18CSC205J – OPERATING SYSTEMS

SEMESTER – IV

Name of the Student :

Register Number :

DEPARTMENT OF COMPUTING TECHNOLOGIES

SCHOOL OF COMPUTING

SRM INSTITUTE OF SCIENCE AND TECHNOLOGY
(Deemed University u/s 3 of UGC Act 1956)
Kattankulathur, Chengalpattu District, 603 202.

COLLEGE OF ENGINEERING AND TECHNOLOGY
SRM INSTITUTE OF SCIENCE AND TECHNOLOGY
(Deemed University u/s 3 of UGC Act 1956)
Kattankulathur, Chengalpattu District, 603 202

B O N A F I D E C E R T I F I C A T E

Register Number : _____

Certified to be the bonafide record of work done by _____

of **B.Tech.(CSE)** Degree course in the Practical **18CSC205J–Operating Systems** in SRM Institute of Science and Technology, Kattankulathur during the Academic Year _____.

Date : **Faculty In-charge**

Head of the Department

Submitted for the University Examinations held on_____, at _____, SRM Institute of Science and Technology, Kattankulathur.

Date : **Examiner-1** **Examiner-2**

C O N T E N T

Ex. No. 1a	OPERATING SYSTEM INSTALLATION	Date :

Linux operating system can be installed as either dual OS in your system or you can install through a virtual machine (VM).

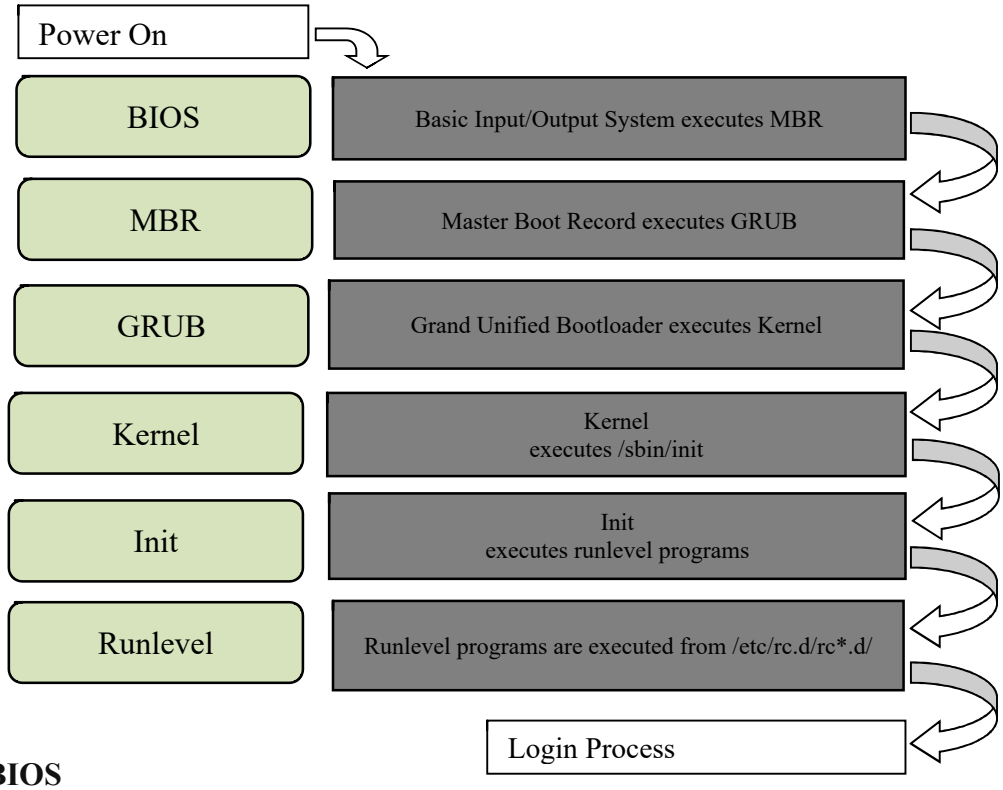
Installation of Ubuntu in your Windows OS through a Virtual machine

Steps

1. Download VMware Player or Workstation recent version.
2. Download Ubuntu LTS recent version.
3. Install VM ware Player in your host machine.
4. Open VMware Workstation and click on "New Virtual Machine".
5. Select "Typical (recommended)" and click "Next".
6. Select "Installer disc image (ISO)", click "Browse" to select the Ubuntu ISO file, click "Open" then "Next".
7. You have to type in "Full name", "User name" that must only consist of lowercase and numbers then you must enter a password. After you finished, click "Next".
8. You can type in a different name in "Virtual machine name" or leave as is and select an appropriate location to store the virtual machine by clicking on "Browse" that is next to "Location" -- you should place it in a drive/partition that has at least 5GB of free space. After you selected the location click "OK" then "Next".
9. In "Maximum disk size" per Ubuntu recommendations you should allocate at least 5GB -- double is recommended to avoid running out of free space.
10. Select "Store virtual disk as a single file" for optimum performance and click "Next".
11. Click on "Customize" and go to "Memory" to allocate more RAM -- 1GB should suffice, but more is always better if you can spare from the installed RAM.
12. Go to "Processors" and select the "Number of processors" that for a normal computer is 1 and "Number of cores per processor" that is 1 for single core, 2 for dual core, 4 for quad core and so on -- this is to insure optimum performance of the virtual machine.
13. Click "Close" then "Finish" to start the Ubuntu install process.
14. On the completion of installation, login to the system

Ex. No. 1b	BOOTING PROCESS OF LINUX	Date :

Press the power button on your system, and after few moments you see the Linux login prompt. From the time you press the power button until the Linux login prompt appears, the following sequence occurs. The following are the 6 high level stages of a typical Linux boot process.



Step 1.BIOS

- BIOS stands for Basic Input/Output System
- Performs some system integrity checks
- Searches, loads, and executes the boot loader program.
- It looks for boot loader in floppy, CD-ROMs, or hard drive. You can press a key (typically F12 or F2, but it depends on your system) during the BIOS startup to change the boot sequence.
- Once the boot loader program is detected and loaded into the memory, BIOS gives the control to it.
- So, in simple terms BIOS loads and executes the MBR boot loader.

Step 2. MBR

- MBR stands for Master Boot Record.
- It is located in the 1st sector of the bootable disk. Typically /dev/hda, or /dev/sda
- MBR is less than 512 bytes in size. This has three components 1) primary boot loader info in 1st 446 bytes 2) partition table info in next 64 bytes 3) mbr validation check in last 2 bytes.
- It contains information about GRUB (or LILO in old systems).
- So, in simple terms MBR loads and executes the GRUB boot loader.

Step 3. GRUB

- GRUB stands for Grand Unified Bootloader.
- If you have multiple kernel images installed on your system, you can choose which one to be executed.
- GRUB displays a splash screen, waits for few seconds, if you don't enter anything, it loads the default kernel image as specified in the grub configuration file.
- GRUB has the knowledge of the filesystem (the older Linux loader LILO didn't understand filesystem).
- Grub configuration file is /boot/grub/grub.conf (/etc/grub.conf is a link to this). The following is sample grub.conf of CentOS.

```
#boot=/dev/sda
default=0
timeout=5
splashimage=(hd0,0)/boot/grub/splash.xpm.gz
hiddenmenu
title CentOS(2.6.18-194.el5PAE)
root(hd0,0)
kernel/boot/vmlinuz-2.6.18-194.el5PAE ro root=LABEL=/
initrd /boot/initrd-2.6.18-194.el5PAE.img
```

- As you notice from the above info, it contains kernel and initrd image.
- So, in simple terms GRUB just loads and executes Kernel and initrd images.

