



D Y PATIL

— RAMRAO ADIK
INSTITUTE OF —

TECHNOLOGY

NAVI MUMBAI

***Department of Computer Science and
Engineering***

**Programme: B.Tech Artificial Intelligence and Machine Learning
Lab Manual**

Second Year Semester-IV

Course Name: Operating System Lab

Even Semester

Institutional Vision and Mission

Our Vision

To foster and permeate higher and quality education with value added engineering, technology programs, providing all facilities in terms of technology and platforms for all round development with societal awareness and nurture the youth with international competencies and exemplary level of employability even under highly competitive environment so that they are innovative adaptable and capable of handling problems faced by our country and world at large.

Our Mission

The Institution is committed to mobilize the resources and equip itself with men and materials of excellence thereby ensuring that the Institution becomes pivotal center of service to Industry, academia, and society with the latest technology. RAIT engages different platforms such as technology enhancing Student Technical Societies, Cultural platforms, Sports excellence centers, Entrepreneurial Development Center and Societal Interaction Cell. To develop the college to become an autonomous Institution & deemed university at the earliest with facilities for advanced research and development programs on par with international standards. To invite international and reputed national Institutions and Universities to collaborate with our institution on the issues of common interest of teaching and learning sophistication.

Our Quality Policy

ज्ञानधीनं जगत् सर्वम् ।

Knowledge is supreme.

Our Quality Policy

It is our earnest endeavour to produce high quality engineering professionals who are innovative and inspiring, thought and action leaders, competent to solve problems faced by society, nation and world at large by striving towards very high standards in learning, teaching and training methodologies.

Our Motto: If it is not of quality, it is NOT RAIT!

Departmental Vision and Mission

Vision

To excel in emerging fields of Computer Science and Engineering by imparting knowledge, practical skills, and core human values, transforming students into valuable contributors capable of driving innovation through advanced computing in real-world situations

Mission

M1: To promote academic excellence by providing a rigorous curriculum, encouraging critical thinking, and supporting ongoing learning in emerging fields, thereby contributing to the advancement of computing.

M2: To create a learning environment that prioritizes the practical application of knowledge, ethical conduct, and effective communication, preparing graduates to face the challenges of the constantly evolving tech industry.

M3: To broaden the scope of knowledge by supporting interdisciplinary research, fostering collaborations with industry and academic institutions, and promoting publication of research findings.

Departmental Program Educational Objectives (PEOs)

Program Educational Objectives (PEOs)

PEO 1:

Graduates will apply interdisciplinary approaches to develop and implement AI based solutions that require expertise in multiple domains

PEO 2:

Graduates will engage in lifelong learning to adapt to evolving AI and data science technologies While showcasing strong professional skills and ethical standards in leading industries and organizations.

PEO 3:

Graduates will be equipped to perform innovative research in the field of Data Sciences and Artificial Intelligence, significantly contributing to the advancement of knowledge and solving emerging challenges in the field.

Departmental Program Outcomes (POs)

PO1: Engineering knowledge: Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.

PO2: Problem analysis: Identify, formulate, review research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.

PO3 : Design/development of solutions: Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations.

PO4: Conduct investigations of complex problems: Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.

PO5: Modern tool usage: Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modelling to complex engineering activities with an understanding of the limitations.

PO6: The engineer and society: Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.

PO7: Environment and sustainability: Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.

PO8: Ethics: Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.

PO9: Individual and team work: Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.

PO10 : Communication: Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.

PO11 : Project management and finance: Demonstrate knowledge and understanding of the engineering and management principles and apply these to one's own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.

PO12 : Life-long learning: Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.

Program Specific Outcomes: PSO

PSO 1:

Graduates will be able to design, develop, and implement intelligent systems, leveraging knowledge of Computer science and Artificial Intelligence

PSO 2:

Graduates will be able to apply artificial intelligence based tools and techniques to solve complex problems across various industries.

PSO 3:

Graduates will be proficient in using advanced mathematical and statistical techniques to develop and validate AIML models, staying abreast of the latest research trends and technologies.

Index

Sr. No.	Contents	Page No.
1.	List of Experiments	8
2.	Course Objective, Course Outcome & Experiment Plan	9
3.	CO-PO ,CO-PSO Mapping	11
4.	Study and Evaluation Scheme	14
5.	Experiment No. 1	15
6.	Experiment No. 2	23
7.	Experiment No. 3	27
8.	Experiment No. 4	31
9.	Experiment No. 5	36
10.	Experiment No. 6	40
11.	Experiment No. 7	44
12.	Experiment No. 8	48
13.	Experiment No. 9	54
14.	Experiment No. 10	60

