Prepared By Dr. N.THENMOZHI M.C.A., M.S., M.Phil., Ph.D.,

Govt. Arts College (Autonomous), Coimbatore-18

Department of Information Technology

OPEN SOURCE TOOLS (18MIT33C) -----II MSc

UNIT-II: Open Source OS Linux: Linux Basics: Introduction - Kernel/User Mode – Process – Advanced Concept-Scheduling – Personalities- Cloning - Signals - Development with Linux - OSS Installation. Linux shell Commands – Vi Editor - Shell programming: Shell Syntax - Variables – conditions – control structures – functions – commands – command execution.

TEXT BOOKS

1. M.N. Rao, "Fundamentals of open source software", PHI Learning Private Limited, 2015.
2. Neil Matthew and Richard Stones, "Beginning Linux Programming", 4th Edition, WROX, 2011.

REFERENCE BOOKS

1. Yashavant P. Kanetkar," Unix Shell Programming", BPB publications, 2003. *2018-2019*

UNIT-II: Open Source OS Linux: Linux Basics: Introduction - Kernel/User Mode – Process – Advanced Concept-Scheduling – Personalities- Cloning - Signals - Development with Linux - OSS Installation. Linux shell Commands – Vi Editor - Shell programming: Shell Syntax - Variables – conditions – control structures – functions – commands – command execution.

2. Open Source OS Linux : Linux Basics

2.1 INTRODUCTION

Linux is an operating system that enables the computer operator and applications to perform the required functions by accessing the devices on the computer. In 1969, a team of developer started working in the Bell Labs laboratories, to find out a solution for a software issue, which had been raised in advanced stage of computers. So, a new operating system was developed in C language which is simple, elegant and able to reuse the code. The developers at Bell Labs named this as UNIX.

Earlier, all commercially available computer systems were written in a code particularly developed for one system. UNIX includes a special code called kernel. By adapting kernel to specific system, it acts as a base of the UNIX system. The operating system and all other required functions were built around this kernel and are written using higher programming language C. This language was specifically developed for creating the UNIX system. By this, we can say that the development of an operating system that runs on different types of hardware is easy.

In the beginning of 90's, Linus Torvalds, studied computer science at the University of Helsinki, at that time, he thought, that it would be nice if a freely accessible academic version of UNIX exists, and finally started to code for that idea. On August 25, 1991, he declared to a newsgroup comp.os.minix, that he has developed an operating system which may include the features that are not in minix, as his operating system resembles it to some extent. The minix can be referred as a variant of the UNIX operating system. He used that as a guideline for the operating system that he wanted to develop and wanted to run on x86-based consumer PCs of the day. General Public Licence (GNU), can be referred to the set of GNU tools which was first put together by Richard Stallman in 1983.

A kernel built by Torvalds became the core of the Linux operating system. The GNU tools interfacing with the Torvalds kernel is treated as the beginning of the Linux Operating System. Linux became popular among UNIX developers because of its portability feature to many platforms, and also due to its free software licence. IBM's 2000 decided to invest $2 billions for the development and sales of Linux. This represents a positive result to the growth of Linux. Governments around the world are deploying and using Linux for its security, flexibility and also to save money and time.

2.1.1 Setting Up Environment

One can execute applications by setting up the following Linux environment in the system. Following are the requirements to install Linux on PC:

- Ubuntu Boot-able CD
- x86 PC with 512 Mb RAM and minimum 4 GB hard disk space

Following is the step by step procedure to install Linux on PC:
Step 1: Select the Boot options to boot from CD ROM and place the Boot-able CD.
Step 2: Ubuntu boots the live CD and displays on the screen 'Ubuntu Version Number'.
Step 3: To install select 'Install Ubuntu' option which is displayed on the screen.
Step 4: It will display 'Preparing to Install Ubuntu', then press 'continue' option.
Step 5: It will display the message 'Installation Type' on the screen, choose the required one and press 'continue'.
Step 6: It requests about Zone and GMT, then choose it and press 'continue'.
Step 7: It displays a message to select keyboard layout, choose English (US Keyboard) and press 'continue'.
Step 8: As Linux is a client-server architecture, you have to give username, password and press 'continue'.
Step 9: Now, it will display the message 'copying files' on the screen, mentioning that the process is going on.
Step 10: It will customise Ubuntu and update the software directly from the network. It finishes with configuring users.

