Linux

Prepared By: Lok Prakash Pandey

Lab 1

Prepared By: Lok Prakash Pandey

Introduction

- The Unix operating system was developed at AT&T Bell Laboratories by Ken Thompson and Dennis Ritchie in 1969.
- Later source code of kernel of Unix was provided for academic and research purposes in licensed form.
- The basic idea of the kernel was the same, however different organizations gave different names to their modified Unix product.
  - IBM-launched Unix → AIX
  - Sun Microsystems → Sun OS (Solaris)
  - HP → Ux
- Andrew S. Tanenbaum made Unix kernel for academic purposes and added certain features into it and named it **MINIX** (1987).

Introduction…

- Linus Torvalds, at the University of Helsinki in Finland, in 1991, made a notice in a Usenet posting to the newsgroup "comp.os.minix", and he demanded feedback for his newly developed kernel (Linux kernel). His main points were:
  - Free kernel
  - Not large like GNU project
  - MINIX feedback (likes/dislikes on MINIX)
- ***Prof. Richard Stallman***: He founded ***FSF,*** *Free Software Foundation*. His concept was software should be free (i.e. if a vendor wants to get paid for his product, it is OK, but the buyer should have freedom in whatever he/she does with the software like he/she may install it on multiple machines or modify the software). One of the work of ***FSF*** was the ***GNU project***. The objective of ***GNU project*** was to make a complete Unix like operating system for free. (**GNU → Is Not Unix**).
- But before GNU project, Torvalds made the Linux kernel. Later, Linux was handed over to the GNU project and that is why we say ***GNU/Linux***.

Introduction…

- ***GPL*:** GNU provided license is called GPL (General Public License).
- **Open source:** That program whose source code is available is called open source and whose is not available is called closed source.
- **OpenOffice.org:** Office package (word processor, spreadsheet, presentation, etc) for Linux by Sun Microsystems Inc.
- **Linux Distributions:**
  - **RedHat (After RedHat Linux version 9, it was not released as version 10, instead released as Fedora 1, Fedora 2, …, Fedora 16)**
  - **Mandrake**
  - **Ubuntu**
  - **Suse (Novell)**

Installing Linux

1. To completely install Linux (Fedora 16) on your machine, free a logical partition on your computer running Windows XP/7 by going to:
   - Control Panel
   - Administrative Tools
   - Computer Management
   - Disk Management
   - (Delete Logical Partition)
   - Now run the installation CD by restarting the machine

   Note: After booting Linux installation is initiated. Linux installation is a wizard and if we provide correct information, Linux will be installed.

2. To install Linux on a Virtual Machine, first install a VMware and then install Linux using its ***iso*** image.

Using the Visual Editor

- An editor is a program that is used to edit source code. There are many text editors available for Linux, but the two most widely used are Visual Editor Improved (VIM) and Emacs. Here we discuss VIM editor. To use VIM editor:
  - Click on Applications
  - Click on System Tools
  - Click on Terminal

Creating and opening file

- $ vi <filename> - opens file if it is already exist, otherwise creates new file.
- $ vi - opens VIM editor, without filename.

Deleting file

- $ rm <filename> - deletes file.

VIM Modes

- The working environment of VIM is defined in two modes: ***Command Mode*** (***Normal Mode***) and ***Insert Mode***.
- In ***Command Mode***, VIM interprets everything typed as a command to VIM.
- To insert the text, the VIM must be in ***Insert Mode***.
- To handle the file such as saving/quitting from file, VIM must be in ***Command Mode***.
- Initially VIM will always be in ***Command Mode*** (**default**).

Adding text to the file

- There are two basic commands to enter to the Insert Mode.
  - a - opens Insert mode and enables to insert text after the cursor.
  - i - opens Inserts mode and enables to insert text before the cursor.
- To return to the Normal mode press Esc key.

- ***Insert Mode Commands***
  - a - Append text after the cursor and enter Insert mode.
  - A - Append text at the end of line and enter insert mode.
  - i - Insert text before the cursor and enter insert mode.
  - I - Insert text at the beginning of the line and enter insert mode.
  - o - Open a line below the cursor and enter the insert mode.
  - O - Open a line above the cursor and enter insert mode.

Save Changes and Exit the Visual Editor

- ***Normal Mode operation***
  - :w - write (save) the file.
  - :q - quit if saved.
  - :wq - save and quit. Or :x
  - :w <filename> - save file with filename.
  - :q! - Quit without saving any changes.
  - :e! - Abandon the changes and reload the last saved file.

For help on vim: *$ vimtutor*

Tasks

1. Practice each and every command mentioned above.
2. Write the C source file for execute ***"Hello World"***, and save it as filename '***hello.c***' in your directory.
3. Write the C source file for calculating interest, and save it as file name '***calcinterest.c***' in your directory.
4. Close **vim** editor and reopen each of above file and enjoy!

Compiling with GCC

- A compiler turns human-readable code into machine readable object code that can actually run. The compilers of choice on Linux systems are all part of the GNU compiler collection, usually known as **gcc**. '***gcc***' supports the ANSI standard syntax for C.

- ***Compile, link and run the program***

  $ gcc -o <exefile> <sourcefile>

  $ ./<exefile>

  Example: **$ gcc -o hello hello.c**

  $ ./hello

  Note: If you compile without <exefile>, ***gcc*** saves the executable file as '***a.out***' file.

  $ gcc <sourcefile>

  $ ./a.out - to execute.

- ***Compiling a single source file***

  $ gcc -c <sourcefile>

  Example: $ gcc -c hello.c

  The resulting object file is named '***hello.o***'.

Task

- ***Compile and run the C programs you have created.***

Getting familiar with Linux commands

- # denotes root user (*Like Administrator in Windows*).
- $ denotes normal user.
- ***To clear the screen*: $ clear**
- ***To add users*:** Only root user can add other users.
  - ***# useradd username***
  - ***# passwd username***
- ***To print working directory*: $ pwd**

Note: Generally, the working directory is '/home/user'.

Getting familiar with Linux commands...

- ***To create a directory*: $ mkdir <directory_name>**

  Note: Space means argument separator.