Step 4. Kernel

- Mounts the root file system as specified in the "root=" in grub.conf
- Kernel executes the /sbin/init program
- Since init was the 1st program to be executed by Linux Kernel, it has the process id (PID) of 1. Do a 'ps -ef | grep init' and check the pid.
- initrd stands for Initial RAM Disk.
- initrd is used by kernel as temporary root file system until kernel is booted and the real root file system is mounted. It also contains necessary drivers compiled inside, which helps it to access the hard drive partitions, and other hardware.

Step 5. Init

- Looks at the /etc/inittab file to decide the Linux run level.
- Following are the available run levels
    - 0 – halt
    - 1 – Single user mode
    - 2 – Multiuser, without NFS
    - 3 – Full multiuser mode
    - 4 – unused
    - 5 – X11
    - 6 – reboot
- Init identifies the default initlevel from /etc/inittab and uses that to load all appropriate program.

- Execute ‘grep initdefault /etc/inittab’ on your system to identify the default run level
- If you want to get into trouble, you can set the default run level to 0 or 6. Since you know what 0 and 6 means, probably you might not do that.
- Typically you would set the default run level to either 3 or 5.

Step 6. Runlevel programs

- When the Linux system is booting up, you might see various services getting started. For example, it might say “starting sendmail …. OK”. Those are the runlevel programs, executed from the run level directory as defined by your run level.
- Depending on your default init level setting, the system will execute the programs from one of the following directories.
    - Run level 0 – /etc/rc.d/rc0.d/
    - Run level 1 – /etc/rc.d/rc1.d/
    - Run level 2 – /etc/rc.d/rc2.d/
    - Run level 3 – /etc/rc.d/rc3.d/
    - Run level 4 – /etc/rc.d/rc4.d/
    - Run level 5 – /etc/rc.d/rc5.d/
    - Run level 6 – /etc/rc.d/rc6.d/
- Please note that there are also symbolic links available for these directory under /etc directly. So, /etc/rc0.d is linked to /etc/rc.d/rc0.d.
- Under the /etc/rc.d/rc*.d/ directories, you would see programs that start with S and K.
- Programs starts with S are used during startup. S for startup.
- Programs starts with K are used during shutdown. K for kill.
- There are numbers right next to S and K in the program names. Those are the sequence number in which the programs should be started or killed.
- For example, S12syslog is to start the syslog deamon, which has the sequence number of 12. S80sendmail is to start the sendmail daemon, which has the sequence number of 80. So, syslog program will be started before sendmail.

Login Process

1. Users enter their username and password
2. The operating system confirms your name and password.
3. A "shell" is created for you based on your entry in the "/etc/passwd" file
4. You are "placed" in your "home"directory.
5. Start-up information is read from the file named "/etc/profile". This file is known as the system login file. When every user logs in, they read the information in this file.
6. Additional information is read from the file named ".profile" that is located in your "home" directory. This file is known as your personal login file.

Ex. No. 2a	BASIC LINUX COMMANDS	Date :

a) Basics

1. *echo* SRM → to display the string SRM
2. *clear* → to clear the screen
3. *date* → to display the current date and time
4. *cal* 2003 → to display the calendar for the year 2003
   cal 6 2003 → to display the calendar for the June-2003
5. *passwd* → to change password

b) Working with Files

1. *ls* → list files in the present working directory
   ls –l → list files with detailed information (long list)
   ls –a → list all files including the hidden files
2. cat > f1 → to create a file (Press ^d to finish typing)
3. *cat* f1 → display the content of the file f1
4. *wc* f1 → list no. of characters, words & lines of a file f1
   wc –c f1 → list only no. of characters of file f1
   wc –w f1 → list only no. of words of file f1
   wc –l f1 → list only no. of lines of file f1
5. *cp* f1 f2 → copy file f1 into f2
6. *mv* f1 f2 → rename file f1 as f2
7. *rm* f1 → remove the file f1
8. *head* –5 f1 → list first 5 lines of the file f1
   tail –5 f1 → list last 5 lines of the file f1

c) Working with Directories

1. *mkdir* elias → to create the directory elias
2. *cd* elias → to change the directory as elias
3. *rmdir* elias → to remove the directory elias
4. *pwd* → to display the path of the present working directory
5. *cd* → to go to the home directory
   cd .. → to go to the parent directory
   cd - → to go to the previous working directory
   cd / → to go to the root directory

d) File name substitution

1. *ls f?* → list files start with 'f' and followed by any one character
2. *ls* *.c → list files with extension 'c'
3. *ls* [gpy]et → list files whose first letter is any one of the character g, p or y and followed by the word et
4. *ls* [a-d,l-m]ring → list files whose first letter is any one of the character from a to d and l to m and followed by the word ring.

e) I/O Redirection

1. Input redirection

   wc –l < ex1 → To find the number of lines of the file 'ex1'

2. Output redirection

   who > f2 → the output of 'who' will be redirected to file f2

3. *cat* >> f1 → to append more into the file f1

f) Piping

Syntax : Command1 | command2

Output of the command1 is transferred to the command2 as input. Finally output of the command2 will be displayed on the monitor.

ex. *cat* f1 | *more* → list the contents of file f1 screen by screen

head –6 f1 |*tail* –2 → prints the 5^{th} & 6^{th} lines of the file f1.

g) Environment variables

1. *echo $HOME* → display the path of the home directory
2. *echo $PS1* → display the prompt string $
3. *echo $PS2* → display the second prompt string (> symbol by default)
4. *echo $LOGNAME* → login name
5. *echo $PATH* → list of pathname where the OS searches for an executable file

h) File Permission

-- chmod command is used to change the access permission of a file.

Method-1

Syntax : *chmod* [ugo] [+/-] [rwxa] filename

u : user, g : group, o : others
+ : Add permission - : Remove the permission
r : read, w : write, x : execute, a : all permissions

ex. *chmod* ug+rw f1
adding 'read & write' permissions of file f1 to both user and group members.

Method-2

Syntax : *chmod* octnum file1

The 3 digit octal number represents as follows

- first digit -- file permissions for the user
- second digit -- file permissions for the group
- third digit -- file permissions for others

Each digit is specified as the sum of following
4 – read permission, 2 – write permission, 1 – execute permission

ex. *chmod* 754 f1

it change the file permission for the file as follows

- read, write & execute permissions for the user ie; 4+2+1 = 7
- read, & execute permissions for the group members ie; 4+0+1 = 5
- only read permission for others ie; 4+0+0 = 4

Verified by

Faculty In-charge Sign :	**Date :**

Ex. No. 2b	FILTERS and ADMIN COMMANDS	Date :

FILTERS

1. cut

- Used to cut characters or fileds from a file/input

Syntax : **cut** **-c**chars filename
-ffieldnos filename

- By default, tab is the filed separator(delimiter). If the fileds of the files are separated by any other character, we need to specify explicitly by **–d** option

cut **-d**delimitchar **-f**fileds filname

2. grep

- Used to search one or more files for a particular pattern.

Syntax : **grep** pattern filename(s)

- Lines that contain the *pattern* in the file(s) get displayed
- pattern can be any regular expressions
- More than one files can be searched for a pattern

-v option displays the lines that do not contain the *pattern*
-l list only name of the files that contain the *pattern*
-n displays also the line number along with the lines that matches the *pattern*

3. sort

- Used to sort the file in order

Syntax : **sort** filename

- Sorts the data as text by default
- Sorts by the first filed by default

-r option sorts the file in descending order
-u eliminates duplicate lines
-o filename writes sorted data into the file *fname*
-tdchar sorts the file in which fileds are separated by *dchar*
-n sorts the data as number
+1n skip first filed and sort the file by second filed numerically

4. Uniq

- Displays unique lines of a sorted file

Syntax : **uniq** filename

-d option displays only the duplicate lines
-c displays unique lines with no. of occurrences.

