mask);
```

Returns: 0 if OK, 1 on error

- The new_mask argument defines a set of signals to be set or reset in a calling process signal mask, and thecmd argument specifies how the new_mask value is to be used by the API.

- The possible values of cmd and the corresponding use of the new_mask value are:

Cmd value	Meaning
SIG_SETMASK	Overrides the calling process signal mask with the value specified in the new_mask argument.
SIG_BLOCK	Adds the signals specified in the new_mask argument to the calling process signal mask.
SIG_UNBLOCK	Removes the signals specified in the new_mask argument from the calling process signal mask.

- If the actual argument to new_mask argument is a NULL pointer, the cmd argument will be ignored, and the current process signal mask will not be altered.
- If the actual argument to old_mask is a NULL pointer, no previous signal mask will be returned.
- The sigset_t contains a collection of bit flags.

The BSD UNIX and POSIX.1 define a set of API known as sigsetops functions:

```
#include<signal.h>

int sigemptyset (sigset_t* sigmask);
int sigaddset (sigset_t* sigmask, const int sig_num);
int sigdelset (sigset_t* sigmask, const int sig_num);
int sigfillset (sigset_t* sigmask);
int sigismember (const sigset_t* sigmask, const int sig_num);
```

- The sigemptyset API clears all signal flags in the sigmask argument.
- The sigaddset API sets the flag corresponding to the signal_num signal in the sigmask argument. The
- sigdelset API clears the flag corresponding to the signal_num signal in the sigmask argument. The
- sigfillset API sets all the signal flags in the sigmask argument.
  [all the above functions return 0 if OK, -1 on error]
- The sigismember API returns 1 if flag is set, 0 if not set and -1 if the call fails.

The following example checks whether the SIGINT signal is present in a process signal mask and adds it to the mask ifit is not there.

```
#include<stdio.h>
#include<signal.h>
int main()
{
        sigset_t      sigmask;
        sigemptyset(&sigmask);            /*initialise set*/

        if(sigprocmask(0,0,&sigmask)==-1)        /*get current signal mask*/
        {
                perror("sigprocmask");
                exit(1);
        }
        else sigaddset(&sigmask,SIGINT); /*set SIGINT flag*/
        sigdelset(&sigmask, SIGSEGV);            /*clear SIGSEGV flag*/
        if(sigprocmask(SIG_SETMASK,&sigmask,0)==-1)
```

```
            perror("sigprocmask");
}
```

A process can query which signals are pending for it via the sigpending API:

```
#include<signal.h>
int sigpending(sigset_t* sigmask);
```

Returns 0 if OK, -1 if fails.

- The sigpending API can be useful to find out whether one or more signals are pending for a process and to set up special signal handling methods for these signals before the process calls the sigprocmask API to unblock them.

The following example reports to the console whether the SIGTERM signal is pending for the process:

```
#include<iostream.h>
#include<stdio.h>
#include<signal.h>
int main()
{
      sigset_t    sigmask;
      sigemptyset(&sigmask);
      if(sigpending(&sigmask)==-1)
            perror("sigpending");
      else cout << "SIGTERM signal is:"
            << (sigismember(&sigmask,SIGTERM) ? "Set" : "No Set")  << endl;
}
```

In addition to the above, UNIX also supports following APIs for signal mask manipulation:

```
#include<signal.h>

int sighold(int signal_num);
int sigrelse(int signal_num);
int sigignore(int signal_num);
int sigpause(int signal_num);
```

4. SIGACTION

- The sigaction API blocks the signal it is catching allowing a process to specify additional signals to be blocked when the API is handling a signal.

The sigaction API prototype is:

```
#include<signal.h>
int sigaction(int signal_num, struct sigaction* action, struct sigaction* old_action);
```

Returns: 0 if OK, 1 on error

The struct sigaction data type is defined in the <signal.h> header as:

```
struct sigaction
{
       void          (*sa_handler)(int);
       sigset_t      sa_mask;
       int           sa_flag;
}
```

- The sa_handler field can be set to SIG_IGN, SIG_DFL, or a user defined signal handler function.
- The sa_mask field specifies additional signals that process wishes to block when it is handling signo signal.
- The signalno argument designates which signal handling action is defined in the action argument.
- The previous signal handling method for signalno will be returned via the
- oldaction argument if it is not a NULL pointer.
- If action argument is a NULL pointer, the calling process‘s existing signal handling method for signalno will be unchanged.

The following program illustrates the uses of sigaction:

```
#include<iostream.h>
#include<stdio.h>
#include<unistd.h>
#include<signal.h>


void callme(int sig_num)
{
       cout<<"catch signal:"<<sig_num<<endl;
}
int main(int argc, char* argv[])
{
       sigset_t sigmask;
       struct sigaction action,old_action;
       sigemptyset(&sigmask);
       if(sigaddset(&sigmask,SIGTERM)==-1 || sigprocmask(SIG_SETMASK,&sigmask,0)==-1)
              perror("set signal mask");
       sigemptyset(&action.sa_mask);
       sigaddset(&action.sa_mask,SIGSEGV);
       action.sa_handler=callme;
       action.sa_flags=0;
       if(sigaction(SIGINT,&action,&old_action)==-1)
              perror("sigaction");
       pause();
       cout<<argv[0]<<"exists\n";
       return 0;
}
```

5. THE SIGCHLD SIGNAL AND THE waitpid API

- When a child process terminates or stops, the kernel will generate a SIGCHLD signal to its parent process. Depending on how the parent sets up the handling of the SIGCHLD signal, different events may occur:
  - Parent accepts the **default action** of the SIGCHLD signal:
    - SIGCHLD does not terminate the parent process.

- Parent process will be awakened.
- API will return the child's exit status and process ID to the parent.
- Kernel will clear up the Process Table slot allocated for the child process.
- Parent process can call the waitpid API repeatedly to wait for each child it created.

❖ Parent **ignores** the SIGCHLD signal:
- SIGCHLD signal will be discarded.
- Parent will not be disturbed even if it is executing the waitpid system call.
- If the parent calls the waitpid API, the API will suspend the parent until all its child processes haveterminated.
- Child process table slots will be cleared up by the kernel.
- API will return a -1 value to the parent process.

❖ Process **catches** the SIGCHLD signal:
- The signal handler function will be called in the parent process whenever a child processterminates.
- If the SIGCHLD arrives while the parent process is executing the waitpid system call, the waitpid APImay be restarted to collect the child exit status and clear its process table slots.
- Depending on parent setup, the API may be aborted and child process table slot not freed.

6. THE sigsetjmp AND siglongjmp APIs

The function prototypes of the APIs are:

```
#include <setjmp.h>

int sigsetjmp(sigjmp_buf env, int savemask);
int siglongjmp(sigjmp_buf env, int val);
```

- The sigsetjmp and siglongjmp are created to support signal mask processing.
- Specifically, it is implementation-dependent on whether a process signal mask is saved and restored when it invokes the setjmp and longjmp APIs respectively.
- The only difference between these functions and the setjmp and longjmp functions is that sigsetjmp has an additional argument.
- If savemask is nonzero, then sigsetjmp also saves the current signal mask of the process in env. When siglongjmp is called, if the env argument was saved by a call to sigsetjmp with a nonzero savemask, then siglongjmp restores the saved signal mask.
- The siglongjmp API is usually called from user-defined signal handling functions. This is because a process signal mask is modified when a signal handler is called, and siglongjmp should be called to ensure the process signal maskisrestoredproperlywhen"jumpingout"fromasignalhandlingfunction.

The following program illustrates the uses of sigsetjmp and siglongjmp APIs.

```
#include<iostream.h>
#include<stdio.h>
#include<unistd.h>
#include<signal.h>
#include<setjmp.h>


sigjmp_buf   env;
void callme(int sig_num)
{
```

```
        cout<< "catch signal:" <<sig_num <<endl;
        siglongjmp(env,2);
}
int main()
{
        sigset_t      sigmask;
        struct sigaction action,old_action;


        sigemptyset(&sigmask);


        if(sigaddset(&sigmask,SIGTERM)==-1) || sigprocmask(SIG_SETMASK,&sigmask,0)==-1)
                perror("set signal mask");
        sigemptyset(&action.sa_mask);
        sigaddset(&action.sa_mask,SIGSEGV);
        action.sa_handler=(void(*)())callme;
        action.sa_flags=0;
        if(sigaction(SIGINT,&action,&old_action)==-1)
                perror("sigaction");
        if(sigsetjmp(env,1)!=0)
        {
                cerr<<"return from signal interruption";
                return 0;
        }
        else
             cerr<<"return from first time sigsetjmp is called";
             pause();
      }
```

7. KILL

- ✓ A process can send a signal to a related process via the kill API. This is a simple means of inter-processcommunication or control. The function prototype of the API is:

```
#include<signal.h>
int kill(pid_t pid, int signal_num);
```
Returns: 0 on success, -1 on failure.

- ✓ The signal_num argument is the integer value of a signal to be sent to one or more processes designated bypid. The possible values of pid and its use by the kill API are:

pid > 0	The signal is sent to the process whose process ID is pid.
pid == 0	The signal is sent to all processes whose process group ID equals the process group ID of the sender and for which the sender has permission to send the signal.
pid < 0	The signal is sent to all processes whose process group ID equals the absolute value of pid and for which the sender has permission to send the signal.
pid == 1	The signal is sent to all processes on the system for which the sender has permission to send the signal.

The following program illustrates the implementation of the UNIX kill command using the kill API:

```
#include<iostream.h>
#include<stdio.h>
#include<unistd.h>
#include<string.h>
#include<signal.h>
int main(int argc,char** argv)
{
       int pid, sig =
       SIGTERM;
       if(argc==3)
       {
              if(sscanf(argv[1],"%d",&sig)!=1)
              {
                     cerr<<"invalid number:" << argv[1] <<
                     endl; return -1;
              }
              argv++,argc--;
       }
       while(--argc>0)
       if(sscanf(*++argv, "%d", &pid)==1)
       {
          if(kill(pid,sig)==-1)
                 perror("kill");

   }
   else
          cerr<<"invalid pid:" << argv[0] <<endl;
           return 0;

          }

}
```

The UNIX kill command invocation syntax is:

Kill [-<signal_num>] <pid>.....

Where signal_num can be an integer number or the symbolic name of a signal. <pid> is process ID.

8. ALARM

- The alarm API can be called by a process to request the kernel to send the SIGALRM signal after a certain number of real clock seconds. The function prototype of the API is:

```
#include<signal.h>
Unsigned int alarm(unsigned int time_interval);   Returns: 0 or number of seconds until previously set alarm
```

The alarm API can be used to implement the sleep API:

```
#include<signal.h>
#include<stdio.h>
#include<unistd.h>
void wakeup( )
{ ; }

unsigned int sleep (unsigned int timer )
{
       struct sigaction action;
       action.sa_handler=wakeup
       ; action.sa_flags=0;
       sigemptyset(&action.sa_m
       ask);
       if(sigaction(SIGALARM,&a
       ction,0)==-1)
       {
              perror("
              sigactio
              n");
              return -
              1;
       }
       (void)
       alarm
       (timer);
       (void)
       pause( );
       return 0;
 }
```

9. INTERVAL TIMERS

- The interval timer can be used to schedule a process to do some tasks at a fixed time interval, to time the execution of some operations, or to limit the time allowed for the execution of some tasks.

The following program illustrates how to set up a real-time clock interval timer using the alarm API:

```
#include<stdio.h>
#include<unistd.h>
#include<signal.h>

#define INTERVAL 5
void callme(int sig_no)
{
        alarm(INTERVAL);
         /*do scheduled tasks*/
}
int main()
{
        struct sigaction action;
        sigemptyset(&action.sa_ma
        sk);
        action.sa_handler=(void(*
        )( )) callme;
        action.sa_flags=SA_RESTAR
        T;
        if(sigaction(SIGALARM,&ac
        tion,0)==-1)
        {
                perror("
                sigactio
                n");
                return
                1;
        }
        if(alarm(INTERVAL)==-1)
                 perr
        or("alarm")
        ;else
        while(1)
        {
                /*do normal operation*/
        }
        return 0;
}
```

In addition to alarm API, UNIX also invented the setitimer API, which can be used to define up to three differenttypes of timers in a process:

- Real time clock timer
- Timer based on the user time spent by a process
- Timer based on the total user and system times spent by a process

The getitimer API is also defined for users to query the timer values that are set by the setitimer API.The setitimer and getitimer function prototypes are:

```
#include<sys/time.h>

int setitimer(int which, const struct itimerval * val, struct itimerval * old);
int getitimer(int which, struct itimerval * old);
```

The arguments to the above APIs specify which timer to process. Its possible values and the correspondingtimer types are:

ITIMER_REAL	decrements in real time and generates a SIGALRM signal when it expires
ITIMER_VIRTUAL	decrements in virtual time (time used by the process) and generates a SIGVTALRM signal when it expires.
ITIMER_PROF	decrements in virtual time and system time for the process and generates a SIGPROF signal when it expires.

The struct itimerval datatype is defined as:

```
struct itimerval
{
        struct timeval it_value;
                                        /*cur
        rent value*/struct timeval it_interval;
                                        /*
        time interval*/
};
```

Example program:

```
#include<stdio.h>
#include<unistd.h>
#include<signal.h>
#define INTERVAL
5 void callme(int
sig_no)
{
         /*do scheduled tasks*/
}
int main()
{
        struct
        itimerval
        val; struct
        sigaction
        action;

        sigemptyset(&action.sa_ma
        sk);
        action.sa_handler=(void(*
        )( )) callme;
        action.sa_flags=SA_RESTAR
```

```
        T;
        if(sigaction(SIGALARM,&ac
        tion,0)==-1)
        {
               perror("
               sigactio
               n");
               return
               1;
        }
        val.it_interval.tv_sec     =INTERVAL;val.it_interval.tv_usec     =0; val.it_value.tv_sec    =IN
        val.it_value.tv_usec       =0;


        if(setitimer(ITIMER_REAL,
               &val , 0)==-1)
               perror("alarm");


        else while(1)
        {
               /*do normal operation*/
        }
        return 0;
}
```

The setitimer and getitimer APIs return a zero value if they succeed or a -1 value if they fail.

10. POSIX.1b TIMERS

POSIX.1b defines a set of APIs for interval timer manipulations. The POSIX.1b timers are more flexible and powerfulthan are the UNIX timers in the following ways:

- Users may define multiple independent timers per system clock.
- The timer resolution is in nanoseconds.
- Users may specify the signal to be raised when a timer expires.
- The time interval may be specified as either an absolute or a relative time.

The POSIX.1b APIs for timer manipulations are:

```
#include<signal.
h>
#include<time.h>

int timer_create(clockid_t clock, struct sigevent* spec, timer_t* timer_hdrp);
int timer_settime(timer_t timer_hdr, int flag, struct itimerspec* val, struct
itimerspec* old);int timer_gettime(timer_t timer_hdr, struct itimerspec* old);
int timer_getoverrun(timer_t
```

DAEMON PROCESSES

11. INTRODUCTION

- Daemons are processes that live for a long time. They are often started when the system is bootstrapped andterminate only when the system is shut down.

12. DAEMON CHARACTERISTICS

The characteristics of daemons are:

- Daemons run in background.
- Daemons have super-user privilege.
- Daemons don't have controlling terminal.
- Daemons are session and group leaders.

13. CODING RULES

- **Call umask to set the file mode creation mask to 0**. The file mode creation mask that's inherited could be set to deny certain permissions. If the daemon process is going to create files, it may want to set specific permissions.
- **Call fork and have the parent exit**. This does several things. First, if the daemon was started as a simple shell command, having the parent terminate makes the shell think that the command is done. Second, the child inherits the process group ID of the parent but gets a new process ID, so we're guaranteed that the child is not a process group leader.
- **Call setsid to create a new session**. The process (a) becomes a session leader of a new session, (b) becomes the process group leader of a new process group, and (c) has no controlling terminal.
- **Change the current working directory to the root directory**. The current working directory inherited from the parent could be on a mounted file system. Since daemons normally exist until the system is rebooted, if the daemon stays on a mounted file system, that file system cannot be unmounted.
- **Unneeded file descriptors should be closed.** This prevents the daemon from holding open any descriptors that it may have inherited from its parent.
- **Some daemons open file descriptors 0, 1, and 2 to /dev/null so that any library routines that try to read from standard input or write to standard output or standard error will have no effect**. Since the daemon is not associated with a terminal device, there is nowhere for output to be displayed; nor is there anywhere to receive input from an interactive user. Even if the daemon was started from an interactive session, the daemon runs in the background, and the login session can terminate without affecting the daemon. If other users log in on the same terminal device, we wouldn't want output from the daemon showing up on the terminal, and

the users wouldn't expect their input to be read by the daemon.

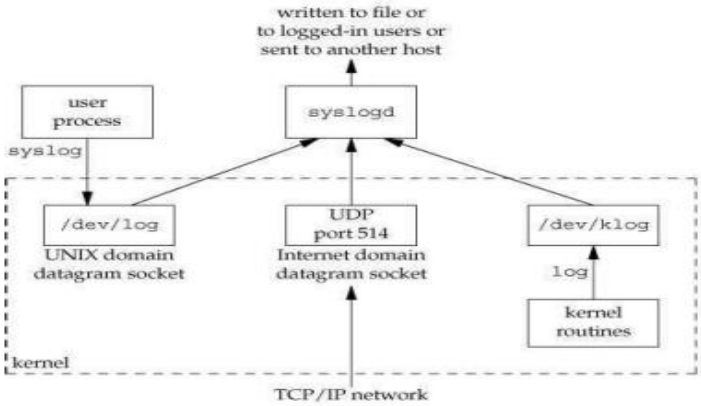
Example Program:

```
#include <unistd,h>
#include
<sys/types.h>
#include
<fcntl.h>

int daemon_initialise( )
{
      pid_t pid;
     if (( pid =
     for() ) < 0)
     return –1;
      else if ( pid != 0)
      exit(0);      /* parent exits */

      /* child
      continue
      s */
      setsid(
      );
      chdir("/
      ");
      umask(0)
      ;
      return 0;
}
```

14. ERROR LOGGING

- ✓ One problem a daemon has is how to handle error messages. It can't simply write to standard error, since it shouldn't have a controlling terminal. We don't want all the daemons writing to the console device, since on many workstations, the console device runs a windowing system. Acentral daemon error-logging facility is required.

Figure 13.2. The BSD syslog facility



There are three ways to generate log messages:

- Kernel routines can call the logfunction. These messages can be read by any user process that opens and reads the /dev/klogdevice.
- Most user processes (daemons) call the syslog(3) function to generate log messages. This causes the message to be sent to the UNIX domain datagram socket /dev/log.
- A user process on this host, or on some other host that is connected to this host by a TCP/IP network, can send log messages to UDP port 514. Note that the syslog function never generates these UDP datagrams: they require explicit network programming by the process generating the log message.

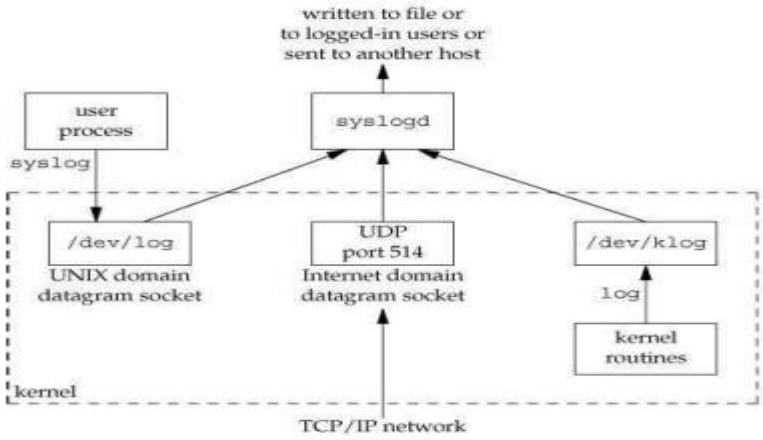
Normally, the syslogd daemon reads all three forms of log messages. On start-up, this daemon reads a configuration file, usually /etc/syslog.conf, which determines where different classes of messages are to be sent. For example, urgent messages can be sent to the system administrator (if logged in) and printed on the console, whereas warnings may be logged to a file. Our interface to this facility is through the syslog function.

```
#include <syslog.h>
void openlog(const char *ident, int option, int facility);
void syslog(int priority, const char *format, ...);
void closelog(void);
int setlogmask(int maskpri);
```

15. CLIENT-SERVER MODEL

In general, a server is a process that waits for a client to contact it, requesting some type of service. In Figure 13.2 [REFER PAGE 10], the service being provided by the syslogd server is the logging of an error message.

Figure 13.2. The BSD syslog facility



In Figure 13.2, the communication between the client and the server is one-way. The client sends its service request to the server; the server sends nothing back to the client. In the upcoming chapters, we'll see numerous examples of two-way communication between a client and a server. The client sends a request to the server, and the server sends a reply back to the client.