  $ mkdir dir1 dir2 (Two directories are created)

  So, to create a directory as BSc CSIT do:

  $ mkdir 'BSc CSIT'

- ***To create a zero-length file*: $ touch <filename>**
- ***To list the contents of directory*: $ ls (Blue color for directory)**
- ***To create file*: $ cat>filename**

  … … … … <Enter>

  Ctrl+D

- ***To open file*: $ cat filename**
- ***To remove empty directory*: $ rmdir <directory_name>**
- ***To remove directory and its contents*: $ rm –r <directory_name>**
- ***To check which user you are:* $ whoami**
- ***To reboot:* $ reboot**

Getting familiar with Linux commands…

- / - means root directory
- . – means current directory
- .. – means parent directory
- ~ - means home directory

Example:

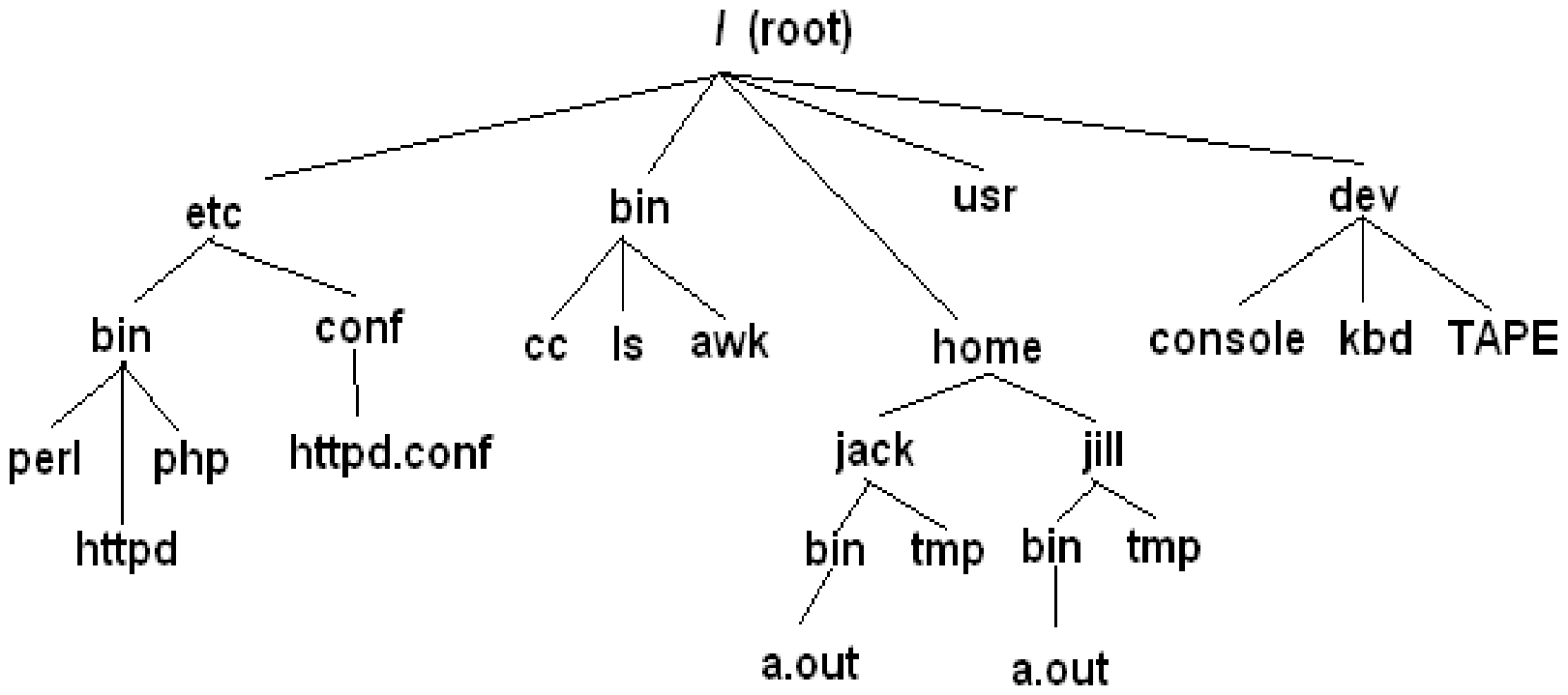
$ cd / (Goes to Root Directory)

$ cd ~ (Goes to Home Directory)

$ cd (Goes to Home Directory)

$ cd .. (Goes to Parent Directory i.e. One step back)

Linux directory structure



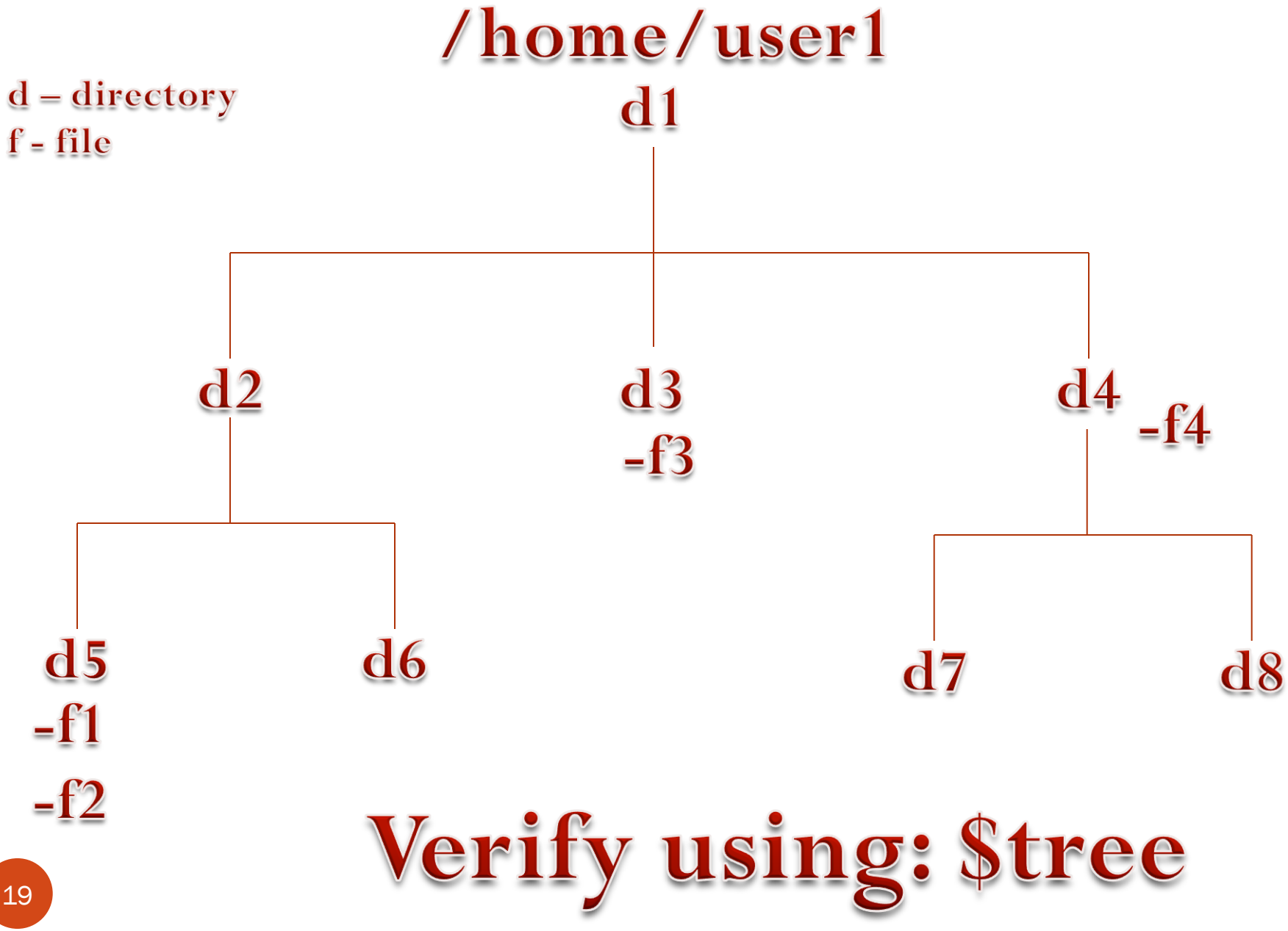
Getting familiar with Linux commands...