5. diff

- Used to differentiate two files

Syntax : **diff** f1 f2
compare two files f1 & f2 and prints all the lines that are differed between f1 & f2.

Q1. Write a command to cut 5 to 8 characters of the file *f1*.
$

Q2. Write a command to display user-id of all the users in your system.
$

Q3. Write a command to check whether the user j*udith* is available in your system or not. (use grep)
$

Q4. Write a command to display the lines of the file *f1* starts with SRM.
$

Q5. Write a command to sort the file /etc/passwd in descending order
$

Q6. Write a command to display the unique lines of the sorted file *f21*. Also display the number of occurrences of each line.
$

Q7. Write a command to display the lines that are common to the files *f1* and *f2*.
$

SYSTEM ADMIN COMMANDS

INSTALLING SOFTWARE

To Update the package repositories

```
sudo apt-get update
```

To update installed software

```
sudo apt-get upgrade
```

To install a package/software

```
sudo apt-get install  <package-name>
```

To remove a package from the system

```
sudo apt-get remove <package-name>
```

To reinstall a package

```
sudo apt-get install  <package-name> --reinstall
```

Q8. Update the package repositories

Q9. Install the package “simplescreenrecorder”

Q10. Remove the package “simplescreenrecorder”

MANAGING USERS

- Managing users is a critical aspect of server management.
- In Ubuntu, the root user is disabled for safety.
- Root access can be completed by using the sudo command by a user who is in the “admin” group.
- When you create a user during installation, that user is added automatically to the admin group.

To add a user:

```
sudo adduser username
```

To disable a user:

```
sudo passwd -l username
```

To enable a user:

```
sudo passwd -u username
```

To delete a user:

```
sudo userdel –r username
```

To create a group:

```
sudo addgroup groupname
```

To delete a group:

```
sudo delgroup groupname
```

To create a user with group:

```
sudo adduser username groupname
```

To see the password expiry value for a user,

```
sudo chage -l username
```

To make changes:

```
sudo chage username
```

GUI Tool for user management

If you do not want to run the commands in terminal to manage users and groups, then you can install a GUI add-on .

```
sudo apt install gnome-system-tools
```

Once done, type

```
users-admin
```

Q11. Create a user ‘elias’. Login to the newly created user and exit.

Q12. Disable the user ‘elias’, try to login and enable again.

Verified by

Faculty In-charge Sign :	**Date :**

Ex. No. 3	SHELL PROGRAMS	Date :

How to run a Shell Script

- Edit and save your program using editor
- Add execute permission by *chmod* command
- Run your program using the name of your program

```
./program-name
```

Important Hints

- No space before and after the assignment operator Ex. `sum=0`
- *Single quote* ignores all special characters. Dollar sign, Back quote and Back slash are not ignored inside *Double quote*. *Back quote* is used as command substitution. *Back slash* is used to remove the special meaning of a character.
- Arithmetic expression can be written as follows : `i=$((i+1))` or `i=$(expr $i + 1)`
- Command line arguments are referred inside the programme as `$1, $2`, ..and so on
- `$*` represents all arguments, `$#` specifies the number of arguments
- `read` statement is used to get input from input device. Ex. `read a b`

Syntax for if statement

```
if [ condition ]
then
      ...
elif [ condition ]
then
      ...
else
      ...
fi
```

Syntax for case structure

```
case value in
pat1)   ...
                statement;;
pat2)   ...
                Statement;;
*)      ...
                Statement;;
esac
```

Syntax for for-loop

```
for var in list-of-values
do
          ...
          ...
done
```

Syntax for While loop

```
while command_t
do
          ...
          ...
done
```

Syntax for printf statement

```
printf  "string and format"  arg1  arg2 … …
```

- Break and continue statements functions similar to C programming
- Relational operators are –lt, -le, -gt, -ge, -eq,-ne
- Ex. (i>= 10) is written as [$i -ge 10]
- Logical operators (and, or, not) are -o, -a, !
- Ex. (a>b) && (a>c) is written as [$a –gt $b –a $a –gt $c]
- Two strings can be compared using = operator

Q1. Given the following values

num=10, x=*, y=`date` a="Hello, 'he said'"

Execute and write the output of the following commands

Command	Output
`echo num`	
`echo $num`	
`echo $x`	
`echo '$x'`	
`echo "$x"`	
`echo $y`	
`echo $(date)`	
`echo $a`	
echo \$num	
echo \$$num	

Q1. Find the output of the following shell scripts

```
$ vi  ex31
     echo Enter value for n
     read n
     sum=0
     i=1
     while [ $i -le $n ]
     do
            sum=$((sum+i))
            i=$((i+2))
     done
     echo Sum is $sum
```

Output :

Q2. Write a program to check whether the file has execute permission or not. If not, add the permission.

```
$ vi ex32
```

Q3. Write a shell script to list only the name of sub directories in the present working directory

```
$ vi ex33
```

Q4. Write a program to check all the files in the present working directory for a pattern (passed through command line) and display the name of the file followed by a message stating that the pattern is available or not available.

```
$ vi ex34
```

Verified by

Faculty In-charge Sign :	**Date :**

Ex. No. 4	PROCESS CREATION	Date :

Compilation of C Program

Step 1 : Open the terminal and edit your program and save with extension ".c"

Ex. nano test.c

Step 2 : Compile your program using gcc compiler

Ex. gcc test.c → Output file will be "a.out"

(or)

gcc –o test text.c → Output file will be "test"

Step 3 : Correct the errors if any and run the program

Ex. ./a.out (or) ./test

Syntax for process creation

int fork();

Returns 0 in child process and child process ID in parent process.

Other Related Functions

int getpid() → returns the current process ID

int getppid() → returns the parent process ID

wait() → makes a process wait for other process to complete

Virtual fork

vfork() is similar to fork but both processes shares the same address space.

Q1. Find the output of the following program

```
#include <stdio.h>
#include<unistd.h>
int main()
{
  int a=5,b=10,pid;
  printf("Before fork a=%d b=%d \n",a,b);
  pid=fork();

  if(pid==0)
  {
    a=a+1; b=b+1;
    printf("In child a=%d b=%d \n",a,b);
  }
  else
  {
    sleep(1);
    a=a-1; b=b-1;
    printf("In Parent a=%d b=%d \n",a,b);
  }
  return 0;
}
```

Output :

Q2. Rewrite the program in Q1 using vfork() and write the output

Q3. Calculate the number of times the text "SRMIST" is printed.

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

int main()
{
    fork();
    fork();
    fork();
    printf("SRMIST\n");
    return 0;
}
```

Output :

Q4. Complete the following program as described below :

The child process calculates the sum of odd numbers and the parent process calculate the sum of even numbers up to the number 'n'. Ensure the Parent process waits for the child process to finish.

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

int main()
{
  int pid,n,oddsum=0,evensum=0;

  printf("Enter the value of n : ",a);
  scanf("%d",&n);
  pid=fork();
  // Complete the program









  return 0;
}
```

Sample Output :

Enter the value of n : 10
Sum of odd numbers : 25
Sum of even numbers : 30

Q5. How many child processes are created for the following code?