List of Experiments

Sr. No.	Experiments Name
1	Explore Linux Commands Explore usage of basic Linux Commands and system calls for file, directory and process management. For eg: (mkdir, chdir, cat, ls, chown, chmod, chgrp, ps etc. system calls: open, read, write, close, getpid, setpid, getuid, getgid, getegid, geteuid. sort, grep, awk, etc.)
2	Linux shell script Write shell scripts to do the following: <ul style="list-style-type: none">a. Display OS version, release number, kernel versionb. Display top 10 processes in descending orderc. Display processes with highest memory usage.d. Display current logged in user and log name.e. Display current shell, home directory, operating system type, current path setting, current working directory.
3	Linux- API Implement basic commands of linux like ls, cp, mv and others using kernel APIs.
4	Linux- Process Create a child process in Linux using the fork system call. From the child process obtain the process ID of both child and parent by using getpid and getppid system call.
5	Process Management: Scheduling Write a program to implement the concept of non-pre-emptive and Pre-emptive scheduling algorithms and demonstrate using CPU OS simulator.
6	Process Management: Synchronization Write a C program to implement solution of Producer consumer problem through Semaphore
7	Process Management: Deadlock Write a program to demonstrate the concept of deadlock avoidance through Banker's Algorithm

8	Memory Management Write a program to demonstrate the concept of dynamic partitioning placement algorithms i.e. Best Fit, First Fit, Worst-Fit etc.
9	Memory Management: Virtual Memory Write a program in C demonstrate the concept of page replacement policies for handling page faults eg: FIFO, LRU etc.
10	File Management & I/O Management Write a C program to simulate file organization of multi-level directory structure. c. Write a program in C to do disk scheduling - FCFS, SCAN, C-SCAN

Course Objective, Course Outcome &Experiment Plan

Course Objective:

1	To gain practical experience with designing and implementing concepts of operating systems such as system calls, CPU scheduling, process management, memory management, file systems and deadlock handling using C language in Linux environment
2	To familiarize students with the architecture of Linux OS
3	To provide necessary skills for developing and debugging programs in Linux environment
4	To learn programmatically to implement simple operation system mechanisms

Course Outcomes:

CO1	Demonstrate basic Operating system Commands, Shell scripts, System Calls and API wrt Linux
CO2	Implement various process scheduling algorithms and evaluate their performance
CO3	Implement and analyze concepts of synchronization and deadlocks
CO4	Implement various Memory Management techniques and evaluate their performance
CO5	Implement and analyze concepts of virtual memory
CO6	Demonstrate and analyze concepts of file management and I/O management techniques

Experiment Plan:

Module No.	Week No.	Experiments Name	Course Outcome	Weightage
1	W1	Explore Linux Commands Explore usage of basic Linux Commands and system calls for file, directory and process management. For eg: (mkdir, chdir, cat, ls, chown, chmod, chgrp, ps etc. system calls: open, read, write, close, getpid, setpid, getuid, getgid, getegid, geteuid. sort, grep, awk, etc.)	CO1	03
2	W2	Linux shell script Write shell scripts to do the following: a. Display OS version, release number, kernel version b. Display top 10 processes in descending order c. Display processes with highest memory usage. d. Display current logged in user and log name. e. Display current shell, home directory, operating system type, current path setting, current working directory.	CO1	03
3	W3	Linux- API Implement basic commands of linux like ls, cp, mv and others using kernel APIs.	CO1	04
4	W4	Linux- Process Create a child process in Linux using the fork system call. From the child process obtain the process ID of both child and parent by using getpid and getppid system call.	CO2	05
5	W5	Process Management: Scheduling Write a program to implement the concept of non-pre-emptive and Pre-emptive scheduling algorithms and demonstrate using CPU OS simulator.	CO2	05
6	W6	Process Management: Synchronization	CO3	02

		Write a C program to implement solution of Producer consumer problem through Semaphore		
7	W7	Process Management: Deadlock Write a program to demonstrate the concept of deadlock avoidance through Banker's Algorithm	CO3	03
8	W8	Memory Management Write a program to demonstrate the concept of dynamic partitioning placement algorithms i.e. Best Fit, First Fit, Worst-Fit etc.	CO4	03
9	W9	Memory Management: Virtual Memory Write a program in C demonstrate the concept of page replacement policies for handling page faults eg: FIFO, LRU etc.	CO5	04
10	W10	File Management & I/O Management Write a program in C to do disk scheduling - FCFS, SCAN, C-SCAN	CO6	04

Subj ect Weig ht	Course Outcomes (Weightage-100%)		Program Outcomes											
			PO 1	PO 2	PO 3	PO 4	PO 5	PO 6	PO 7	PO 8	PO 9	PO 10	PO 11	PO 12
PRATI CAL 100%	CO1	Demonstrate basic Operating system Commands, Shell scripts, System Calls and API wrt Linux	1	3	3	2								1
	CO2	Implement various process scheduling algorithms and evaluate their performance	2	2	3		1		2					
	CO3	Implement and analyze concepts of synchronization and deadlocks	2	2	3		1		2					
	CO4	Implement various Memory Management techniques and evaluate their performance	1	2	2	1	1		2					1
	CO5	Implement and analyze concepts of virtual memory	1	2	2				2				2	1
	CO6	Demonstrate and analyze concepts of file management and I/O management techniques	1	2	2		2				1		1	1