Parts of the Linux Operating System

The following are the parts of the Linux operating system:

- Kernel: It is the heart of the operating system which allocates time and memory to the programs, and it also handles the storage and communication in response to the system calls. It is responsible for providing the required abstraction to hide low level hardware details to the system or application programs. The Linux kernel is unique and flexible for its modularity. If one module of the kernel code fails, the remaining modules of the kernel will not get disturbed.
- Operating system: Developers can write applications that talk to the kernel by using special tools, such as command lines and compilers. This set of tools including kernel will be known as an operating system.
- Environments: Linux includes a lot of choices for users to decide which desktop environment and windowing system can be used. This is difficult to do in OSX and not possible in windows.
- Applications: There exist two kinds of applications in the operating system, namely, the applications that are vital parts of the operating system, and the applications that the users will install later. The developers who develop the applications using Linux will have a wide range of choices providing more flexibility to build that application.

Advantages of Linux

- Linux is portable and scalable.
- It is secure and versatile.
- It is free of cost and provides freedom to control every aspect of the operating system.
- It is a true multi-user system.
- The Linux operating system and the related applications have very short debug-times.

- It supports a wide range of hardware.
- It is free from viruses, malware, trojans, etc.,
- It is relatively stable and flexible.

2.1.2 Simple C File–C Compilation

Now, let us start developing programs for UNIX with C by writing, compiling and running the first UNIX program. Let us consider an example:

Example 2.1 Source code for hello.c file

```
#include <stdio.h>
int main()
{
    printf("Hello World\n");
    exit(0);
}
```

- The text editor is used to enter the above program. There exist many to choose from a Linux system. vi is the popular text editor among many users. Emacs is learned with the help of the following steps:

    - After starting it, press Ctrl-H followed by t for tutorial purpose.
    - The entire manual is available online.
    - For information, press Ctrl-H and then i . Menus for accessing the manual and tutorial are available for some versions of Emacs.
    - The C compiler is called c89 on POSIX compliant systems. Historically, the C compiler is simply called cc .

Over several years, different vendors have been providing UNIX-like systems, with C compilers and with various facilities and options. Then also, it is still called cc.

When the POSIX standard got prepared, it was tough at that time to define a common cc command with which most of the vendors would be compatible. So, as an alternative, the committee has decided to define a new common command for the C compiler, c89. It will take the same options which are independent of the machine.

The commands gcc, cc and c89 refer to the system C compiler, usually gcc or GNU C compiler on Linux systems. On UNIX systems, the C compiler is usually called as cc.

GNU C is provided with Linux distributions, and it also supports the American National Standards Institute (ANSI) standard syntax for C. You can easily obtain and install GNU C from the website 'http://www.gnu.org', if you have a UNIX system without GNU C.

Example 2.2 Hello World program

$cc–o hello hello.c
$./hello

The output generated by this program is:

Hello World

How It Works?

The system C compiler is invoked which translates the source code into an executable file 'hello'. The system C compiler is invoked which translates the source code into an executable file 'hello'. If it did not work well, make sure whether, the C compiler is installed in the correct way or not. Red Hat Linux is having an install option called C Development which you can select. The 'hello' program is usually in the home directory. If PATH does not include a reference to the home directory, the shell will be unable to find hello. Suppose, if any of the existing directories in the PATH contains other program with the same name hello, this will be executed instead. This situation will occur if such a directory is specified in PATH before the home directory.

This problem will be solved by prefixing the file name with ./(eg: ./hello). This clearly initiates the shell to run the program in the present directory with the specified name. If you forget to place the −o name option which indicates the compiler where to place the executable, then the compiler will place that particular program in a.out file (meaning assembler output).

How to Compile C-Programs Distributed in Multiple Files?

Developing project involves many modules. A module can be an object file or a library or a set of source codes. In OSS, the module is distributed in source code format with utilities to compile and build into software. So, let us start with compiling multiple C-Programs.

Take Hello World Example given in Example 2.2 and distribute it in multiple C-Files.

Example 2.3 Hello world example and distribution in multiple C-Files
$ vi Main.c

```
#include <stdio.h>
main()
{
        helloworld();
 }
```

$ vi helloworld.c

```
 #include <stdio.h>
 helloworld()
{
        printf("\nHello World\n");
}
```

In Example 2.3, the main() function calls the function 'helloworld()' which is defined in file 'helloworld.c'. We should compile the above two files, to get the executable file, because Main.c depends on 'helloworld.c' for its function definition.

```
$ gcc Main.c helloworld.c -o hello
```

This command line compilation generates Executable and Linker Format (ELF) file 'hello' which can be executed as follows:

```
$ ./hello
Hello World
```

2.2 KERNEL/USER MODE

There are two CPU operation modes in Linux, namely, the kernel mode and the user mode. Kernel component code can be viewed as a single process which gets executed in a single address space and does not require any context switch. This will be executed in kernel mode which is very efficient and fast. In kernel mode, it is possible to execute any CPU instruction, and can refer to any memory address because the executing code has no restriction to access the underlying hardware.