- ***To logout and shift to root user (or another user):***
  - Click on System
  - Click on Log Out user
  - Click on Log Out
  - Provide Username and Password

What is bash?

The shell program name in Linux is called bash.

Task



Verify using: $tree

Lab 2

Getting familiar with Linux Commands

- ***File permissions*: $ ls-l**

Normal file

Directory

```
-rwxrwxr-x    root    root
drwxr-xr-x    root    root
... ... ... ...
```

- *The first character is file type while the last nine characters are file permissions [first 3 is permission for owner user section, next 3 is permission for owner group, and the final 3 is permission for others].*
  - *r – read*
  - *w – write*
  - *x - execute*

Getting familiar with Linux Commands...

- ***To change file permissions:***

  chmod *who=permissions filename*

  u - The **u**ser that own the file.

  g - The **g**roup the file belongs to.

  o - The **o**ther users i.e. everyone else.

  a - **a**ll of the above - use this instead of having to type **ugo**.

Interaction with the Execution Environment

- When the operating system executes a program, it automatically provides certain facilities that help the program communicate with the operating system and the user. We consider the environment in which the programs, how they can use that environment to gain information about the operating system conditions, and how users of the program can alter their behavior.

The Argument List

- We run a program from a shell prompt by typing the name of the program. Optionally, we can supply the additional information to the program by typing one or more words after the program name, separated by spaces. These are called the ***command line arguments*** or the ***argument list***.
- Command line arguments are the arguments (or parameters) supplied after the name of a program in command line of operating systems like DOS and UNIX and are passed into the program via main() function.
- The main() function can accept two arguments: one argument counts the number of arguments and the other argument represents the full list of all of the command line arguments.

The Argument List

- So, the complete header of main() function that accepts command line arguments is:

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

  where, *argc* refers to the number of arguments passed (argument counter), and *argv[]* is an array of character pointers which points to each argument. The array of character pointers is the listing of all the arguments. Here, *argv[0]* is the name of the program, *argv[1]* is first argument, *argv[2]* is second argument and so on upto *argv[argc-1]*. Here, each *argv* element is used as a pointer to a string.

The following program demonstrates how to use *argc* and *argv*.

```
//arglist.c
#include < stdio.h>
int main ( int argc, char *argv[])
    {
    printf ("The name of this program is: %s\n", argv[0]+2);
    printf ("This program is invoked with %d arguments (excluding filename) \n", argc-1);
            if (argc>1)
            {
            int i;
            printf("\nThe arguments are :\n");
                    for (i = 1; i < argc; ++i)
                            printf("%s \n", argv[i]);
            }
    return 0;
    }
//Compile: $ gcc –o arglist arglist.c        //Run: $ ./arglist a b c
```

The Environment

- Linux provides each running program with an environment. The environment is a collection of variable/value pairs. Both environment variable names and their values are character strings. By convention, environment variable names are spelled in all capital letters.
- Examples:
  - USER: contains user name.
  - HOME: contains the path to the home directory.
  - PATH: contains the colon-separated list of directories through which Linux search for the command.
- ***Note: In Linux 'printenv' command is used to print the current environment variables.***

How to access environment variables?

```
//showenv.c
#include<stdio.h>
extern char **environ;
int main()
{
char **env = environ;
    while(*env)
    {
    printf("%s \n", *env);
    env++;
    }
return 0;
}
```

```
/* Note: After saving the above program, issue the following command to execute as command.
PATH=$PATH:
$ <filename> */
```

Tasks

1. Run the above programs at least three times and analyze the output.
2. Write the program which creates a new file called - ***makefile*** (equivalent to '***touch***' command).
3. Write the program equivalent with '***cp***' command of Unix with the name '***filecopy***'.
4. Execute: ***$echo $HOME*** command.

//filecopy program

```
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char *argv[])
{
FILE *f1,*f2;
char c;
if(argc!=3)
    {
    printf("\n Cannot process
    command");
    exit(1);
    }
f1=fopen(argv[1],"r");
f2=fopen(argv[2],"w");
if((f1==NULL)||(f2==NULL))
    {
    printf("\n Cannot open or create
    file");
    exit(1);
    }

while(fscanf(f1,"%c", &c)!=EOF)
    {
    fprintf(f2,"%c", c);
    }
return 0;
}
```

Lab 3

Looking at Processes

Process ID

- Each process in a Linux system is identified by its **unique** process identifier, referred to as ***pid***. Process IDs are 1- 32768 numbers that are assigned sequentially by Linux as new processes are created.
- Every process also has a parent process (except *init*). Thus, we can think that the processes in the Linux are arranged in a tree, with the *init* process at its root. The parent process ID, or ***ppid***, is simply the process ID of the process's parent.
- The process manipulation functions are declared in the header file ***<unistd.h>***. A program can obtain the process ID of the process it is running with the ***getpid()*** system call, and it can obtain the process ID of its parent process with the ***getppid()*** system call.

Printing the Process ID

```
//print_pid.c
#include<stdio.h>
#include<unistd.h>
int main()
{
int pid, ppid;
pid = getpid();
ppid = getppid();
printf("The Process ID is %d \n", pid);
printf("The Parent Process ID is %d \n", ppid);
return 0;
}

//Run this program three times and analyze the output
```

Analysis

- Process ID is an integer number greater than zero that OS provides for each running process. PID is the identification of a process. In Linux, we can see the process ID with the help of ***getpid()*** function call.
- As the process starts, it gets a PID. It's parent process ID can also be retrieved through ***getppid()*** call. Here we see that, each time the command is run, the process ID is different but its parent ID remains same (shell's ID). The reason is that the parent process is same but the process itself gets a different ID each time it runs.

Viewing Processes

- **ps - display the process running on that system.**
- **pstree - display the all process in process hierarchy.**
- **ps [options] - ps has different options.**

  Example: ***ps -e -o pid, user, start_time, command***

  ps -o pid, ppid, user

Creating a process

Using system (Building new process)

- The **system** function in the standard C library provides an easy way to execute a command within a program. The **system** creates a new sub-process running the standard **Broune shell** (/bin/sh) and hands the command to that shell for execution.

```
//system.c
#include<stdio.h>
int main()
{
system("ls");
return 0;
}
```

Task

- Modify the above program for creating three child processes using '**system**' system call.

```
//sys.c
#include <stdio.h>
int main()
{
printf("\n Watch out 'mkdir' below \n");
system("mkdir a");
printf("\n Watch out 'vi' below \n");
system("vi");
printf("\n Watch out 'ls' below \n");
system("ls");
return 0;
}
```

Creating a process...