Course Outcomes		Contribution to Program Specific outcomes		
		PSO1	PSO2	PSO3
CO1	Demonstrate basic Operating system Commands, Shell scripts, System Calls and API wrt Linux	3	2	2
CO2	Implement various process scheduling algorithms and evaluate their performance	2	3	-
CO3	Implement and analyze concepts of synchronization and deadlocks	1	2	2
CO4	Implement various Memory Management techniques and evaluate their performance	-	2	2
CO5	Implement and analyze concepts of virtual memory	1	1	2
CO6	Demonstrate and analyze concepts of file management and I/O management techniques	2	2	3

Study and Evaluation Scheme

Course Code	Course Name	Teaching Scheme			Credits Assigned			
231CAUCL42	Operating System Lab	Theory	Practical	Tutorial	Theory	Practical	Tutorial	Total
		0	02	--	0	01	--	01

Course Code	Course Name	Examination Scheme		
231CAUCL42	Operating system Lab	Term Work	Practical & Oral	Total
		25	25	50

Term Work:

1. Term work should consist of at least 10 experiments on above content.
2. The final certification and acceptance of term work ensures that satisfactory performance of laboratory work and minimum passing marks in term work.
3. Term Work: 25 Marks (Total) = 10 Marks (Experiments) + 5 Marks (Mini Project) + 5 Marks (Assignments) + 5 Marks (Theory + Practical Attendance).

Practical/Experiments:

1. The final certification and acceptance of term work ensures that satisfactory performance of laboratory work and minimum passing marks in term work.
2. Practical exam will be based on the above syllabus.

Operating System

Experiment No. : 1

Explore Linux Commands

Experiment No. 1

1. **Aim:** Explore usage of basic Linux Commands and system calls for file, directory and process management. For eg: (mkdir, chdir, cat, ls, chown, chmod, chgrp, ps etc. system calls: open, read, write, close, getpid, setpid, getuid, getgid, getegid, geteuid. sort, grep, awk, etc.)
2. **Objectives:** From this experiment, the student will be able
 - To understand the working of Linux programming.
 - To understand the functionality of Linux operating system.
 - To learn some of the basic and useful commands of Linux.
3. **Outcomes:** The learner will be able to
 - Understand basic concept of Operating System, its type, architecture.
 - Explore various Operating system commands, system calls and able to write shell scripts, shell commands using kernel APIs.
4. **Hardware / Software Required :** Linux Operating System

5. Theory:

NAME	
	pwd - print name of current/working directory
SYNTAX	
	pwd
DESCRIPTION	
	Print the full filename of the current working directory.
NAME	
	ls - list directory contents
SYNTAX	
	ls [OPTION]... [FILE]...
DESCRIPTION	
	List information about the FILES (the current directory by default). Sort entries alphabetically if none of -cftuvSUX nor --sort is specified.
	-a :: do not ignore entries starting with .
	-d :: list directory entries instead of contents
	-l :: use a long listing format
	-r :: :reverse order while sorting

NAME mkdir - make directories SYNTAX mkdir DIRECTORY(ies) DESCRIPTION Create the DIRECTORY(ies), if they do not already exist.
NAME rmdir - remove empty directories SYNTAX rmdir [OPTION]... DIRECTORY... DESCRIPTION Remove the DIRECTORY(ies), if they are empty.
NAME whatis - display one-line manual page descriptions SYNTAX whatis [command name(s)] DESCRIPTION Each manual page has a short description available within it. whatis searches the manual page names and displays the manual page descriptions of any name matched.
NAME man - an interface to the on-line reference manuals SYNTAX man [command name(s)] DESCRIPTION man is the system's manual pager. Each page argument given to man is normally the name of a program, utility or function. The manual page associated with each of these arguments is then found and displayed.
NAME cat - concatenate files and print on the standard output SYNTAX cat [OPTION]... [FILE]...

<p>DESCRIPTION</p> <p>Concatenate FILE(s), or standard input, to standard output.</p> <p>-n :: number all output lines</p> <p>-E :: display \$ at end of each line</p> <p>-T :: display TAB characters as ^I</p> <p>-A:: equivalent to -vET</p> <p>NOTE:</p> <p>cat > [FILE] creates a new file or if file exist then overwrite its contents</p> <p>cat >> [FILE] Appends the content to the end of a file</p>	<p>NAME</p> <p>cp - copy files and directories</p> <p>SYNTAX</p> <p>cp [OPTION] SOURCE DEST</p> <p>DESCRIPTION</p> <p>Copy SOURCE to DEST, or multiple SOURCE(s) to DIRECTORY.</p>
<p>NAME</p> <p>rm - remove files or directories</p> <p>SYNTAX</p> <p>rm [OPTION]... FILE...</p> <p>DESCRIPTION</p> <p>This manual page documents the GNU version of rm. rm removes each specified file. By default, it does not remove directories.</p> <p>-f :: ignore nonexistent files and arguments, never prompt</p> <p>-i :: prompt before every removal</p> <p>-r :: remove directories and their contents recursively</p> <p>-d :: remove empty directories</p>	<p>NAME</p> <p>head - output the first part of files</p> <p>SYNTAX</p> <p>head [OPTION]... [FILE]...</p> <p>DESCRIPTION</p>

Print the first 10 lines of each FILE to standard output. With more than one FILE, precede each with a header giving the file name. With no FILE, or when FILE is -, read standard input.