User programs and other system programs executes in user mode which has no direct access to hardware and reference memory. Most of the code running on a system will get executed in user mode.

2.2.1 Basic Commands

A Linux command is simply a program which has been written to perform some particular actions and that program has a related name. All Linux commands are almost cryptic, meaning that 'ls' stands for listing, 'ln' stands for link, 'du' stands for disk usage, etc. All commands will be in lower case letters. The commands in Linux may take zero, one or more arguments. There are basically two types of commands namely, internal commands and external commands. An internal command is a command which does not have an independent existence. These commands are also called built-in commands . Examples of internal commands are cd, mkdir, etc. An external command is a command which has an independent existence in the way of a different file. Examples of external commands are cat, ls, etc., because programs regarding these commands exist independently in a directory.

Files and Directories

These commands help in creating directories and handling files. The files and directory related commands are shown in Table 2.1.

TABLE 2.1 Files and Directory-related Commands

Command	*Description*	*Syntax*
cat	To create a file and also to display contents in a file	$ cat >file1–Creates a file named file1 $ cat file1–Display the information in file1
cd	Changes over to a new directory	$ cd dir1–Will take you to directory dir1
chmod	Indicates change mode. To change the file or directory permissions.	$ chmod 744 file1–assigns rwx to owner, r-- to group, r-- to others (read=4, write=2, execute=1)
cp	Copy the contents of one file into another file	$ cp file1 file2–Copy the contents of file1 into file2
file	Recognises different types of files	$ file *
find	Used to locate files that meet the search criteria	$ find . –name file1 -print
grep	Stands for globally search a regular expression and print it	$ grep linux file1–Display the lines containing the word linux in file1.
head	Prints first ten lines in the specified file (default value is 10)	$ head -12 file1–Prints first 12 lines in file1
ln	Creates links between files	$ ln file1 file2–Creates a link file2 to the file1
ls	Lists the files	$ ls–Displays all the files in the directory
mkdir	Creates a directory	$ mkdir dir1–Creates a directory named dir1
more	Helps in viewing a file page by page	$ more file1–Prints the file1 page by page
mv	To rename the file	$ mv file1 file2–Rename the file1 to file2
pwd	Displays present working directory	$ pwd
rm	It removes the specified file or files supplied to it	$ rm file1–Removes file1
rmdir	Removes the specified directory if it is empty	$rmdir dir1–Removes the directory dir1 (only if it is empty)
tail	Displays last ten lines in a file (default value is 10)	$ tail -13 file1–Displays last 13 lines in file1
touch	Changes the access time of specified file to current time	$ touch -a file1–Updates the access time of file1 to current time

Manipulating Data

The file content can be manipulated with the following commands. Table 2.2 shows commands related to manipulation of data.

TABLE 2.2 DATA MANIPULATION COMMANDS

Command	*Description*
awk	Pattern scanning and processing
cmp	Compare the two specified files byte by byte
comm	Compares two sorted files line by line
cut	Cuts specified number of character or fields from the given file
diff	Displays differences between two files
expand	Convert tabs to spaces
join	Join lines of two specified files on a common field
perl	Data manipulation language
sed	Stream editor for modifying files
sort	To sort the contents of a file
split	Split a file into smaller files
tr	Translate or delete and to squeeze the repeated characters
uniq	Removes duplicate entries in the specified file
wc	Displays words, lines, and characters in the given file
vi	Opens vi text editor
vim	Represents vi improved. Used to modify all kinds of simple text
fmt	Simple text formatter
spell	Displays spelling mistakes in the text
ispell	i stands for interactive. Works same as spell but suggests how to correct those words
emacs	GNU project Emacs
ex, edit	Line editor

Network Communication

The commands given in Table 2.3 help in sending or receiving files from a local UNIX hosts, and are called Network-related commands.

TABLE 2.3 NETWORK-RELATED COMMANDS	
Command	*Description*
ftp	File transfer program
rcp	Remote file copy
rlogin	Remote login to a UNIX host
rsh	Remote shell
tftp	Trivial file transfer protocol
telnet	Uses telnet protocol to connect to any remote computer
ssh	Provides a secured connection between two hosts

Miscellaneous Commands

Table 2.4 shows that miscellaneous commands which help to list or alter information regarding the system.