Hint : Check with small values of 'n'.

```
for (i=0; i<n; i++)
        fork();
```

Output :

Q6. Write a program to print the Child process ID and Parent process ID in both Child and Parent processes

```
#include <stdio.h>
#include<unistd.h>
int main()
{









return 0;
}
```

Sample Output:

```
In Child Process
Parent Process ID        :      18
Child Process ID         :      20

In Parent Process
Parent Process ID        :      18
Child Process ID         :      20
```

Q7. How many child processes are created for the following code?

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

int main()
{
    fork();
    fork()&&fork()||fork();
    fork();
    printf("Yes ");
    return 0;
}
```

Output :

Verified by

Faculty In-charge Sign :	**Date :**

Ex. No. 5a	SIMPLE TASK AUTOMATION	Date :

Linux Cron utility is an effective way to schedule a routine background job at a specific time and/or day on an on-going basis. You can use this to schedule activities, either as one-time events or as recurring tasks.

Crontab Syntax `m h dom mon dow command`

`m` – The minute when the cron job will run (0-59)
`h` - a numeric value determining the hour when the tasks will run (0-23)
`dom` – Day of the Month when the cron job will run (1-31)
`mon` - The month when the cron job will run (1-12)
`dow` – Day Of the Week from 0-6 with Sunday being 0
`command`- The linux command you wish to execute

Scheduling of Tasks (For Ubuntu)

Step 1 : Open terminal and type the command
```
crontab -e
```
Step 2 : Choose the editor. Better to select `nano` editor
Step 3 : Edit the file based on the syntax given above
Step 4 : Save and Exit the file
Step 5 : Start cron daemon using the following command
```
systemctl start cron
```

Example of crontab entry
```
0 8 * * 1 echo Have a Good Week > >tmpfile
```
Every Monday 8:00 am the message "Have a Good Week" transferred to the file 'tmpfile'

Special Crontab Characters

`*` represents all possible value
`/` represents partial value. Ex. `*/10` in minute column specifies every 10 minutes
`-` represent range of values. Ex. `6-9` in hour column specifies 6am to 9 am
`,` (Comma) represent different set of values. Ex. `1,4` in month specifies Jan and Apr month

Q1. Schedule a task to display the following message on the monitor for every 2 minutes.

Q2. Schedule a task to take backup of your important file (say file `f1`) for every 30 minutes

Q3. Schedule a task to take backup of login information everyday 9:30am

Verified by

Faculty In-charge Sign :	Date :

Ex. No. 5b	OVERLAY CONCEPTS	Date :

Exec() System Call – Overlay Calling process and run new Program
The exec() system call replaces (**overwrites**) the current process with the new process image**.** The PID of the new process remains the same however code, data, heap and stack of the process are replaced by the new program.

There are 6 system calls in the family of exec().All of these functions mentioned below are layered on top of execve(), and they differ from one another and from execve() only in the way in which the program name, argument list, and environment of the new program are specified.

Syntax

```
int execl(const char* path, const char* arg, …)
int execlp(const char* file, const char* arg, …)
int execle(const char* path, const char* arg, …, char* const envp[])
int execv(const char* path, const char* argv[])
int execvp(const char* file, const char* argv[])
int execvpe(const char* file, const char* argv[], char *const envp[])
```

- The names of the first five of above functions are of the form **exec*XY***.
- X is either l or v depending upon whether arguments are given in the list format (arg0, arg1, ..., NULL) or arguments are passed in an array (vector).
- Y is either absent or is either a p or an e. In case Y is p, the PATH environment variable is used to search for the program. If Y is e, then the environment passed in *envp* array is used.
- In case of execvpe, X is v and Y is e. The execvpe function is a GNU extension. It is named so as to differentiate it from the execve system call.

Q1. Execute the Following Program and write the output

```
$vi ex51.c
     #include <stdio.h>
     #include<unistd.h>
     int main()
     {
       printf("Transfer to execlp function \n");
       execlp("head", "head","-2","f1",NULL);
       printf("This line will not execute \n");
       return 0;
     }
```

Output :

Why second printf statement is not executing? _____

_____

Q2. Rewrite question Q1 with execl() function. Pass the 3rd and 4th argument of the function execl() through command line arguments.

```
$vi ex52.c
```

Input : `./a.out -3 f1`
Output :

Q3. Rewrite question Q1 with execv() function.

```
$vi ex53.c
```

Output :

Q4. Rewrite question Q1 with execv() function.

```
$vi ex54.c
```

Output :

Verified by

Faculty In-charge Sign :	**Date :**

Ex. No. 6	PIPES	Date :

Pipe is a communication medium between two or more processes. The system call for creating pipe is

```
int pipe(int p[2]);
```

This system call would create a pipe for one-way communication i.e., it creates two descriptors, first one is connected to read from the pipe and other one is connected to write into the pipe.

Descriptor p[0] is for reading and p[1] is for writing. Whatever is written into p[1] can be read from p[0].

Q1. Write the output of the following program

```
#include <stdio.h>
#include<unistd.h>
#include<sys/wait.h>
int main()
{
  int p[2];
  char buff[25];
  pipe(p);
  if(fork()==0)
  {
    printf("Child : Writing to pipe \n");
    write(p[1],"Welcome",8);
    printf("Child Exiting\n");
  }
  else
  {
    wait(NULL);
    printf("Parent : Reading from pipe \n");
    read(p[0],buff,8);
    printf("Pipe content is : %s \n",buff);
  }
  return 0;
}
```

Output :

Implementing command line pipe using exec() family of functions00

Follow the steps to transfer the output of a process to pipe:

(i) Close the standard output descriptor

(ii) Use the following system calls, to take duplicate of output file descriptor of the pipe

```
int dup(int fd);
int dup2(int oldfd, int newfd);
```

(iii) Close the input file descriptor of the pipe

(iv) Now execute the process

Follow the steps to get the input from the pipe for a process:

(i) Close the standard input descriptor
(ii) Take the duplicate of input file descriptor of the pipe using dup() system call
(iii) Close the output file descriptor of the pipe
(iv) Now execute the process

Q2. Write a program to implement the following command line pipe using pipe() and dup()

```
ls –l | wc -l
```

Named pipe

Named pipe (also known as FIFO) is one of the inter process communication tool. The system for FIFO is as follows

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

mkfifo() makes a FIFO special file with name **pathname**. Here **mode** specifies the FIFO's permissions. The permission can be like : O_CREAT|0644

Open FIFO in read-mode (O_RDONLY) to read and write-mode (O_WRONLY) to write

Q3. Write the output of the following program

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

int main()
{
  char buff[25];
  int rfd,wfd;

  mkfifo("fif1",O_CREAT|0644);

  if (fork()==0)
  {
    printf("Child writing into FIFO\n");
    wfd=open("fif1",O_WRONLY);
    write(wfd,”Hello”,6);
  }
  else
  {
    rfd=open("fif1",O_RDONLY);
    read(rfd,buff,6);
    printf("Parent reads from FIFO : %s\n",buff);
  }
  return 0;
}
```

Output :

Verified by

Faculty In-charge Sign :	**Date :**

Ex. No. 7	MESSAGE QUEUE & SHARED MEMORY	Date :

Message Queue

Message queue is one of the interprocess communication mechanisms. here are two varieties of message queues, System V message queues and POSIX message queues. Both provide almost the same functionality but system calls for the two are different.