-c K :: print the first K bytes of each file

-n K :: print the first K lines instead of the first 10

NAME

tail - output the last part of files

SYNTAX

tail [OPTION]... [FILE]...

DESCRIPTION

Print the last 10 lines of each FILE to standard output. With more than one FILE, precede each with a header giving the file name. With no FILE, or when FILE is -, read standard input.

-c K :: output the last K bytes; alternatively

-n K :: print the last K lines instead of the first 10

NAME

chmod - change file mode bits

SYNTAX

chmod [OPTION]... MODE[,MODE]... FILE...

chmod [OPTION]... OCTAL-MODE FILE...

DESCRIPTION

This manual page documents the GNU version of chmod. chmod changes the file mode bits of each given file according to mode, which can be either a symbolic representation of changes to make, or an octal number representing the bit pattern for the new mode bits.

File Permissions

Example:

```
ls -l myfile
0123456789  L User  Group      Size  Last-Updated  File-Name
-rwxrwxrwx  1 fred  student    2134  Feb 17 14:05  myfile
```

where:

L: number of links
0: type of file {"-": ordinary, "d": directory}
123: owner (user) rights
456: group rights
789: world (others) rights

More specifically, the access rights when applied to files or directories are:

	Files	Directories
147 = "r"	read	read file names (ls)
258 = "w"	write	create/delete files (cp, rm)
369 = "x"	execute	access files in sub-directory (called search permission)

Example:

```
drwxr-xr-- 2 fred student 1024 Mar 21 19:35.
```

0 = "d" its a directory

123 = "rwx", the owner has full permission

456 = "r-x" the student group members can list this directory and access its files

789 = "r--" everyone else can list the directory but not touch its files

or "--x" access directory files but only if file names are known, can't list directory

• Mode bits:

File permissions are stored in one number as a set of bits called the mode

User	Group	Other	
r w x	r w x	r w x	
4	4	4	read
2	2	2	write
1	1	1	execute

Example

```
chmod 644 myfile            sets "rw-r--r--" permissions
```

```
chmod 751 myprog            sets "rwxr-x--x" permissions
```

NAME

cut - remove sections from each line of files

SYNTAX

cut OPTION... [FILE]...

DESCRIPTION

Print selected parts of lines from each FILE to standard output.

-b :: select only these bytes

-c :: select only these characters

-d :: use DELIM instead of TAB for field delimiter

-f :: select only these fields

NAME

grep - print lines matching a pattern

SYNTAX

grep [OPTIONS] PATTERN [FILE...]

<pre>grep [OPTIONS] [-e PATTERN -f FILE] [FILE...]</pre> <p>DESCRIPTION</p> <p>grep searches the named input FILEs (or standard input if no files are named, or if a single hyphen-minus (-) is given as file name) for lines containing a match to the given PATTERN. By default, grep prints the matching lines.</p> <p>-i :: Ignore case distinctions in both the PATTERN and the input files.</p> <p>-v :: Invert the sense of matching, to select non-matching lines.</p> <p>-o :: Print only the matched (non-empty) parts of a matching line, with each such part on a separate output line.</p> <p>-n :: Prefix each line of output with the 1-based line number within its input file</p>	<p>NAME</p> <p>wc - print newline, word, and byte counts for each file</p> <p>SYNTAX</p> <p>wc [OPTION]... [FILE]...</p> <p>DESCRIPTION</p> <p>Print newline, word, and byte counts for each FILE, and a total line if more than one FILE is specified. With no FILE, or when FILE is -, read standard input. A word is a non-zero-length sequence of characters delimited by white space. The options below may be used to select which counts are printed, always in the following order: newline, word, character, byte, maximum line length.</p> <p>-c :: print the byte counts</p> <p>-l :: print the newline counts</p> <p>-L :: print the length of the longest line</p> <p>-w :: print the word counts</p>
<p>NAME</p> <p>sort - sort lines of text files</p> <p>SYNTAX</p> <p>sort [OPTION]... [FILE]...</p> <p>DESCRIPTION</p> <p>Write sorted concatenation of all FILE(s) to standard output.</p>	<p>NAME</p> <p>cal — displays a calendar</p> <p>SYNTAX</p>