TABLE 2.4 MISCELLANEOUS COMMANDS	
Command	*Description*
chgrp	Changes the group ownership of given file
chown	To change the owner of given file
date	Displays or sets the system date and time
du	Displays disk space used by files and directories
exit	Quit the system
finger	Outputs information regarding system users
who	Displays all the users who are logged-in
Whoami	Represents the user name related with current user-ID
kill	Sends a signal to a process
echo	Used to print messages

2.3 PROCESS

A process is defined as a computer program under execution. It is a dynamic entity which includes program instructions, program counter, data, all CPU's registers and process stacks containing temporary data, such as saved variables, return addresses and routine parameters.

Linux is a multiprocessing system, where each process can be treated as separate tasks having their own rights and responsibilities. Every individual process executes in its virtual address space, and it is allowed to interact with any other process only through secure, kernel-managed mechanisms. If one process crashes for any reason, it will never become a cause for any other process in the system to crash. Processes in Linux can kill other processes, communicate with other processes, and spawn other processes and much more.

2.3.1 Types of Processes

init process: When the Linux system is booted, the first process to be loaded into memory is called init process . The PID of the init process is one. For all processes, the init process is the super parent. If the parent of any child process is terminated, then the init process will become the new parent for them. The init process has special privileges, so that it cannot be killed. This process will get terminated only when the Linux system is shut down.
Foreground process: It is a process which gives the user an interface, through which the user can interact with the program. The user has to wait to run another process, while one foreground process is under execution. A process running in the foreground will take control over the terminal until it ends its activity. When you run commands on the terminal, they run as a foreground processes.
Background process: The process which run in the background and does not take any user input from the terminal is known as background process . These processes will never block the terminal and allows users to use the terminal irrespective of its completion. To make a process to run in the background, the character '&' should be placed after the process name in the command.
Batch process: Processes which are in the queue, executes one by one in FIFO order are called batch processes . The tasks which come under this category can be executed either at a certain date and time or when the total load on the system is low to accept extra jobs.
Daemons: The server processes that run continuously are known as daemons. These processes run in the background and are independent of terminal and user interaction. To develop a daemon process service, the user has to separate the process from its parent process by killing it. This makes the process free from the terminal and now it is controlled or restricted by its init process which is the grandparent process.
Zombie process: The process which has been completely deallocated, but still exists in the process table is called zombie process. This process results in a process state when the child dies before the parent process. The existence of many zombie processes indicates a software bug.

2.3.2 Types of Identifiers Related to Processes

Process identifier: Every process in Linux is identified by its unique ID. This ID is simply called PID . Pid's are 16-bit mode that are assigned sequentially by Linux whenever new processes are created.
Parent process ID: Every process in the Linux will have a parent except the init process. So, every process includes another identifier in addition to its own ID, called as parent process ID which is simply written as PPID.
User identifier: It is simply written as a UID. This UID gives the information about the user to which a process belongs to.

Group identifier: The group identifier gives the information about the group to which a process belongs to and is simply written as GID.

2.3.3 Process States

A process may be in any of the following states:

Running: A process is said to be in the running state, if it is the currently running process in the system or, if it is ready to run.

Waiting: A process is said to be in waiting state, if it needs some resource for its execution or some event to occur. There are two different kinds of process existing in this state, one is an interruptable waiting process and the other is an uninterruptable waiting process.

Interruptable: Interruptable waiting process will wait for an event to occur or signal from another process.

Uninterruptable: Uninterruptable waiting process will wait for a hardware condition and cannot handle any signal.

Stopped: A process is in stop or halt state once if it is completed. A stopped process can be restarted by another process.

Zombie: A process which is terminated, but still the process-related information is available in the process table comes under this state.

2.3.4 Creation of Process

The below given program shows how to create a process:

```
#include<errno.h>
#include<stdio.h>
#include<unistd.h>
main()
{
pid_t pid;
pid=fork();//This will create a new process
if (pid == −1) //If the process is not created
        perror("fork"); //Printing the error message
else if(pid == 0)
        printf("This is a child process with pid:%d\n",pid);
else
        printf("This is a parent process with pid:%d\n",pid);
}
```

Compiling file process.c
$gcc process.c

Executing the compiled file
$./a.out

The output generated by this program is:

This is a parent process with pid: 5335

This is a child process with pid: 0

fork() function

It is used to create a new process which is a duplicate of existing process from which it is called. The process which calls this function is called as parent process, and the newly created process is called child process. In simple words, a child process is a duplicate copy of the parent process except the following:

- The child process will have a Parent Process ID (PPID) which is same as the PID of the process that created it.
- The child process will have a unique Process Identifier (PID), like all other processes in the system.
- The child process does not have the possibility to inherit any timers from its parent.
- Resource utilization and CPU timers will be reset to zero in the child process.