There are three system wide limits regarding the message queues. These are, `MSGMNI`, maximum number of queues in the system, `MSGMAX`, maximum size of a message in bytes and `MSGMNB`, which is the maximum size of a message queue. We can see these limits with the `ipcs -l` command

Include the following header files for system V message queues

```
<sys/msg.h>, <sys/ipc.h>, <sys/types.h>
```

System V Message Queue System Calls

To create a message queue,

```
int msgget (key_t key, int msg_flags);
```

key → Message queue identifier ex. (key_t)77

flags → `IPC_CREAT|0664` : to create a message queue with permission 0644

`IPC_CREATE|IPC_EXCL|0664` : to create message if doesn't exists

To control the message queue,

```
int msgctl(int msqid, int cmd, struct msqid_ds *buf);
```

msquid→ Message id returned by msgget()

cmd → `IPC_STAT` : to get the status of a message queue

`IPC_SET` : to change the properties of a message queue

`IPC_RMID` : to remove a message queue

To send a message into message queue,

```
int msgsnd(int msquid, const void *msgp, size_t msqsz, int
msgflg);
```

msquid→ Message id returned by msgget()

msgp → message to be sent. Buffer or message structure is used here.

```
struct buffer {
        int len;          // length of the message
        int mtype;        // message number
        char buf[50];     // buffer
    }x;
```

msqsz → size of the message

msgflg → `IPC_NOWAIT` or `IPC_WAIT` for blocking/non-blocking I/O

To receive a message from message queue,

```
int msgrcv(int msquid, void *msgp, size_t msqsz, long
msgtyp, int msgflg);
```

msqp → the buffer/message structure to receive the message

msgtyp → message number

msquid, msqsz, msgflg arguments are similar to msgsnd()

Q1. Write a program to send a message (pass through command line arguments) into a message queue. Send few messages with unique message numbers

```
$ vi ex71.c
```

Q2. Write a program to receive a particular message from the message queue. Use message number to receive the particular message

```
$ vi ex72.c
```

Linux Commands to control the Interprocess communication tools (Message queue/ semaphore/ shared memory)

ipcs → to list all IPC information
ipcs –l → to list the limits of each IPC tools
ipcs –q → to list message queues details
ipcs –s → to list semaphore details
ipcs –m → to list all shared memory details
ipcs –u → to get the current usage of IPC tools
ipcs –h → ipcs help

ipcrm –q <msgid> → to remove a message queue with message-id <msgid>
ipcrm –m <shmid> → to remove a shared memory
ipcrm –s <semid> → to remove a semaphore
ipcrm –h → ipcrm help

Shared memory

Inter Process Communication through shared memory is a concept where two or more process can access the common memory. And communication is done via this shared memory where changes made by one process can be viewed by another process.

Include the following header files for shared memory

```
<sys/ipc.h>,  <sys/shm.h>, <sys/types.h>
```

System V Shared memory System Calls

To create a shared memory,

```
int shmget(key_t key, size_t size, int shmflg)
```

key → shared memory id
size → shared memory size in bytes
shmflg → `IPC_CREATE|0664` : to create a new shared memory segment
`IPC_EXCL|IPC_CREAT|0664` : to create new segment and the call fails, if the segment already exists

To attach the shared memory segment to the address space of the calling process

```
void * shmat(int shmid, const void *shmaddr, int shmflg)
```

shmid → Shared memory id returned by shmget()
shmaddr → the attaching address. If shmaddr is NULL, the system by default chooses the suitable address. If shmaddr is not NULL and SHM_RND is specified in shmflg, the attach is equal to the address of the nearest multiple of SHMLBA (Lower Boundary AddressShmflg → SHM_RND (rounding off address to SHMLBA) or SHM_EXEC (allows the contents of segment to be executed) or SHM_RDONLY (attaches the segment for read-only purpose, by default it is read-write) or SHM_REMAP (replaces the existing mapping in the range specified by shmaddr and continuing till the end of segment)

To control the shared memory segment,

```
int shmctl(int shmid, int cmd, struct shmid_ds *buf)
```

To detach the shared memory segment from the address space of the calling process

```
int shmdt(const void *shmaddr)
```

shmaddr → address of the shared memory to detach

Q3. Write a program to do the following:

- Create two processes, one is for writing into the shared memory (shm_write.c) and another is for reading from the shared memory (shm_read.c)
- In the shared memory, the writing process, creates a shared memory of size 1K (and flags) and attaches the shared memory
- The write process writes the data read from the standard input into the shared memory. Last byte signifies the end of buffer
- Read process would read from the shared memory and write to the standard output

```
$ vi ex73.c
```

Verified by

Faculty In-charge Sign :	**Date :**

Ex. No. 8	SCHEDULING ALGORITHMS	Date :

1. FCFS Scheduling Algorithm

Given n processes with their burst times, the task is to find average waiting time and average turn around time using FCFS scheduling algorithm. First in, first out (FIFO), also known as first come, first served (FCFS), is the simplest scheduling algorithm. FIFO simply queues processes in the order that they arrive in the ready queue.

In this, the process that comes first will be executed first and next process starts only after the previous gets fully executed. Here we are considering that arrival time for all processes is 0.

Turn Around Time: Time Difference between completion time and arrival time.
Turn Around Time = Completion Time – Arrival Time

Waiting Time(W.T): Time Difference between turn around time and burst time.
Waiting Time = Turn Around Time – Burst Time

Algorithm:

Step 1. Input the processes along with their burst time (bt).

Step 2. Find waiting time (wt) for all processes.

Step 3. As first process that comes need not to wait so
waiting time for process 1 will be 0 i.e. wt[0] = 0.

Step 4. Find waiting time for all other processes i.e. for
process i ->
wt[i] = bt[i-1] + wt[i-1]

Step 5. Find *turnaround time = waiting_time + burst_time*
for all processes.

Step 6. Find *average waiting time = total_waiting_time / no_of_processes*

Step 7. Similarly, find *average turnaround time =*
total_turn_around_time / no_of_processes.

Input : Processes Numbers and their burst times

Output : Process-wise burst-time, waiting-time and turnaround-time
Also display Average-waiting time and Average-turnaround-time

Q1. Write a program to implement FCFS Scheduling algorithm

```
#include <stdio.h>
//Write the program here
```

2. ROUND ROBIN Scheduling Algorithm

In this algorithm, each process is assigned a fixed time slot in a cyclic way

Algorithm:

Step 1. Create an array **rem_bt[]** to keep track of remaining burst time of processes. This array is initially a copy of bt[] (burst times array)

Step 2. Create another array **wt[]** to store waiting timesof processes. Initialize this array as 0.

Step 3. Initialize time : t = 0

Step 4. Keep traversing the all processes while all processes are not done. Do following for i'th process if it is not done yet.

a) if rem_bt[i] > quantum

```
        t = t + quantum
        rem_bt[i] -= quantum;
    else // Last cycle for this process
        t = t + rem_bt[i];
        wt[i] = t - bt[i]
        rem_bt[i] = 0;
```

Step 5. Turnaround time tat[i] = wt[i] + bt[i]. for i^{th} process

Step 6. Find average waiting time and average turnaround time

Input : Processes Numbers and their burst times, time-quantum

Output : Process-wise burst-time, waiting-time and turnaround-time

Also display Average-waiting time and Average-turnaround-time

Q2. Write a program to implement Round Robin Scheduling algorithm

```
#include <stdio.h>
//Write the program here
```

Verified by

Faculty In-charge Sign :	**Date :**

Ex. No. 9	PROCESS SYNCHRONIZATION	Date :

Semaphore

Semaphore is used to implement process synchronization. This is to protect critical region shared among multiples processes.