cal [month] [year]
<p>DESCRIPTION</p> <p>The cal utility displays a simple calendar in traditional format. If arguments are not specified, the current month is displayed.</p>
<p>NAME</p> <p>cmp - compare two files byte by byte</p> <p>SYNTAX</p> <p>cmp [OPTION]... FILE1 FILE2</p> <p>DESCRIPTION</p> <p>Compare two files byte by byte.</p>
<p>NAME</p> <p>date - print or set the system date and time</p> <p>SYNTAX</p> <p>date [OPTION]... [+FORMAT]</p> <p>DESCRIPTION</p> <p>Display the current time in the given FORMAT, or set the system date.</p>
<p>NAME</p> <p>clear - clear the terminal screen</p> <p>SYNTAX</p> <p>clear</p> <p>DESCRIPTION</p> <p>clear clears your screen</p>
<p>Both getppid() and getpid() are inbuilt functions defined in unistd.h library.</p> <ol style="list-style-type: none"> 1. getppid() : returns the process ID of the parent of the calling process. If the calling process was created by the fork() function and the parent process still exists at the time of the getppid function call, this function returns the process ID of the parent process. Otherwise, this function returns a value of 1 which is the process id for init process. <p>Syntax:</p> <p>pid_t getppid(void);</p> <p>Return type: getppid() returns the process ID of the parent of the current process. It never throws any error therefore is always successful.</p>

NOTE: At some instance of time, it is not necessary that child process will execute first or parent process will be first allotted CPU, any process may get CPU assigned, at some quantum time. Moreover, process id may differ during different executions.

2. **getpid()** : returns the process ID of the calling process. This is often used by routines that generate unique temporary filenames.

Syntax:

pid_t getpid(void);

Return type: getpid() returns the process ID of the current process. It never throws any error therefore is always successful

Linux provides us a utility called **ps** for viewing information related with the processes on a system which stands as abbreviation for “**Process Status**”. **ps** command is used to list the currently running processes and their PIDs along with some other information depends on different options. It reads the process information from the virtual files in **/proc** file-system. **/proc** contains virtual files, this is the reason it's referred as a virtual file system. **ps** provides numerous options for manipulating the output according to our need.

Syntax –

ps [options]

Options for ps Command :

Simple process selection : Shows the processes for the current shell –

```
[root@rhel7 ~]# ps
```

PID	TTY	TIME	CMD
12330	pts/0	00:00:00	bash
21621	pts/0	00:00:00	ps

Result contains four columns of information.

Where,

PID – the unique process ID

TTY – terminal type that the user is logged into

TIME – amount of CPU in minutes and seconds that the process has been running

CMD – name of the command that launched the process.

View all the running processes :

```
[root@rhel7 ~]# ps -r
```

View all processes owned by you : Processes i.e same EUID as **ps** which means runner of the **ps** command, root in this case –

```
[root@rhel7 ~]# ps -x
```

View process by process ID.

Syntax :

```
ps p process_id
```



```
ps -p process_id
```

```
ps --pid process_id
```

Example :

```
[root@rhel7 ~]# ps p 27223
```

	PID	TTY	STAT	TIME	COMMAND
27223	?		Ss	0:01	sshd: root@pts/2

```
[root@rhel7 ~]# ps -p 27223
```

	PID	TTY	TIME	CMD
27223	?		00:00:01	sshd

```
[root@rhel7 ~]# ps --pid 27223
```

	PID	TTY	TIME	CMD
27223	?		00:00:01	sshd

To view process according to user-defined format.

Syntax :

```
[root@rhel7 ~]# ps --formate column_name
```

```
[root@rhel7 ~]# ps -o column_name
```

```
[root@rhel7 ~]# ps o column_name
```

Example :

```
[root@rhel7 ~]# ps -aN --format cmd,pid,user,ppid
```

CMD	PID	USER	PPID
/usr/lib/systemd/systemd --	1	root	0
[kthreadd]	2	root	0
[ksoftirqd/0]	3	root	2
[kworker/0:0H]	5	root	2
[migration/0]	7	root	2
[rcu_bh]	8	root	2
[rcu_sched]	9	root	2

	<pre>[watchdog/0] 10 root 2</pre> <p>In this example I wish to see command, process ID, username and parent process ID, so I pass the arguments cmd, pid, user and ppid respectively</p> <p>View processes using highest memory.</p> <pre>ps -eo pid,ppid,cmd,%mem,%cpu --sort=-%mem</pre>
	<p>System Calls: File Manipulation system calls</p> <p>Some common system calls are <i>create</i>, <i>delete</i>, <i>read</i>, <i>write</i>, <i>reposition</i>, or <i>close</i>. Also, there is a need to determine the file attributes – <i>get</i> and <i>set</i> file attribute. Many times the OS provides an API to make these system calls.</p>

6. Conclusion and Discussion: Thus, we have studied some of the basic Linux commands. We also used the pipe operator for forwarding output one command to other command as input.

7. Viva Questions:

- What is the difference between UNIX and LINUX?
- What is LILO?
- What is Chmod, umask and tty command?
- How do you open a command prompt when issuing a command?
- What is the pwd command?