After the fork() function completes its work, there exist two processes. Each process continues its execution from the position, where fork() returns. Initially, the child process stack, data and heap segments are exact duplicates of the parent's memory, but after the fork(), each process can be able to modify the variables in its stack, data and heap segments without affecting the other process.

Syntax of fork() function:

```
#include <sys/types.h>
#include <unistd.h>
 pid_t fork(void);
```

Example 2.4 Fork() function:

```
#include <sys/types.h>
#include <unistd.h>
int main( int argc, char *argv)
{
        pid_t pid;
        Pid = fork(); /* creating a new child process*/
        if(pid !=0)
        {
                printf(" The id of child process is : %d", pid);
}
}
```

Return Type

The two processes can be distinguished with the help of value returned by fork(). This fork() function returns a process ID of the created child to its parent process. The fork() returns zero for the child process. For some reason, if a new process is not created, then the fork() returns −1.

vfork() Function

It is used for creating new processes without copying the page tables belonging to the parent process. vfork() is faster than a fork(). It does not make any copy of the parent process address space, rather, it will borrow the parent's memory and thread of the control, until a call to exit or execve() is occurred. The execve() executes the program pointed by specified filename. Whenever, a new process is created by using vfork(), the parent process is suspended temporarily and the child process will proceed by borrowing the parent process address space. This will be continued, until the child process calls execve() or exits at which the parent process will be continued.

vfork() function Syntax:

```
#include <sys/types.h>
#include <unistd.h>
pid_t vfork()
```

getpid() and getppid() Functions

getpid(): The child process can acquire its own process ID by using getpid() function.

getppid(): The child process can acquire its parent process ID by using getppid() function.

A process is said to be in waiting state, if it needs some resource for its execution or some event to occur. Let us see syntax and example of wait.

Syntax of wait() function:

```
#include <sys/wait.h>
int wait(status)
```

Example 2.5 Wait Function

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/wait.h>
main()
{
        int statuspr,pid,cpid;
        pid=fork()
        if(pid!=0)
        {
                printf('process id %d",pid);
                 cpid=wait(&statuspr);
                printf("child process is in wait state");
        }
        else
```

```
        {
                printf("processed %d", getpid());
               exit();
        }
        printf("pid %d is going to be terminated",pid);
}
```

Reasons for Failure of Creating a New Process

The following are the possible reasons for the failure of creation of new processes:

- The resource limit on the number of processes permitted to a particular user ID has been exceeded.
- The system wide limit on the number of processes that can be created has been reached.

2.4 ADVANCED CONCEPT-SCHEDULING

Since, Linux includes multitasking, which means that many processes can run as if they were the only process on the system. It is the responsibility of an operating system to choose which process at a given instant can access to a system's CPU(s). This is called scheduling and what controls this is called the scheduler. Scheduling involves the distribution of the resource 'processor' to the competing tasks. The job of the scheduler is to swap CPU access between different processes and also to choose the order in which processes obtain CPU access. The scheduler responsibility is to run the deserving process out of all the runnable processes. Runnable process is a process which waits for CPU to run.

2.4.1 Objectives of Scheduling

Scheduling policy determines when to switch and what process to choose. The following are some of the scheduling objectives:

- To avoid process starvation
- To achieve good throughput for background jobs
- To improve system performance
- To enforce priorities
- To minimize the wasted resources overhead
- To achieve a balance between a response and utilization
- To support for soft real time processes

Processes running for a long time have their priorities decreased, whereas the processes which are waiting for a long time will have their priorities increased dynamically.

2.4.2 Categories of Processes

While learning scheduling, it is necessary to learn about the two categories of processes:

I / O-bound processes: The processes which use I/O devices, and wait for a long time for I/O operations to complete, can be treated as I/O-bound processes.

CPU-bound processes: The processes which require a lot of CPU time, can be treated as CPU-bound processes.

2.4.3 Types of Processes

Interactive processes: The processes which get started and controlled through a terminal session are called interactive processes. These types of processes interact with the users and spend much time waiting for mouse operations and key presses. It is necessary for someone to connect to the system to start these processes because they cannot start automatically as a part of the system functions. Examples: text editors, graphic applications and command shells.

Batch processes: This type of processes does not need any user interaction, so they often run in the same background. Examples: database search engines, programming language compilers, scientific computations and web server.

Real-time processes: This type of processes should never be blocked by lower-priority processes. They should have a short time for response with minimum variance. Examples: programs that collect data from physical sensors, robot controllers, video and sound applications.

2.4.4 Process Priorities

To run a process, the Linux scheduler should consider the priority of every process. Basically, there are two types of priorities:

1. Static priority: The users assign this priority to real-time processes ranging from 1 to 99. It is not possible for the scheduler to change this static priority.

2. Dynamic priority: It is the sum of base priority time and the CPU time left for remaining process. This has to be considered before its quantum expires in the current epoch.

This dynamic priority is applied only to conventional processes.

Note: The static priority is always higher than dynamic priority. When there is no real-time process in running state, then only the scheduler will prefers to run conventional processes.

2.4.5 Scheduling Policies

The following are the scheduling policies in the kernel:

SCHED_FIFO: It is an easy scheduling algorithm without time-slicing. When the scheduler assigns the CPU to any process, it immediately leaves the described process in the current position of the run queue list. If there is no other process with higher priority, then the process continues to use the CPU according to its requirement. If there are processes with the same priority they are runnable.

SCHED_RR: This policy assigns a fixed time-slice per process and cycles through them. This involves poor average response time and no starvation.

SCHED_OTHER: This policy includes a binary flag SCHED_YIELD. When a process invokes or calls the sched_yield() system call, then the binary flag will be set. The scheduler places the descriptor of the process at the bottom of the run queue list.

2.4.6 A Data Structure Used by the Scheduler

Following is the data structure used by the schedular:

```
struct task_struct
 {
long counter;
long nice;
unsigned long policy;
int has_cpu, processor;
unsigned long cpus_allowed;
 unsigned long rt_priority;
}
```

2.4.7 Scheduling in Multiprocessor Systems

A multiprocessor system is a system in which all of the processors will be busy in running or executing processes. In this, the current process of each system exhausts its time-slice by waiting for a system resource, so it runs its scheduler separately. Linux is a Symmetric Multiprocessing (SMP) operating system, which is capable of evenly balancing work between CPUs in the system. In a single processor system, the idle process is the first task in the task vector, whereas in an SMP system, there will be one idle process per CPU.

2.5 PERSONALITIES

Personality refers to set the process execution domain. The general form for personalities is:

```
#include<sys/personality.h>
 int personality(unsigned long persona);
```

Personality makes the execution domain referenced by ‘persona’ the new execution domain of the current process. Linux supports various execution domains or personalities for every process. Execution domains indicate Linux how to map signal numbers into signal actions. On success, the previous persona is returned and on failure, −1 is returned. It is Linux-specific and of Independent Disks (RAID) levels which are supported by the kernel.

2.6 CLONING

It is a process done by calling the system call clone(). The clone() is a system call in Linux which creates a child process that may share parts of its execution with the parent. The child process can share memory space, the list of signal handlers and the list of file descriptors with the parent.

2.6.1 Use of clone()

The advantage of clone() is to implement multithreading, i.e., several threads of control in a program that execute simultaneously in a distributed memory space. The general form to call clone() is:

```
#include<sched.h>
 int clone (int (*fn) (void *), void *child_stack, int flags, void *arg);
```

Whenever, the child process is created by calling this system call clone(), it immediately executes the fn argument which is a pointer to the function called by the child process at the time of execution. This is different from fork(), where the execution continues in the child process from the point of fork() call. When the fn argument returns, the child process will get terminated since the integer returned by the fn argument is the exit code for the child. The child process gets terminated explicitly by calling exit().

The child_stack argument represents the stack location used by the child process. For a child process it is not possible to execute in the same stack which the parent uses because the child and parent will share the same memory. So, a separate memory space is set up by the parent process for the child stack and to this space a pointer is passed to clone(). The child_stack points to the topmost address of the child stack memory space.

The flag argument contains the value inherited from the parent process. The argument arg is passed to the function which returns the Process ID(PID) of the child process or −1 on failure.

2.7 SIGNALS

Signals are defined as software interrupts, and are introduced in Linux to simplify Inter-Process Communication (IPC). In this, a message is sent to a process or a group of processes containing the number identifying the signal. These are only processed when the process is in user mode. If, suppose a signal has been sent to a process which is in kernel mode, it is dealt immediately on returning to user mode. To use the signals in your program, include the header file. Linux kernel implements nearly 30 signals. Signals are not preferred to carry any argument and their names mostly represent self-explanation.

Signals can be generated either synchronously or asynchronously. A synchronous signal is related to a particular task in the program and is generated during that action. Errors generate signals synchronously, and so make explicit requests through a process to produce a signal for that same process. The signals that are produced by events outside the process control are known as asynchronous signals. During execution, asynchronous signals arrive at unexpected times. External events cause signals asynchronously, and so make explicit requests that apply to some other process.

2.7.1 Types of Signals

Every signal name in Linux begins with characters SIG. Some of the signals are as follows:

SIGABRT: When a process calls the function abort, the signal SIGABRT is generated.
SIGALRM: When the timer of alarm functions goes off, the signal SIGALRM is generated.
SIGILL: This signal is sent to a process when it attempts to execute an illegal instruction. This represents that the program's stack is corrupted.
SIGINT: Linux will send this signal to a process when the user tries to end it by pressing Ctrl+C. This way terminates the programs from the terminal.
SIGKILL: This will end a process immediately and cannot be handled.
SIGTERM: This signal is for process termination.
SIGCHLD: Linux will send this signal to a process when a child process exits.

When the signal occurs, it is the responsibility of the process to notify the kernel what to perform with the signal. The following are the three options to dispose the signal:

- The signal may be ignored, meaning that no action will be taken when the signal occurs. All the signals can be ignored except the signals generated by hardware exceptions, like divide by zero.
- If the signal has to be caught, then the process must register a function with the kernel. The related function is called by the kernel when a particular signal occurs.
- Other than the above two options, the third option is the default action when a signal occurs. Every signal is associated with some default action which may be a process termination, ignore, etc.

Please note that, we cannot ignore SIGSTOP and SIGKILL signals, because these signals provide a way for the kernel or root user to stop or kill any process in any circumstances. The default action for these two signals is to terminate the process. These two signals can neither be caught nor be ignored.

2.7.2 Sending Signals

There exist several ways to deliver signals to some program or script. The most common way is that a user has to type CONTROL+C or the key INTERRUPT when the script is executing. The most common method to deliver signals to a program is to use the command 'kill', where the command is as follows:

```
$ kill -signal pid
```

The signal in the above syntax represents the number or the name of the signal and pid represents the process ID to which the signal should be sent.

It is also possible for this 'kill' command to kill more than one process at a time, i.e., by using single 'kill', command, where the command is as follows:

```
$ kill -9 pid1 pid2
```

2.7.3 Receiving Signals

When a signal is received by a process, the default action happens if that process does not include any signal handler. A handler can be set up for a signal by using the system call signal() with prototype:

```
typedef void (*sighandler_t) (int);
sighandler_t signal(int sig, sighandler_t handler);
```

The above setup now becomes a signal handler for the specified signal(sig). Whenever a signal is arrived, the process will be interrupted, then the current registers get saved and the signal handler is invoked.

2.7.4 Blocking Signals

Every process will have a list of currently blocked signals. When a particular signal is blocked, it is not delivered, but remains pending.

The system call sigprocmask() serves to change and examine the list of blocked signals. The system call sigpending() reveals the signals which are in a pending state.
The system call sigsuspend() suspends the calling process until a particular signal is received.

Example 2.6 Signals Program

```
#include<unistd.h>
#include<sys/types.h>
#include<stdio.h>
#include<stdlib.h>
#include <signal.h>
/* This will be our new SIGINT handler. SIGINT is generated when user presses Ctrl-C. Normally, program will exit with Ctrl-C. With our new handler, it won't exit. */

void mysigint()
{
        printf("I caught the SIGINT signal!\n");
         return;
}
 /* Our own SIGKILL handler */
void mysigkill()
{
        printf("I caught the SIGKILL signal!\n");
        return;
}
/* Our own SIGHUP handler */
void mysighup()
{
         printf("I caught the SIGHUP signal!\n");
         return;
}
 / * Our own SIGTERM handler * /
void mysigterm ()
{
        printf("I caught the SIGTERM signal!\n");
        return;
}
int main()
{
/* Use the signal() call to associate our own functions with the SIGINT, SIGHUP, and SIGTERM signals */

        if (signal(SIGINT, mysigint) == SIG_ERR)
                printf("Cannot handle SIGINT!\n");
        if (signal(SIGHUP, mysighup) == SIG_ERR)
```