SYSTEM V SEMAPHORE SYSTEM CALLS

Include the following header files for System V semaphore

```
<sys/ipc.h>,  <sys/sem.h>, <sys/types.h>
```

To create a semaphore array,

```
int semget(key_t key, int nsems, int semflg)
```

key → semaphore id
nsems → no. of semaphores in the semaphore array
semflg → `IPC_CREATE|0664` : to create a new semaphore
`IPC_EXCL|IPC_CREAT|0664` : to create new semaphore and the call fails if the semaphore already exists

To perform operations on the semaphore sets viz., allocating resources, waiting for the resources or freeing the resources,

```
int semop(int semid, struct sembuf *semops, size_t nsemops)
```

semid → semaphore id returned by semget()
nsemops → the number of operations in that array
semops → The pointer to an array of operations to be performed on the semaphore set. The structure is as follows

```
struct sembuf {
     unsigned short sem_num;    /* Semaphore set num */
     short sem_op;              /* Semaphore operation */
     short sem_flg;  /*Operation flags, IPC_NOWAIT, SEM_UNDO */
};
```

Element, sem_op, in the above structure, indicates the operation that needs to be performed −

- If sem_op is –ve, allocate or obtain resources. Blocks the calling process until enough resources have been freed by other processes, so that this process can allocate.
- If sem_op is zero, the calling process waits or sleeps until semaphore value reaches 0.
- If sem_op is +ve, release resources.

To perform control operation on semaphore,

```
int semctl(int semid, int semnum, int cmd,…);
```

semid → identifier of the semaphore returned by semget()
semnum → semaphore number
cmd → the command to perform on the semaphore. Ex. GETVAL, SETVAL
semun → value depends on the cmd. For few cases, this is not applicable.

Q1. Execute and write the output of the following program for *mutual exclusion* using system V semaphore

```
#include<sys/ipc.h>
#include<sys/sem.h>
int main()
{
 int pid,semid,val;
 struct sembuf sop;

 semid=semget((key_t)6,1,IPC_CREAT|0666);

 pid=fork();

 sop.sem_num=0;
 sop.sem_op=0;
 sop.sem_flg=0;

 if (pid!=0)
 {
   sleep(1);
   printf("The Parent waits for WAIT signal\n");
   semop(semid,&sop,1);
   printf("The Parent WAKED UP & doing her job\n");
   sleep(10);
   printf("Parent Over\n");
 }
 else
 {
   printf("The Child sets WAIT signal & doing her job\n");
   semctl(semid,0,SETVAL,1);
   sleep(10);
   printf("The Child sets WAKE signal & finished her job\n");
   semctl(semid,0,SETVAL,0);
   printf("Child Over\n");
 }
 return 0;
}
```

Output :

THREAD

Thread is the segment of a process means a process can have multiple threads and these multiple threads are contained within a process. The thread takes less time to terminate as compared to the process but unlike the process, threads do not isolate.

THREAD FUNCTIONS

The header file for POSIX thread functions is `pthread.h`. To execute the c file with thread, do as follows:

```
gcc -pthread file.c      (or)
gcc -lpthread file.c
```

To create a new thread,

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

- **thread:** pointer to an unsigned integer value that returns the thread id of the thread created.
- **attr:** pointer to a structure that is used to define thread attributes like detached state, scheduling policy, stack address, etc. Set to NULL for default thread attributes.
- **start_routine:** pointer to a subroutine that is executed by the thread. The return type and parameter type of the subroutine must be of type void *. The function has a single attribute but if multiple values need to be passed to the function, a struct must be used.
- **arg:** pointer to void that contains the arguments to the function defined in the earlier argument

To terminate a thread

```
void pthread_exit(void *retval);
```

This method accepts a mandatory parameter **retval** which is the pointer to an integer that stores the return status of the thread terminated. The scope of this variable must be global so that any thread waiting to join this thread may read the return status.

To wait for the termination of a thread.

```
int pthread_join(pthread_t th, void **thread_return);
```

This method accepts following parameters:

- **th:** thread id of the thread for which the current thread waits.
- **thread_return:** pointer to the location where the exit status of the thread mentioned in th is stored.

POSIX SEMAPHORE

The POSIX system in Linux presents its own built-in semaphore library. To use it, we have to include `semaphore.h` and compile the code by linking with `-lpthread -lrt`

To lock a semaphore or wait

```
int sem_wait(sem_t *sem);
```

To release or signal a semaphore

```
int sem_post(sem_t *sem);
```

To initialize a semaphore

```
sem_init(sem_t *sem, int pshared, unsigned int value);
```

Where,

`sem` : Specifies the semaphore to be initialized.

`pshared` : This argument specifies whether or not the newly initialized semaphore is shared between processes/threads. A non-zero value means the semaphore is shared between processes and a value of zero means it is shared between threads.

`value` : Specifies the value to assign to the newly initialized semaphore.

To destroy a semaphore

```
sem_destroy(sem_t *mutex);
```

Q2. Execute and write the output of the following program for *mutual exclusion* using POSIX semaphore and threads.

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

sem_t mutex;

void* thread(void* arg)
{
      //wait
      sem_wait(&mutex);
      printf("\nEntered..\n");

      //critical section
      sleep(4);

      //signal
      printf("\nJust Exiting...\n");
      sem_post(&mutex);
}
```

```
int main()
{
      sem_init(&mutex, 0, 1);
      pthread_t t1,t2;
      pthread_create(&t1,NULL,thread,NULL);
      sleep(2);
      pthread_create(&t2,NULL,thread,NULL);
      pthread_join(t1,NULL);
      pthread_join(t2,NULL);
      sem_destroy(&mutex);
      return 0;
}
```

Output:

Verified by

Faculty In-charge Sign :	**Date :**

Ex. No. 10a	READER-WRITER PROBLEM	Date :

Problem Statement

There is a shared resource which should be accessed by multiple processes. There are two types of processes in this context. They are **reader** and **writer**. Any number of **readers** can read from the shared resource simultaneously, but only one **writer** can write to the shared resource. When a **writer** is writing data to the resource, no other process can access the resource. A **writer** cannot write to the resource if there are non-zero number of readers accessing the resource at that time.

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

sem_t x,y;
pthread_t tid;
pthread_t writerthreads[50],readerthreads[50];
int readercount;

void *reader(void* param)
{
//Write coding for reader thread









}
```

```
void *writer(void* param)
{
//Write coding for writer thread









}
int main()
{
    int n2,i;
    printf("Enter the number of readers:");
    scanf("%d",&n2);
    int n1[n2];
    sem_init(&x,0,1);
    sem_init(&y,0,1);
    for(i=0;i<n2;i++)
    {
        pthread_create(&writerthreads[i],NULL,reader,NULL);
        pthread_create(&readerthreads[i],NULL,writer,NULL);
    }
    for(i=0;i<n2;i++)
    {
        pthread_join(writerthreads[i],NULL);
        pthread_join(readerthreads[i],NULL);
    }

}
```

Output:

Verified by

Faculty In-charge Sign :	**Date :**

Ex. No. 10b	DINING PHILOSOPHER PROBLEM	Date :

Problem Statement

The dining philosophers problem states that there are 5 philosophers sharing a circular table and they eat and think alternatively. There is a bowl of rice for each of the philosophers and 5 chopsticks. A philosopher needs both their right and left chopstick to eat. A hungry philosopher may only eat if there are both chopsticks available. Otherwise a philosopher puts down their chopstick and begin thinking again.