Using fork and exec (Clone an existing one)

- Linux provides one function, ***fork()***, that makes the child process that is an exact copy of its parent process. Linux also provides another set of functions, the **exec** family, that replaces the current process image with a new process image.

Creating a process...

Using fork

- When program calls **fork()**, the **parent process** continues executing the program from the point that **fork()** was called. The **child process**, too, executes the same process from the same place. The return code for the **fork()** system call is zero for the new (child) process, whereas the (nonzero) process identifier of the child is returned to the parent.

```
//use_fork.c
#include<stdio.h>
int main()
{
printf("This is to demonstrate the fork\n");
fork();
printf("Hi Everybody\n");
}
```

Using fork

```
//fork_use.c
#include < stdio.h>
#include <unistd.h>
int main()
{
int pid;
printf("The main program 's process ID is %d \n", getpid());
pid = fork();     // First child process is run and then the parent process is run. For child
                  // process, return code is 0 and for parent process return code is pid of child.
if (pid != 0)
            {
            printf("This is the parent process, with id % d \n", getpid());
            printf("The child process ID is %d \n", pid);
            }
else
            printf("This is the child process, with id %d\n", getpid());
return 0;
}
```

Task

- Write a program that creates three processes using ***fork()*** system call.

```
#include <stdio.h>
#include <unistd.h>
int main()
{
int pid,pid1,pid2,pid3;
int ppid1,ppid2,ppid3;
int p1,p2,p3;
pid=getpid();
printf("\n Parent's pid=%d\n", pid);
p1=fork();
if(p1==0)
      {
      pid1=getpid();
      printf("\n Child 1's pid=%d\n",pid1);
      ppid1=getppid();
      printf("\n Child 1's parent's pid=%d\n",ppid1);

      p2=fork();
      if(p2==0)
            {
pid2=getpid();
printf("\n Child 2's pid=%d\n",pid2);
ppid2=getppid();
printf("\n Child 2's parent's pid=%d\n",ppid2);

p3=fork();
if(p3==0)
      {
      pid3=getpid();
      printf("\n Child 3's pid=%d\n",pid3);
      ppid3=getppid();
      printf("\n Child 3's parent's pid=%d\n",ppid3);
      }
    }
  }
return 0;
}
```

Task

- List the uses of **exec** family, and write at least one program that uses the **exec** system call.

exec() system call:

The exec() system call is used after a fork() system call by one of the two processes to replace the memory space with a new program. The exec() system call loads a binary file into memory (destroying image of the program containing the exec() system call) and go their separate ways.

exec family:

execl(): It permits us to pass a list of command line arguments to the program to be executed. The list of arguments is terminated by NULL.
Example: ***execl("/bin/ls", "ls", NULL);***

execlp(): It does same job except that it will use environment variable **PATH** to determine which executable to process. Thus a fully qualified path name would not have to be used. The function execlp() can also take the fully qualified name as it also resolves explicitly.
Example: ***execlp("ls", "ls", NULL);***

//Another example of fork

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
int main(int argc, char *argv[])
{
int pid;
pid=fork();     //fork another process
if(pid<0)       // error occurred
{
fprintf(stderr,"Fork Failed");
exit(-1);
}
else if(pid==0)    //child process
{
execlp("/bin/ls","ls", NULL);
}
else    //parent process
{
//parent will wait for child to complete
wait(NULL);
printf("\n Child Complete \n");
exit(0);
}
return 0;
}
```

The wait() system call:

- It blocks the calling process until one of its child processes exits or a signal is received. wait() takes the address of an integer variable and returns the process ID of the completed process.
- The main purpose of wait() is to wait for completion of child process. The execution of wait() could have two possible situations.
  1. If there are at least one child processes running when the call to wait() is made, the caller will be blocked until one of its child processes exits. At that moment, the caller resumes its execution.
  2. If there is no child process running when the call to wait() is made, then this wait() has no effect at all. That is, it is as if no wait() is there.

Lab 4

Process Termination

Process termination occurs either of two ways:

1. Normal exit: When the program calls exit function or when main function returns.
2. Killed by another process: Terminated abnormally in response to a signal.

Signals: (signal system calls are defined under the header <signal.h> and <sys/types.h>)

KILL: Forcefully make free (used in hang).

Example: *KILL pid* - send the signal to the targeted *pid* process.

Waiting for process termination

- Waiting can be done with the **wait** family of system call. These functions allow to wait for a process to finish executing, and enable the parent process to retrieve information about child's termination.

Process States

- The basic process states in Linux are: Running - R, Sleeping - S, Stopped - T and Zombie - Z.
- If the parent process terminated before child process, the executing child is an orphan process.
- A zombie process is a process that has terminated but has not been cleaned up yet. If the child process finish before the parent process calls wait, the child process becomes zombie (not removed from the system).
- ***$ ps –el*** – To look for processes with state information.

Checking process state

```
//orphan process
#include <stdio.h>
#include <unistd.h>
int main()
{
int pid;
printf("\n I am the original process with PID:%d & PPID:%d\n", getpid(), getppid());
pid=fork();
if(pid!=0)
    printf("\n I am the parent with PID:%d & PPID:%d\n", getpid(), getppid());
else
    {
    printf("\n I am the child with PID:%d & PPID:%d\n", getpid(), getppid());
    sleep(8);
    printf("\n I am the child with PID:%d & PPID:%d\n", getpid(), getppid());
    }
printf("\n PID %d terminates.\n", getpid());
}
```

Checking process state…

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
int main()
{
int pid;
pid=fork();
if(pid==0)
    {
    printf("\n I am the child, my pid is:%d\n",getpid());
    printf("\n I am the parent, my pid is:%d\n",getppid());
    sleep(20);

    printf("\n I am child, my pid is:%d\n",getpid());
    printf("\n I am parent, my pid is:%d\n",getppid());
    }
else
    {
    sleep(10);
    printf("\n I am the parent, my pid is:%d\n",getpid());
    printf("\n My parent's pid is:%d\n",getppid());
    for(;;);
    }
}
```

Tasks

1. Run the program above and issue the command ***ps -el*** three times in every 10 seconds and analyze the output.
2. Modify the program above to see running, sleeping and zombie state of program individually.
3. Insert the wait system call in the parent portion of above program and analyze the effect.