```
            printf("Cannot handle SIGHUP!\n");
        if (signal(SIGTERM, mysigterm) == SIG_ERR)
            printf("Cannot handle SIGTERM!\n");
        /* can SIGKILL be handled by our own function? */
        if (signal(SIGKILL, mysigkill) == SIG_ERR)
            printf("Cannot handle SIGKILL!\n");
        printf("My PID is %d.\n", getpid());
        while(1);  /* infinite loop */
        /* exit */
         exit(0);
}
```

2.8 DEVELOPMENT WITH LINUX

Even though Linux is a kernel operating system, it has GNU project development tools. The GNU project was established by Richard Stallman in 1983. The tools invented from GNU project are gcc, and gnat which are compliers, Emacs which is a text editor and a wide variety of other tools. All these tools are GNU project outcomes. The tool gcc is used for C, C++, objective C(Object Oriented Programming) compilations.

gcc

The GNU Compiler Collection (gcc) is a command line compiler on Linux systems. It is also C and C++ compiler developed by the GNU project. For example, if one had written a program whose file name is 'add', the compilation procedure is as follows:

- The primary way of compiling that file 'add.c' into an executable file called 'add' is gcc -o add add.c.
- If the program compiles without any errors, it can be executed by typing './add'.

Please note that the assembly code of 'add.c' can be obtained by replacing the '-o add' option with '-S' as 'gcc -S add.c'.

Some of the various options available for gcc and the related descriptions are as follows:
-c: compile, but do not link
-ansi: use the ANSI standardised version of C
-o: file name for the output program file
-wall: to show all warnings
-lm: to link math library

gdb

The gdb will step through the source code line-by-line or instruction-by-instruction. It is also possible to know the value of any variable at run-time.

gnat

It is an acronym for GNU NYU Ada Translator (GNAT). It is a free software compiler for the programming language 'Ada' which is a part of the GNU compiler collection. The gcc

compiler is having the capability of compiling programs written in several languages including Ada95 and C. If the file extension is either '.ads' or '.adb', it assumes that the given program has been written in Ada and, then it will call the GNAT compiler to compile that particular specified file.

Emacs

It is one of the popular text editors used on Linux introduced by Richard stallman. It is written mostly in 'Lisp' programming language. Emacs can be treated as a different point of view, to different people, as it is handled based on the user requirements. Depending on the need you have, you may get any of the responses such as text editors, mail client, coding and debugging, file management, task management, Email reading, web surfing, news reader, word processor, integrated development environment, etc. Emacs possesses a complete built-in programming language which can be utilised to personalise, extend and alter its behaviour. The GNU Emacs include the following features:

- Editing modes for different types of files including HTML, source code and plain text
- A tutorial for new users
- It supports unicode for their scripts It is highly customisable by using graphical user interface

It is easy to obtain Emacs which is an alternative package that you can install from Linux distribution media like RedHat, Debian, Slackware, etc. Otherwise, you can get the source code of Emacs and compile it manually.

In vi editor, the user makes use of k , j , l , h keys to shift up a line, down a line, ahead by a character and back by a character. The user may require, a few hours or weeks of practice, to move a file by using different key groupings existing in vi editor.

Emacs has also various commands and keystrokes. Similar to vi editor, the users of Emacs has also need to learn the basics commands and keystrokes to get work done. Emacs makes use of multi-key groupings for the reason that it is not a modal editor, just like vi. The keys that Emacs use are mostly abbreviated as C for Ctrl or control and M for meta (Instead of M, we can also use either Esc or Alt keys) Both Esc and Alt keys do the same task in most of the standard configurations. When search the online documents for Emacs.

Emacs presents command completion through 'Tab' key. For example, type 'M-x search' and then hit the Tab key. Emacs will affix a hyphen to point out that there are a number of possible completions, but they all include a hyphen as the next character. Emacs is also available in online tutorial to provide the necessary editing functions and features in order to enhance the familiarity among the users. It makes clear how to use the other help functions in Emacs.

Makefile

An executable file can be created from multiple source file by creating another file called Makefile and build it with the help of 'make' command in Linux. The Makefile includes all the required information that is needed for compiling. The file is represented as Makefile or makefile. Makefiles are special format files, which together with 'make' command will enable you to automatically build and manage your programs or projects. The sample look of the Makefile is as follows:

```
sampleexecfile : sourcefile1.o sourcefile2.o
```

```
gcc-o sampleexecfile sourcefile1.o sourcefile2.o

sourcefile1.o : sourcefile1.c
gcc-c sourcefile1.c

sourcefile2.o : sourcefile2.c
gcc-c sourcefile2.c
```

Options of 'Make' Command

-f : The described makefile will be used as description file
-i : Ignore if any errors
-h : It displays the help information