8. References:

- William Stallings, Operating System: Internals and Design Principles, Prentice Hall, 8th Edition, 2014
- Abraham Silberschatz, Peter Baer Galvin and Greg Gagne, Operating System Concepts, John Wiley & Sons , Inc., 9th Edition, 2016.
- Andrew Tannenbaum, Operating System Design and Implementation, Pearson, 3rd Edition.
- D.M Dhamdhare, Operating Systems: A Concept Based Approach, Mc-Graw Hill
- Maurice J. Bach, “Design of UNIX Operating System”, PHI
- Achyut Godbole and Atul Kahate, Operating Systems, Mc Graw Hill Education, 3rd Edition
- The Linux Kernel Book, Remy Card, Eric Dumas, Frank Mevel, Wiley Publications.

Operating System

Experiment No.: 2

Linux Shell Script

Experiment No. 2

1. Aim: Write shell scripts to do the following:

- a. Display OS version, release number, kernel version
- b. Display top 10 processes in descending order
- c. Display processes with highest memory usage.
- d. Display current logged in user and log name.
- e. Display current shell, home directory, operating system type, current path setting, current working directory.

2. Objective: From this experiment, the student will be able to

- Understand basic concept of Operating System, its type, architecture.
- Explore various Operating system commands, system calls and able to write shell scripts, shell commands using kernel APIs.

3. Outcomes: The learner will be able to

- Explore various Operating system commands, system calls and able to write shell scripts
- Analyze the behavior of Linux Command and shell script.
- Use these commands for computing practice which will be able to match the industry requirements in the domain of operating system
- Recognize the need for Linux and shell script in life-long learning.

4. Hardware / Software Required : Linux Operating System

5. Theory:

The -o (or --format) option of ps allows you to specify the output format. A favorite of mine is to show the processes' PIDs (pid), PPIDs (pid), the name of the executable file associated with the process (cmd), and the RAM and CPU utilization (%mem and %cpu, respectively). Additionally, I use --sort to sort by either %mem or %cpu. By default, the output will be sorted in ascendant form, but personally I prefer to reverse that order by adding a minus sign in front of the sort criteria. To add other fields to the output, or change the sort criteria, refer to the OUTPUT FORMAT CONTROL section in the man page of ps command.

Display the top ten running process sorted by memory usage, ps returns all the running process which are then sorted by the 4th field in numerical order and the top 10 are to STDOUT.

6. Algorithm:

echo "Top 10 processes in descending order are:"

```
ps -eo pid,ppid,cmd,%mem,%cpu --sort=-%mem |head
```

echo "Processes with highest memory usage are:"

```
ps aux |sort -nk +4 |tail
```

echo "current logged in users and logname are:"

```
logname
```

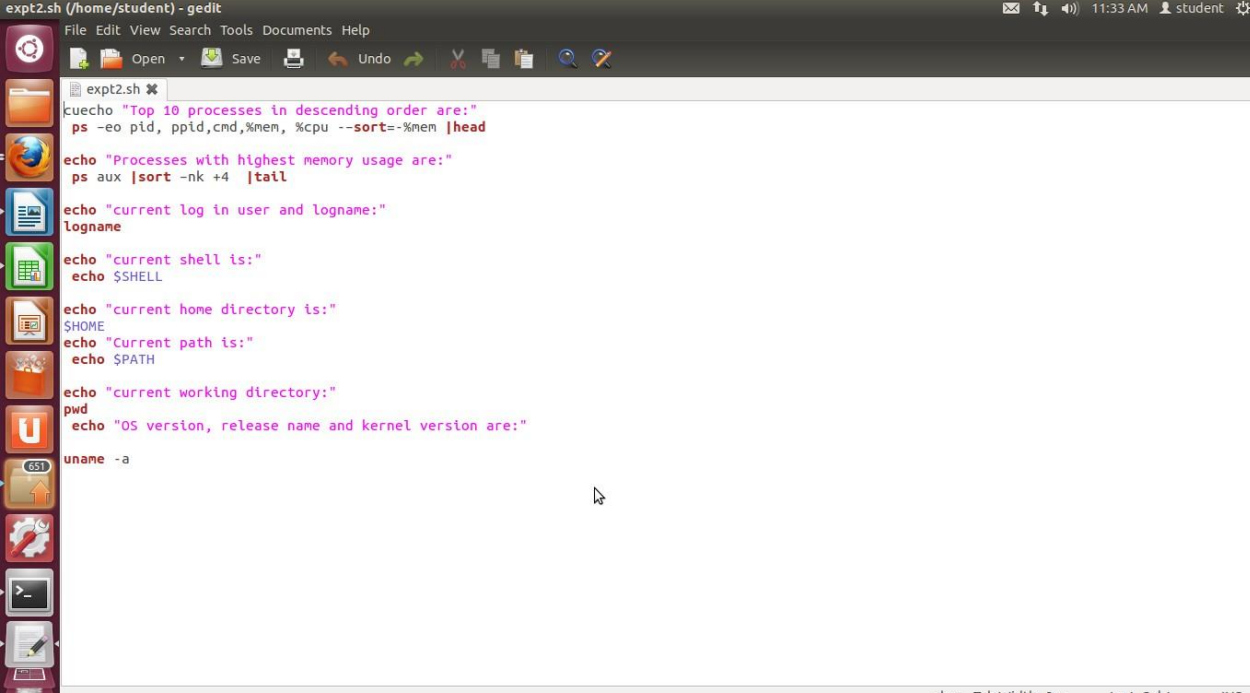
echo "current shell is:"

```
echo $SHELL
```

```
echo "home directory is:"  
$HOME
```

```
echo "Current path is:"  
echo $PATH  
echo "current working directory is"  
pwd
```