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

sem_t room;
sem_t chopstick[5];

void * philosopher(void * num)
{
//Write coding for Main Philosopher thread









}

void eat(int phil)
{
      printf("\nPhilosopher %d is eating",phil);
```

```
}


int main()
{
      int i,a[5];
      pthread_t tid[5];

      sem_init(&room,0,4);

      for(i=0;i<5;i++)
            sem_init(&chopstick[i],0,1);

      for(i=0;i<5;i++){
            a[i]=i;
            pthread_create(&tid[i],NULL,philosopher,(void *)&a[i]);
      }
      for(i=0;i<5;i++)
            pthread_join(tid[i],NULL);
}
```

Output:

Verified by

Faculty In-charge Sign :	**Date :**

Ex. No. 11a	SHELL CODE ANALYSER	Date :

`Shellcheck` is a static analysis tool (or linter) for shell scripts. It detects various types of errors, gives suggestions and warnings for a shell script. It points out syntax issues, symantic problems that cause shell script to behave weird and some other corner cases.

Installation of Shellcheck in Ubuntu

```
sudo apt-get install shellcheck
```

Checking Script with Shellcheck

Edit a shell script (ex. fact.sh), give execute permission and check the script for syntax/semantic errors using shellcheck as follows:

```
shellcheck fact.sh
```

Q1. Check any one of your shellscript for Quotes, syntax errors using Shellcheck

Output:

Verified by

Faculty In-charge Sign :	Date :

Ex. No. 11b	GNU DEBUGGER	Date :

Debug C Programs using gdb debugger

Step 1 : Compile C program with debugging option –g
Ex. gcc –g test.c

Step 2 : Launch gdb. You will get gdb prompt
Ex. gdb a.out

Step 3 : Step break points inside C program
Ex. (gdb) b 10
Break points set up at line number 10. We can have any number of break points

Step 4 : Run the program inside gdb
Ex. (gdb) r

Step 5 : Print variable to get the intermediate values of the variables at break point
Ex. (gdb) p i → Prints the value of the variable 'i'

Step 6 : Continue or stepping over the program using the following gdb commands
c → continue till the next break
n → Execute the next line. Treats function as single statement
s → Similar to 'n' but executes function statements line by line
l → List the program statements

Step 7 : Quit the debugger
(gdb) q

Q1. Take any one c program and debug the error using GNU debugger

Output:

Verified by

Faculty In-charge Sign :	Date :

Ex. No. 11c	BINARY FILE ANALYSER	Date :

Following are some of the commands used to analyse the binary files in linux

1. `file` command is used to determine the file type

```
$ file /bin/ls
```

2. `ldd` command is used to find the shared object dependencies

```
$ ldd /bin/ls
```

3. `ltrace` command displays all the functions that are being called at run time from the library

```
$ ltrace /bin/ls
```

4. `hexdump` command is to display file contents in ASCII, decimal, hexadecimal, or octal

```
$hexdump -C /bin/ls | head
```

5. `readelf` command is used to display information about ELF (Executable and Linkable File Format) files

```
$readelf -h /bin/ls
```

6. `objdump` command is used to display information from an object file.

```
$ objdump -d /bin/ls |head
```

7. `strace` command is used to trace system calls and signals.

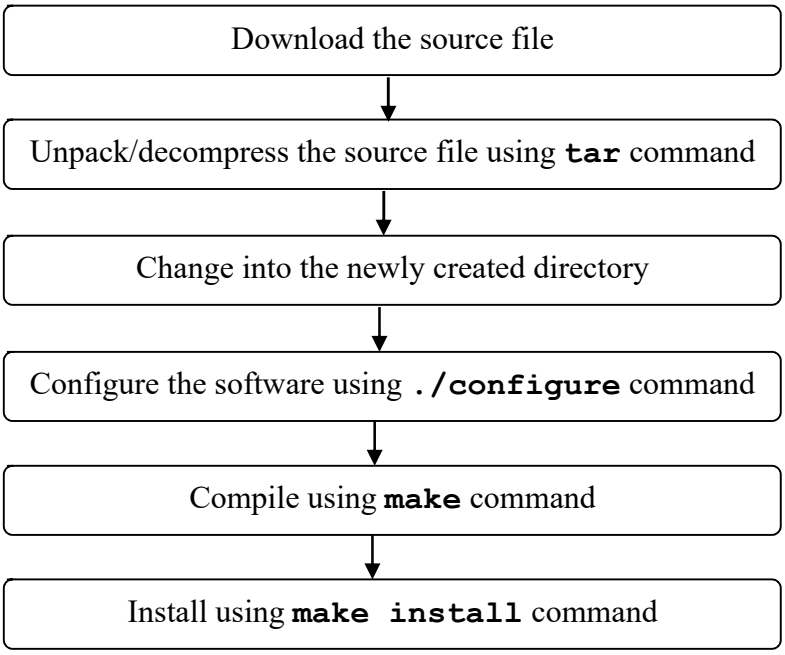
```
$ strace -f /bin/ls
```

Verified by

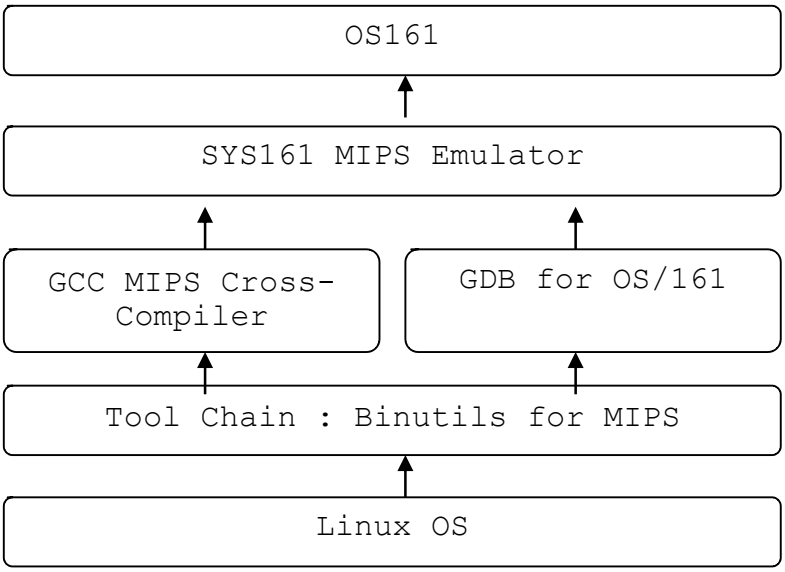
Faculty In-charge Sign :	**Date :**

Ex. No. 12	STUDY OF OS161	Date :

STEPS TO BUILD SOFTWARE FROM SOURCE FILE



BUILD SOFTWARE FRAMEWORK FOR OS/161



OS/161 INSTALLATION

Prerequisites

- Linux desktop with UBUNTU Version 12.04 or later.
- Internet connections to download and install packages

Pre Installation Steps

1. Install the packages gettext (to translate native language statements into English) , textinfo (to transalate source code into other formats) and libncurses5-dev (allows users to write text-base GUI)

```
sudo apt-get install gettext
sudo apt-get install texinfo
sudo apt-get install libncurses5-dev
```

2. Include the paths `$HOME/sys161/bin` and `$HOME/sys161/tools/bin` into PATH environment variable. Add the following line at the end of the file **`.bashrc`**

```
export PATH=$HOME/sys161/bin:$HOME/sys161/tools/bin:$PATH
```

Now logout and login to get the PATH updated. You can check the current setting of the `PATH` environment variable using the command