//Task 3: *insert_wait.c*

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
int main()
{
int pid;
pid=fork();
if(pid==0)
    {
    printf("\n I am the child, my pid is:%d\n",getpid());
    printf("\n I am the parent, my pid is:%d\n",getppid());
    sleep(20);

    printf("\n I am child, my pid is:%d\n",getpid());
    printf("\n I am parent, my pid is:%d\n",getppid());
    }
else
    {
    sleep(10);
    printf("\n I am the parent, my pid is:%d\n",getpid());
    printf("\n My parent's pid is:%d\n",getppid());
    wait(NULL);
    }
}
```

Process Switching

```
//pswitch.c
#include <stdio.h>
#include <unistd.h>
int main()
{
int pid;
pid=fork();
if(pid==0)
   {
   for(;;)
         printf("C");
   }
else
   {
   for(;;)
         printf("P");
   }
}
```

Note: fork() creates an exact copy of the parent process. So the child and parent processes are different with different PIDs. Here, first the child process is run so that C is printed infinitely. Then the process switch occurs and the parent process is run, so that P is printed infinitely. Again the process switch occurs, so that C is printed infinitely and so on.

Task

- Run the program above, check states and analyze the output.

Lab 5

Threads

- As with processes, threads appear to run concurrently; the Linux kernel schedules them asynchronously, interrupting each thread time to time to give others a chance to execute.
- Threads exist within a process. GNU/Linux implements the POSIX standard thread API (Pthreads). All thread functions and data types are declared in the header file <pthread.h>. The pthread functions are not included in the standard C library; they are in lpthread, therefore -lpthread should be added when linking program.

Thread Creation

- Each thread have their own thread ID and is referred by type ***pthread_t***.
- The statement ***pthread_t tid*** declares the identifier for the thread we will create.
- Each thread has a set of attributes including stack size and scheduling information.
- The ***pthread_attr_t attr*** declaration represents the attributes for the thread such as priority.
- To use the default attributes provided, set the attributes in the function call as ***pthread_attr_init(&attr)*** *or use* ***NULL***.

Thread Creation

- The ***pthread_create*** function creates new threads. The new thread starts execution by invoking *start_routine*(); *arg* is passed as the sole argument of *start_routine*(). It has following format:

pthread_create(pthread_t *thread, pthread_attr_t *attr, void *(*start_routine)(void*), void *arg);

- The new thread terminates in one of the following ways:
  - It calls **pthread_exit**(), specifying an exit status value that is available to another thread in the same process that calls **pthread_join**().
  - It returns from *start_routine*(). This is equivalent to calling **pthread_exit**() with the value supplied in the *return* statement.
- The ***pthread_exit*** function terminates the thread:

  pthread_exit(void *return_val);

- The ***pthread_join*** function waits other thread for termination - equivalent of *wait*.

  pthread_join(pthread_t th, void **thread_return);

//thread5.c

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#define NO_OF_THREADS 5

void *print_hello_world(void *tid)
{
//This function prints thread's identifier and then exits
printf("Hello World from thread %d\n\n",*((int *)tid));
//casting void pointer
pthread_exit(NULL);
}
```

```
int main()
{
//The main program creates 5 threads
pthread_t threads[NO_OF_THREADS];
int status, i;

for(i=0;i<NO_OF_THREADS;i++)
    {
    printf("Main here. Creating thread %d\n", i);
    status=pthread_create(&threads[i],NULL,&print_hello_world,&i);
    }
}
```

//compile: gcc -o thread5 thread5.c –lpthread

//run: ./thread5

//threadc.c

```
#include <stdio.h>
#include <unistd.h>
#include <pthread.h>
struct param
{
char ch; //The character to print
int count; //Number of times to print it
};

void *printc(void *parameter) //prints no. of characters in stderr
{
struct param *p=(struct param *)parameter;
int i;
for(i=0;i<p->count;++i)
        fputc(p->ch, stderr);
return NULL;
}
```

```
int main()
{
pthread_t thread1_id;
pthread_t thread2_id;
struct param thread1_args;
struct param thread2_args;

thread1_args.ch='T'; //new thread to print 30000 T's
thread1_args.count=30000;
pthread_create(&thread1_id,NULL,&printc,&thread1_args);

thread2_args.ch='t'; //new thread to print 20000 t's
thread2_args.count=20000;
pthread_create(&thread2_id,NULL,&printc,&thread2_args);

pthread_join(thread1_id, NULL);//wait first thread to finish
pthread_join(thread2_id, NULL);//wait second thread to finish
return 0;
}
```

Task

1. Run the program above and analyze the output; what changes will be in your output when you remove last two lines (*pthread_join*), if any changes, give reason.
2. Write a program using threads that prints sum of numbers up to given positive number *n*.
3. Write a program that have two threads, one reads a word from keyboard and another checks for valid word (you can use your own word list (at least 5) to check for validity).

//sum.c

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

int sum;            //this data is shared by the thread(s)
void *runner(void *param);        //the thread

int main(int argc, char *argv[])
{
pthread_t tid;    //thread identifier
pthread_attr_t attr;                    //set of thread attributes
if(argc!=2)
      {
      fprintf(stderr,"usage: ./a.out <integer value>\n");
      exit(1);
      }

if(atoi(argv[1])<0)
      {
      fprintf(stderr, "The number '%d' must be >=0\n", atoi(argv[1]));
      exit(1);
      }

//get the default attributes
pthread_attr_init(&attr);

//create the thread
pthread_create(&tid,&attr,&runner,argv[1]);

//now wait for the thread to exit
pthread_join(tid, NULL);

printf("sum=%d\n",sum);
return 0;
}

//thread begins control in this section
void *runner(void *param)
{
int upper=atoi(param);
int i;
sum=0;
if(upper>0)
      {
      for(i=1;i<=upper;i++)
                  sum += i;
      }
}
```

//2threads.c

```
#include <stdio.h>
#include <pthread.h>

void *input(void *s)
{
char *p=(char *)s;
printf("Enter word:");
scanf("%s", p);
printf("The entered word is:%s", p);
return NULL;
}

void *check(void *s)
{
nt i=0;
char *p=(char *)s;
//5 names having at most 6 characters
char valid[5][6]={"ram","shyam","hari","lok","shere"};
while(i<5)
        {
        if(strcmp(p,valid[i])==0)
                        {
                        printf("\n Valid Word");
                        break;
                        }
                else
                        {
                        ++i;
                        }
        }
if(i==5)
        {
        printf("\n Invalid word");
        }
return NULL;
}

int main()
{
char s[20];
pthread_t t1;
pthread_t t2;
pthread_create(&t1,NULL,&input,s);
pthread_join(t1,NULL);

pthread_create(&t2,NULL,&check,s);
pthread_join(t2,NULL);
return 0;
}
```

Lab 6

Interprocess Communication and Synchronization

- ***Task:*** Review race condition, critical region, and mutual exclusion with busy waiting from your Textbook (Tanenbaum).

Synchronization Problem Implementation

- The following program demonstrates the use of ***strict alternation*** method to achieve mutual exclusion for critical region problem.