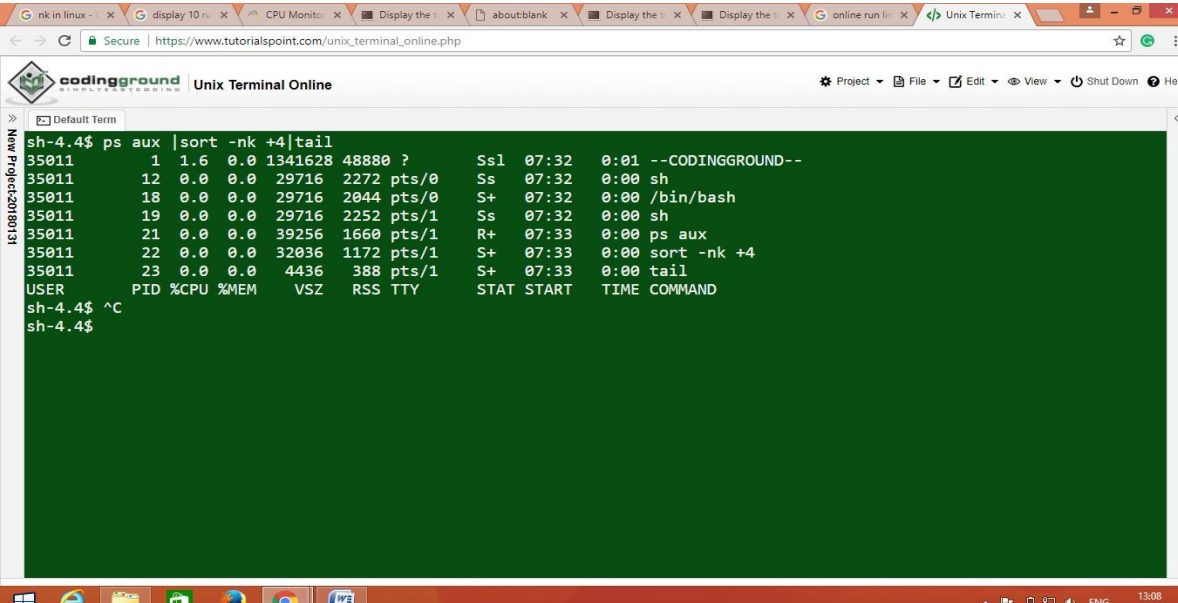
```
echo "current OS version, release number, kernel version are:"  
uname -a
```



The screenshot shows a gedit editor window titled 'expt2.sh (/home/student) - gedit'. The window contains a script with the following content:

```
#!/bin/bash  
echo "Top 10 processes in descending order are:"  
ps -eo pid,ppid,cmd,%mem,%cpu --sort=-%mem | head  
  
echo "Processes with highest memory usage are:"  
ps aux | sort -nk +4 | tail  
  
echo "current log in user and logname:"  
logname  
  
echo "current shell is:"  
echo $SHELL  
  
echo "current home directory is:"  
echo $HOME  
echo "Current path is:"  
echo $PATH  
  
echo "current working directory:"  
pwd  
echo "OS version, release name and kernel version are:"  
uname -a
```

The status bar at the bottom indicates 'sh', 'Tab Width: 8', 'Ln 1, Col 1', and 'INS'.



The screenshot shows a web browser window with the URL 'https://www.tutorialspoint.com/unix_terminal_online.php'. The browser displays a terminal window titled 'Unix Terminal Online' with the following output:

```
sh-4.4$ ps aux | sort -nk +4 | tail  
35011 1 1.6 0.0 1341628 48880 ? Ssl 07:32 0:01 --CODINGGROUND--  
35011 12 0.0 0.0 29716 2272 pts/0 Ss 07:32 0:00 sh  
35011 18 0.0 0.0 29716 2044 pts/0 S+ 07:32 0:00 /bin/bash  
35011 19 0.0 0.0 29716 2252 pts/1 Ss 07:32 0:00 sh  
35011 21 0.0 0.0 39256 1660 pts/1 R+ 07:33 0:00 ps aux  
35011 22 0.0 0.0 32036 1172 pts/1 S+ 07:33 0:00 sort -nk +4  
35011 23 0.0 0.0 4436 388 pts/1 S+ 07:33 0:00 tail  
USER PID %CPU %MEM VSZ RSS TTY STAT START TIME COMMAND  
sh-4.4$ ^C  
sh-4.4$
```