```
printenv PATH
```

Installation Steps

STEP 1: Download the following source codes one by one

- **Binutils for MIPS**
  http://www.student.cs.uwaterloo.ca/~cs350/os161_repository/os161-binutils.tar.gz

- **GCC MIPS Cross-Compiler**
  http://www.student.cs.uwaterloo.ca/~cs350/os161_repository/os161-gcc.tar.gz

- **GDB for Use with OS/161**
  http://www.student.cs.uwaterloo.ca/~cs350/os161_repository/os161-gdb.tar.gz

- **bmake for use with OS/161**
  http://www.student.cs.uwaterloo.ca/~cs350/os161_repository/os161-bmake.tar.gz

- **mk for use with OS/161**
  http://www.student.cs.uwaterloo.ca/~cs350/os161_repository/os161-mk.tar.gz

- **sys/161**
  http://www.student.cs.uwaterloo.ca/~cs350/os161_repository/sys161.tar.gz

- **OS/161**
  http://www.student.cs.uwaterloo.ca/~cs350/os161_repository/os161.tar.gz

Note : `bmake` and `mk` utilities are BSD make utilities used for OS161

STEP 2: Build and Install the Binary Utilities (Binutils)

Unpack the binutils archive:

```
tar -xzf os161-binutils.tar.gz
```

Move into the newly-created directory:

```
cd binutils-2.17+os161-2.0.1
```

Configure binutils:

```
./configure --nfp --disable-werror --target=mips-
harvard-os161 --prefix=$HOME/sys161/tools
```

Make binutils:

```
make
```

Finally, once **make** has succeeded, install the binutils into their final location:

```
make install
```

This will create the directory **$HOME/sys161/tools/** and populate it.

Step 3: Install the GCC MIPS Cross-Compiler

Unpack the gcc archive:

```
tar -xzf os161-gcc.tar.gz
```

Move into the newly-created directory:

```
cd gcc-4.1.2+os161-2.0
```

Configure gcc

```
./configure -nfp --disable-shared --disable-threads --
disable-libmudflap --disable-libssp --target=mips-
harvard-os161 --prefix=$HOME/sys161/tools
```

Make it and install it:

```
make
make install
```

Step 4: Install GDB

Unpack the gdb archive:

```
tar -xzf os161-gdb.tar.gz
```

Move into the newly-created directory:

```
cd gdb-6.6+os161-2.0
```

Configure gdb

```
./configure --target=mips-harvard-os161 --
prefix=$HOME/sys161/tools --disable-werror
```

Make it and install it:

```
make
make install
```

Step 5: Install bmake

Unpack the bmake archive:

```
tar -xzf os161-bmake.tar.gz
```

Move into the newly-created directory:

```
cd bmake
```

Unpack mk within the bmake directory:

```
tar -xzf ../os161-mk.tar.gz
```

Run the bmake bootstrap script

```
./boot-strap --prefix=$HOME/sys161/tools
```

As the **boot-strap** script finishes, it should print a list of commands that you can run to install bmake under **$HOME/sys161/tools.** The list should look something like this:

```
mkdir -p /home/kmsalem/sys161/tools/bin
cp /home/kmsalem/bmake/Linux/bmake
/home/kmsalem/sys161/tools/bin/bmake-20101215
rm -f /home/kmsalem/sys161/tools/bin/bmake
ln -s bmake-20101215
/home/kmsalem/sys161/tools/bin/bmake
mkdir -p /home/kmsalem/sys161/tools/share/man/cat1
cp /home/kmsalem/bmake/bmake.cat1
/home/kmsalem/sys161/tools/share/man/cat1/bmake.1
sh /home/kmsalem/bmake/mk/install-mk
/home/kmsalem/sys161/tools/share/mk
```

Run the commands printed by boot-strap in the order in which they are listed in your terminal screen.

Step 6: Set Up Links for Toolchain Binaries

```
mkdir $HOME/sys161/bin
cd $HOME/sys161/tools/bin

sh -c 'for i in mips-*; do ln -s $HOME/sys161/tools/bin/$i
$HOME/sys161/bin/cs350-`echo $i | cut -d- -f4-`; done'

ln -s $HOME/sys161/tools/bin/bmake $HOME/sys161/bin/bmake
```

When you are finished with these steps, a listing of the directory **$HOME/sys161/bin** should look similar to this:

```
bmake@            cs350-gcc@        cs350-ld@
cs350-run@
cs350-addr2line@  cs350-gcc-4.1.2@  cs350-nm@
cs350-size@
```

```
cs350-ar@          cs350-gccbug@      cs350-objcopy@
cs350-strings@
cs350-as@          cs350-gcov@        cs350-objdump@
cs350-strip@
cs350-c++filt@     cs350-gdb@         cs350-ranlib@
cs350-cpp@         cs350-gdbtui@      cs350-readelf@
```

Step 7: Build and Install the sys161 Simulator

Unpack the sys161 archive:

```
tar -xzf sys161.tar.gz
```

Move into the newly-created directory:

```
cd sys161-1.99.06
```

Next, configure sys161:

```
./configure --prefix=$HOME/sys161 mipseb
```

Build sys161 and install it:

```
make
make install
```

Finally, set up a link to a sample sys161 configuration file

```
cd $HOME/sys161
ln -s share/examples/sys161/sys161.conf.sample
sys161.conf
```

Step 8: Install OS/161

First, create a directory to hold the OS/161 source code, your compiled OS/161 kernels, and related test programs.

```
cd $HOME
mkdir cs350-os161
```

Next, move the OS/161 archive into your new directory and unpack it:

```
mv os161.tar.gz cs350-os161
cd cs350-os161
tar -xzf os161.tar.gz
```

This will create a directory called **os161-1.99** (under **cs350-os161**) containing the OS/161 source code. You should now be able build, install, and run an OS/161 kernel and related application and test programs by following steps.

Step 9: Configure OS/161 and Build the OS/161 Kernel

The next step is to configure OS/161 and compile the kernel. From the **cs350-os161** directory, do the following:

```
cd os161-1.99
./configure --ostree=$HOME/cs350-os161/root --
toolprefix=cs350-
```

```
cd kern/conf
./config ASST0
cd ../compile/ASST0
bmake depend
bmake
bmake install
```

Step 10: Build the OS/161 User-level Programs

Next, build the OS/161 user level utilities and test programs:

```
cd $HOME/cs350-os161/os161-1.99
bmake
bmake install
```

Step 11: Try Running OS/161

You should now be able to use the SYS/161 simulator to run the OS/161 kernel that you built and installed. The SYS/161 simulator requires a configuration file in order to run. To obtain one, do this:

```
cd $HOME/cs350-os161/root
cp $HOME/sys161/sys161.conf sys161.conf
sys161 kernel-ASST0
```

You should see some output that looks something like this:

```
sys161: System/161 release 1.99.06, compiled Aug 23
2013 10:23:34

OS/161 base system version 1.99.05
Copyright (c) 2000, 2001, 2002, 2003, 2004, 2005, 2008,
2009
President and Fellows of Harvard College.  All rights
reserved.

Put-your-group-name-here's system version 0 (ASST0 #1)

316k physical memory available
Device probe...
lamebus0 (system main bus)
emu0 at lamebus0
ltrace0 at lamebus0
ltimer0 at lamebus0
beep0 at ltimer0
rtclock0 at ltimer0
lrandom0 at lamebus0
random0 at lrandom0
lhd0 at lamebus0
lhd1 at lamebus0
lser0 at lamebus0
con0 at lser0
```

```
cpu0: MIPS r3000
OS/161 kernel [? for menu]:
```

The last line is a command prompt from the OS/161 kernel. For now, just enter the command **`q`** to shut down the simulation and return to your shell. After logging out, to get back again into OS/161 just follow **STEP 12**.

Verified by

Faculty In-charge Sign :	**Date :**