```
#include <stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<pthread.h>

  void *thread1f(void *arg);
  void *thread2f(void *arg);

  int turn=1;

  int main()
   {
    pthread_t thid1;
    pthread_t thid2;

    pthread_create(&thid1,NULL,&thread1f,NULL);
    pthread_create(&thid2,NULL,&thread2f,NULL);

    pthread_join(thid1,NULL);
    pthread_join(thid2,NULL);

    return 0;
   }
```

```
void *thread1f(void *arg)
 {
  int a=0;

  while(a++<20)
    {
     while(turn!=1);
      fputc('b',stderr);
      turn=0;
    }

 }

void *thread2f(void *arg)
  {
   int b=0;

   while(b++<20)
     {
      while(turn!=0);

      fputc('a',stderr);
      turn=1;
      }
  }
```

Tasks

1. Run the program above and analyze the output.
2. Modify the above program to demonstrate lock variables solution, and comment on the deficiencies of this solution.
3. Write the problems in the solution of above program and develop new version of the program using Peterson's solution.

Solution 1

- ***OUTPUT:***

  [root@localhost lab6]# ./stralt

  bababababababababababababababababababa

- ***Analysis:*** This is an example of how strict alternation works in IPC. Here we have two processes (threads). Each process (thread) have a task of printing the letter 'a' and 'b' twenty times. Here we have used the turn variable which keeps track of which process (thread) is currently in its CR. A process P_i which wants to enter CR is allowed only when turn=i, otherwise it goes in a tight loop until turn becomes i. That is why this method is called strict alternation. We see the output as slowly printing 'b' and 'a' alternately. This method greatly wastes CPU time.

Solution 2: To demonstrate the use of 'lock' variable in IPC

```
#include<stdlib.h>
#include<unistd.h>
#include<pthread.h>
#include<stdio.h>
void *thread1f(void *arg);
void *thread2f(void *arg);
int lock=0;
int main()
{
pthread_t thid1;
pthread_t thid2;
pthread_create(&thid1,NULL,&thread1f,NULL);
pthread_create(&thid2,NULL,&thread2f,NULL);
pthread_join(thid1,NULL);
pthread_join(thid2,NULL);
return 0;
}
```

```
void *thread1f(void *arg)
{
 int a=0;
while(a++<20000)
      {
       if(lock==0)
        {
         lock=1;
         fputc('b',stderr);
         lock=0;
        }
       else
        while(lock!=0);
      }
}
```

```
void *thread2f(void *arg)
{
 int b=0;
 while(b++<20000)
      {
       if(lock==0)
        {
         lock=1;
         fputc('a',stderr);
         lock=0;
        }
       else
        while(lock!=0);
      }
}
```

Output of Solution 2

OUTPUT:

bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb
bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb
bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb
bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb....

aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa

Analysis of Solution 2

ANALYSIS:

The above program is a demonstration of the use of ***lock*** variable in IPC. Here we have two processes (threads) - one prints 'b' 20000 times and other prints 'a' 20000 times. The task of printing is a critical task and we want that if one process is printing, the other should not intervene. For this, we used the lock variable. Lock is initially 0. If a process wants to enter its critical region, it first tests the lock whether it is zero or not. If the lock is zero, the process first makes the lock 1 and performs its critical task and again makes lock 0 before leaving its CR. If the lock is not zero , the process just waits until it is zero. Here, if we do not use lock then we will have mixed output i.e. some b's printed first then some a's and so on upto 20000 due to cpu switching between two processes. But with the use of lock, one process starts printing and whole of its output is printed even if its quantum expires. When all of the output of one process is printed then only the other process gets chance and goes into its CR.

Solution 3: Using Peterson's solution for achieving mutual exclusion

```
#include<stdlib.h>
#include<unistd.h>
#include<pthread.h>
#include<stdio.h>
#define FALSE 0
#define TRUE 1
#define N 2
void *thread1f(void *arg);
void *thread2f(void *arg);
void enter_region(int);
void leave_region(int);
int interested[N];
int turn;
int main()
{
pthread_t thid1;
pthread_t thid2;
pthread_create(&thid1,NULL,&thread1f,NULL);
pthread_create(&thid2,NULL,&thread2f,NULL);
pthread_join(thid1,NULL);
pthread_join(thid2,NULL);
return 0;
}

void *thread1f(void *arg)
{
int a=0;
enter_region(0);
while(a++<20000)
  {
   fputc('b',stderr);
  }
 leave_region(0);
}

void *thread2f(void *arg)
{
  int b=0;
  enter_region(1);
  while(b++<20000)
     {
      fputc('a',stderr);
     }
   leave_region(1);
  }

void enter_region(int process)
{
int other;
other=1-process;
interested[process]=TRUE;
turn=process;
while(turn==process&&interested[other]==TRUE);
}

void leave_region(int process)
{
interested[process]=FALSE;
}
```

Output of Solution 3

- ***OUTPUT:***

bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb
bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb
bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb
bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb
bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb
bbbbbbbbbbbbbbbbbbbbbbbbbb...

aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa...

Analysis of Solution 3

ANALYSIS: Peterson's solution is a simpler way to achieve mutual exclusion. It makes the use of theory of lock variable and strict alternation and provides a new way of achieving mutual exclusion. Any process that wants to enter its CR calls enter_region() with its own process number as argument and calls leave_region() after leaving its CR. Here we have two processes which have their own critical task of printing a letter 'a' and 'b'. We see that one process prints its whole output and other process do not interfere even the quantum of first process expires. When all the output of first process is finished then other process gets chance and starts executing its CR.

Lab 7

Process Scheduling

- We implement all the process scheduling algorithms in Linux platform using C++ programming language. Here we simulate the nature of different scheduling algorithms (FCFS, SJF, SRTN, RR and Priority) and make the analysis of their average waiting time and average turnaround time.

- ***Final Goal:*** Develop a menu-driven program by implementing the various scheduling algorithms.

FCFS

- ***Task 1:*** Write a program in C++ to implement the FCFS process scheduling algorithm and calculate the average waiting time for the following set of data:

Processes	***Burst Time***
P0	24
P1	3
P2	3

```
//fcfs.cpp
#include<iostream>
using namespace std;
    //Uses standard namespace std and is compulsory for
    //C++ programs in Linux. All basic elements of
    //standard C++ library are declared within a default
    //scope called the namespace with name std. We use
    //this expression in order to access basic elements of
    //C++ library. This statement is very frequent in C++
    //programs to use standard library. Some compilers
    //don't support namespace.
class ProcessScheduling
{
int n, burst_time[20];
float waiting_time[20], total_waiting_time;
float average_waiting_time;
public:
    //Getting the No of processes & burst time
    void GetData();
    //First come First served Algorithm
    void FCFS();
};
    //Getting no. of processes and Burst time
void ProcessScheduling::GetData()
{
 int i;
 cout<<"Enter the no. of processes(<20):";
 cin>>n;
 for(i=0;i<n;i++)
  {
   cout<<"Enter burst time for process P"<<i<<"=";
   cin>>burst_time[i];
   }
}
```

```
//First come First served Algorithm
void ProcessScheduling::FCFS()
{
int i;
total_waiting_time=0.0;
for(i=0;i<n;i++)
    {
    cout<<"Burst time for process P"<<i<<"=";
    cout<<burst_time[i]<<endl;
    }

waiting_time[0]=0;
for(i=1;i<n;i++)
    {
    waiting_time[i]=burst_time[i-1]+waiting_time[i-1];
    }

    //Calculating Average Waiting Time
    for(i=0;i<n;i++)
        total_waiting_time = total_waiting_time +
                             waiting_time[i];
average_waiting_time = total_waiting_time/n;

cout<<"Total Waiting Time="<<total_waiting_time;
cout<<endl;
cout<<"Average Waiting Time="<<
average_waiting_time;
}

int main()
{
ProcessScheduling p;
p.GetData();
p.FCFS();
return 0;
}

//Compile: g++ fifo.cpp
//Run: ./a.out
```

- ***Task 2:*** **Add the calculation for turnaround time in the function FCFS and print the average turnaround time for the set above.**

Hint:

```
class ProcessScheduling
{
        int n, burst_time[20];
        float waiting_time[20], total_waiting_time, average_waiting_time;
        float turnaround_time[20], total_turnaround_time, average_turnaround_time;
    public:
        void GetData();
        void FCFS();
};
//Add the following code in function FCFS() after finding the waiting time
for(i=0;i<n;i++)
    {
    cout<<endl<<"Turnaround time for P"<<i<<"=";
    turnaround_time[i]=burst_time[i]+waiting_time[i];
    cout<<turnaround_time[i];
    }
    total_turnaround_time=0;
    for(i=0;i<n;i++)
        total_turnaround_time += turnaround_time[i];
    average_turnaround_time=total_turnaround_time/n;
    cout<<endl<<"Average Turnaround Time="<<average_turnaround_time;
```

- ***Task 3:*** Advance the FCFS algorithm to take input for arrival times of processes, then sort the processes according to arrival times and finally apply the FCFS algorithm.

- ***Task 4:*** Insert another function SJF() into the class ProcessScheduling and write a suitable function that implements the SJF algorithm. Make the program menu-driven.
- [Hint: SJF selects the process with shortest CPU burst time. Here the idea of coding is same as of FCFS. The only difference is that we first sort the processes according to their burst time(smaller to larger).]

```
#include<iostream>
using namespace std;

class ProcessScheduling
{
    int n, burst_time[20];
    float waiting_time[20], total_waiting_time,
    average_waiting_time;
public:
    //Getting the No of processes & burst time
    void GetData();
    void FCFS();
    void SJF();
};
```

```
//Getting no. of processes and Burst time
void ProcessScheduling::GetData()
{
    int i;
    cout<<"Enter the no. of processes(<20):";
    cin>>n;
    for(i=0;i<n;i++)
    {
    cout<<"Enter burst time for process P"<<i<<"=";
    cin>>burst_time[i];
    }
}
```

```
//First come First served Algorithm
void ProcessScheduling::FCFS()
{
    int i;
    total_waiting_time=0.0;
    for(i=0;i<n;i++)
    {
    cout<<"Burst time for process P"<<i<<"=";
    cout<<burst_time[i]<<endl;
    }

    waiting_time[0]=0;
    for(i=1;i<n;i++)
    {
    waiting_time[i]=burst_time[i-1]+waiting_time[i-1];
    }
```

```
    //Calculating Average Waiting Time
for(i=0;i<n;i++)
  total_waiting_time=total_waiting_time+waiting_time[i];

   average_waiting_time=total_waiting_time/n;

     cout<<"Total Waiting
     Time="<<total_waiting_time<<endl;
     cout<<"Average Waiting
     Time="<<average_waiting_time;
}
```

```
void ProcessScheduling::SJF()
{
int i, j, temp,burst_t[20];
total_waiting_time=0.0;
for(i=0;i<n;i++)
    {
    burst_t[i]=burst_time[i];
    cout<<"Burst time for process P"<<i<<"=";
    cout<<burst_t[i]<<endl;
    }
//sort processes according to burst time
for(i=0;i<n-1;i++)
    {
    for(j=i+1;j<n;j++)
            {
            if(burst_t[i]>burst_t[j])
                    {
                    temp=burst_t[i];
                    burst_t[i]=burst_t[j];
                    burst_t[j]=temp;
                    }
            }
    }
waiting_time[0]=0;
for(i=1;i<n;i++)
    waiting_time[i]=burst_t[i-1]+waiting_time[i-1];

for(i=0;i<n;i++)
  total_waiting_time=total_waiting_time+waiting_time[i];

average_waiting_time=total_waiting_time/n;
cout<<"Total Waiting
    Time="<<total_waiting_time<<endl;
cout<<"Average Waiting
    Time="<<average_waiting_time;
}
```

```
int main()
{
char ch='y';
int option;
ProcessScheduling p;
cout<<"Implementation of Process Scheduling
    Algorithms"<<endl;
cout<<"Enter information about processes"<<endl;
p.GetData();
cout<<"Which scheduling algorithm you wanna
    use?"<<endl;
cout<<"1.FCFS"<<endl;
cout<<"2.SJF"<<endl;

while(ch=='y')
    {
    cout<<"Enter your option:";
    cin>>option;
switch(option)
        {
        case 1:
                p.FCFS();
                break;
        case 2:
                p.SJF();
                break;
        default:
                cout<<"Wrong Option!!!";
        }
    cout<<endl<<"Wanna test other algorithms(y/n)?";
    cin>>ch;
    }
return 0;
}
```

- ***Task 5:*** Advance the SJF algorithm to take input for arrival times of processes and then apply the SJF algorithm.

```
//Shortest job First Algorithm with Nonpreemption and Arrival Time
void ProcessScheduling::SJF_AT()
{
int i, burst_t[20], Tt=0,temp,j;
char S[20];
float A[20],temp1,t,w;
total_waiting_time=0.0;
w=0.0;
for(i=0;i<n;i++)
    {
    burst_t[i]=burst_time[i];
    cout<<"Burst time for process p"<<i<<"=";
    cout<<burst_t[i];
    S[i]='T';
    Tt=Tt+burst_t[i];
    cout<<endl<<"Enter the Arrival Time for"<<i<<"th process=";
    cin>>A[i];
    }
for(i=1;i<n-1;i++)
{
  for(j=i+1;j<n;j++)
  {
        if(burst_t[i]>burst_t[j])
        {
        temp=burst_t[i];
        burst_t[i]=burst_t[j];
        burst_t[j]=temp;

        temp1=A[i];
        A[i]=A[j];
        A[j]=temp1;
        }
  }
}
```

```
//processes sorted according to burst time
for(i=0;i<n;i++)
cout<<endl<<"P"<<i<<"\t"<<A[i]<<"\t"<<burst_t[i]<<endl;

//For the 1st process
waiting_time[0]=0;
w=w+burst_t[0];
t=w;
S[0]='F';

while(w<Tt)
    {
    i=1;
    while(i<n)
            {
            if(S[i]=='T'&& A[i]<=t)
                    {
                    waiting_time[i]=w;
                    S[i]='F';
                    w=w+burst_t[i];
                    t=w;
                    }
            else
                    i++;
            }
    }

//calculating average waiting Time
for(i=0;i<n;i++)
    total_waiting_time=total_waiting_time+(waiting_time[i]-A[i]);
average_waiting_time=total_waiting_time/n;
cout<<"Total Waiting Time="<<total_waiting_time<<endl;
cout<<"Average Waiting Time="<<average_waiting_time;
}
```

- ***Task 6:*** Add other process scheduling algorithms **SRTN**, **RR** and **Priority scheduling** and advance the menu program for computing their average waiting time and average turnaround time.

C function for SRTN

```
void ProcessScheduling::SRTN()
{
int i,j,burst_t[20],arrival_t[20],Tt=0;
char S[20],start[20];
int max=0,Time=0,min;
float total_waiting_time=0.0,average_waiting_time;

for(i=0;i<n;i++)
{
burst_t[i]=burst_time[i];
cout<<"Burst time for process P"<<i<<"="<<burst_t[i];
if(burst_t[i]>max)
    max=burst_t[i];
waiting_time[i]=0;
S[i]='T';
start[i]='F';
Tt=Tt+burst_t[i];
cout<<endl<<"Enter the Arrival Time for"<<i<<"th
    process=";
cin>>arrival_t[i];
if(arrival_t[i]>Time)
    Time=arrival_t[i];
}
```

```
int w=0,flag=0,t=0;
i=0;
while(t<Time)
{
    if(arrival_t[i]<=t && burst_t[i]!=0)
    {
      if(flag==0)
      {
      waiting_time[i]=waiting_time[i]+w;

    cout<<endl<<"WT["<<i<<"]="<<waiting_ti
    me[i];
      }
    burst_t[i]=burst_t[i]-1;
    if(burst_t[i]==0)
              S[i]='F';
    start[i]='T';
    t++;
    w=w+1;
if(S[i]!='F')
   {
    j=0;flag=1;
    while(j<n && flag!=0)
          {
          if(S[j]!='F'  &&  burst_t[i]>burst_t[j]  &&
arrival_t[j]<=t && i!=j )
             {
              flag=0;
              waiting_time[i]=waiting_time[i]-w;
              i=j;
             }
          else
             {
              flag=1;
             }
          j++;
          }
}
```

```
else
                {
                                i++;
                                j=0;
                                while(arrival_t[j]<=t && j<n)
                                {
                                                if(burst_t[i]>burst_t[j] && S[j]!='F')
                                                {
                                                                flag=0;
                                                                i=j;
                                                }
                                                j++;
                                }
                }
        }
        else
                if(flag==0)
                                i++;
}


cout<<endl<<"Printing remaining burst time";
for(i=0;i<n;i++)
        cout<<endl<<"B["<<i<<"]="<<burst_t[i];
```

```
while(w<Tt)
{
    min=max+1;
    i=0;
    while(i<n)
    {
        if(min>burst_t[i] && S[i]=='T')
        {
            min=burst_t[i];
            j=i;
        }
        i++;
    }
    i=j;
```

```
if(w==Time && start[i]=='T')
    {
        w=w+burst_t[i];
        S[i]='F';
    }
    else
    {
        waiting_time[i]=waiting_time[i]+w;
        w=w+burst_t[i];
        S[i]='F';
    }
}
```

```
cout<<endl<<"Waiting information:";
for(i=0;i<n;i++)
    cout<<endl<<"WT["<<i<<"]="<<waiting_time[i];

cout<<endl<<"After subtracting arrival time";
for(i=0;i<n;i++)
{
waiting_time[i]=waiting_time[i]-arrival_t[i];
cout<<endl<<"WT["<<i<<"]="<<waiting_time[i];
}

//Calculating Average Waiting time
for(i=0;i<n;i++)
    total_waiting_time=total_waiting_time+waiting_time[i];
average_waiting_time=total_waiting_time/n;
cout<<endl<<"Average Waiting Time="<<average_waiting_time;
}
```

C function for RR

```
void ProcessScheduling::RoundRobin()
{
int i, j, k=0,time_quantum,tem[20], burst_t[20], turnaround_time[20];
static int gantt[50],temp;
float average_turnaround_time, total_turnaround_time=0;
total_waiting_time=0;

cout<<"Enter the time quantum:";
cin>>time_quantum;

for(i=0;i<n;i++)
        burst_t[i]=burst_time[i];

for(i=0;i<n;i++)
{
if((burst_time[i]%time_quantum)==0)
        tem[i]=burst_time[i]/time_quantum;
else
        tem[i]=(burst_time[i]/time_quantum)+1;
}
```

```
for(i=0;i<n;i++)
{
    if(tem[i]>tem[0])
    {
    temp=tem[0];
    tem[0]=tem[i];
    tem[i]=temp;
    }
}


for(i=1;i<=tem[0];i++)
{
for(j=0;j<n;j++)
    {
    if(burst_time[j]!=0)
            {
            if(burst_time[j]>=time_quantum)
                    {
                    gantt[k+1]=gantt[k]+time_quantum;
                    burst_time[j]=burst_time[j]-time_quantum;
                            if(burst_time[j]==0)
                            {

            waiting_time[j]=gantt[k+1]-burst_t[j];

            turnaround_time[j]=gantt[k+1];
                            }
                    }
                    else
                            {
                            gantt[k+1]=gantt[k]+burst_time[j];
                            burst_time[j]=0;
                            turnaround_time[j]=gantt[k+1];
                            waiting_time[j]=gantt[k+1]-burst_t[j];
                            }
                    k++;
                    }
        }
}

cout<<"\tProcess\t\tBurst time\tWaiting time\tTurnaround time\n";
for(i=0;i<n;i++)
{
cout<<"\tp["<<i<<"]\t\t"<<burst_t[i]<<"\t\t"<<waiting_time[i]<<"\t\t"<<t
    urnaround_time[i];
total_waiting_time=total_waiting_time+waiting_time[i];
total_turnaround_time = total_turnaround_time+turnaround_time[i];
cout<<endl;
}

average_waiting_time=total_waiting_time/n;
average_turnaround_time=total_turnaround_time/n;
cout<<"Average waiting time:"<<average_waiting_time<<endl;
cout<<"Average turn around time:"<<average_turnaround_time;
}
```

C function for priority scheduling

- ***Priority scheduling:*** Each process is to be assigned a priority value. The process selected is the one with the highest priority. Hence we require another array *p[20]* to store the priority value of each process. The priority value of process is assigned at the runtime.

```
void ProcessScheduling::Priority()
{
int burst_t[20], p[20], i, j;
float w;
int max=1;
total_waiting_time=0.0;
for(i=0;i<n;i++)
    {
    burst_t[i]=burst_time[i];
    cout<<"Burst time for process p"<<i<<"=";
    cout<<burst_t[i]<<endl;
    cout<<"Enter priority for process P"<<i<<"=";
    cin>>p[i];
    if(max<p[i])
            max=p[i];
    }

w=0.0;
for(j=1;j<=max;j++)
    {
    for(i=0;i<n;i++)
        {
        if(p[i]==j)
        //search for highest priority (lowest value) process
                {
                waiting_time[i]=w;
                w=w+burst_time[i];
        //previous burst time is waiting time for next process
                }
        }
    }

//calculate average waiting time
for(i=0;i<n;i++)
    total_waiting_time+=waiting_time[i];
average_waiting_time=total_waiting_time/n;
cout<<"Average waiting time="<<
    average_waiting_time;
}
```

Insert code for turnaround time yourself

Task 7: *Modify the above function for* ***priority scheduling*** *so that it first sorts the processes according to their priority and then use the same coding technique as for other algorithms to calculate average waiting time and average turnaround time.*

Lab 8