The terminal window is titled 'Default Term' and shows a 'New Project' button on the left. The browser's status bar at the bottom indicates '13:08' and '31-01-2018'.

```

student@student-HP-Pro-3330-MT: ~
-w,w wide output      n numeric WCHAN,UID  -H process hierarchy
student@student-HP-Pro-3330-MT:~$ ./expt2.sh
./expt2.sh: line 1: cuecho: command not found
ERROR: Garbage option.
***** simple selection ***** ***** selection by list *****
-A all processes      -C by command name
-N negate selection   -G by real group ID (supports names)
-a all w/ tty except session leaders
-d all except session leaders
-e all processes      -U by real user ID (supports names)
-T all processes on this terminal
-a all w/ tty, including other users
-g OBSOLETE -- DO NOT USE -g by session OR by effective group name
-r only running processes
-x processes w/o controlling ttys
-p by process ID
-s processes in the sessions given
-t by tty
-u by effective user ID (supports names)
-U processes for specified users
-t by tty
***** output format ***** ***** long options *****
-o user-defined -f full      --Group --User --pid --cols --ppid
-j,j job control s signal    --group --user --sid --rows --info
-O preloaded -o v virtual memory
--cumulative --format --deselect
-l,l long          u user-oriented --sort --tty --forest --version
-F extra full      X registers      --heading --no-heading --context
***** misc options *****
-L list format codes f ASCII art forest
-n,m,-L,-T,H threads S children in sum -y change -l format
-M,Z security data c true command name -c scheduling class
-w,w wide output    n numeric WCHAN,UID -H process hierarchy
Processes with highest memory usage are:
sort: open failed: -nk: No such file or directory
current log in user and logname:
student
current shell is:
/bin/bash
current home directory is:
./expt2.sh: line 14: /home/student: Is a directory
Current path is:
/usr/lib/lightdm/lightdm:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games
current working directory:
/home/student
OS version, release name and kernel version are:
Linux student-HP-Pro-3330-MT 3.2.0-29-generic-pae #46-Ubuntu SMP Fri Jul 27 17:25:43 UTC 2012 i686 i686 i386 GNU/Linux
student@student-HP-Pro-3330-MT:~$

```

7. Conclusion and Discussion:

Above given statement tested and executed successfully using shell script.

8. Viva Questions:

- What is shell script?
- How to run shell script?
- What are the types of shell?
- What is command line argument?
- What needs to be done before you can run a shell script from the command line prompt?
- How do you terminate a shell script if statement?
- What code would you use in a shell script to determine if a directory exists?
- How do you access command line arguments from within a shell script?
- Within a UNIX shell scripting loop construct, what is the difference between the break and continue?

9. References:

- William Stallings, Operating System: Internals and Design Principles, Prentice Hall, 8th Edition, 2014
- Abraham Silberschatz, Peter Baer Galvin and Greg Gagne, Operating System Concepts, John Wiley & Sons, Inc., 9th Edition, 2016.
- Andrew Tannenbaum, Operating System Design and Implementation, Pearson, 3rd Edition.
- D.M Dhamdhare, Operating Systems: A Concept Based Approach, Mc-Graw Hill
- Maurice J. Bach, "Design of UNIX Operating System", PHI
- Achyut Godbole and Atul Kahate, Operating Systems, Mc Graw Hill Education, 3rd Edition
- The Linux Kernel Book, Remy Card, Eric Dumas, Frank Mevel, Wiley Publications.

Operating System

Experiment No.: 3

Linux - APIs

Experiment No. 3

1. **Aim:** Implement basic commands of linux like ls, cp, mv and others using kernel APIs.
2. **Objectives:** From this experiment, the student will be able
 - To understand the functionality of Linux operating system.
 - To learn some of the basic and useful commands of Linux.
3. **Outcomes:** The learner will be able to
 - Understand basic concept of Operating System, its type, architecture.
 - Explore various Operating system commands, system calls and able to write shell scripts, shell commands using kernel APIs.
4. **Hardware / Software Required :** Linux operating system, C
5. **Theory:**
 - **cp—Copy files**

SYNOPSIS cp [options] source dest
 cp [options] source... directory

DESCRIPTION

If the last argument names an existing directory, cp copies each other given file into a file with the same name in that directory. Otherwise, if only two files are given, it copies the first onto the second. It is an error if the last argument is not a directory and more than two files are given. By default, it does not copy directories.

OPTIONS

cp -a	archive files
cp -f	force copy by removing the destination file if needed
cp -i	interactive - ask before overwrite
cp -l	link files instead of copy
cp -L	follow symbolic links
cp -n	no file overwrite
cp -R	recursive copy (including hidden files)
cp -u	update - copy when source is newer than dest
cp -v	verbose - print informative messages

- **ls—List contents of directories**

SYNOPSIS ls [options] [file|dir]

DESCRIPTION

These programs list each given file or directory name. Directory contents are sorted alphabetically. For ls, files are by default listed in columns, sorted vertically, if the standard

output is a terminal; otherwise, they are listed one per line. For dir, files are by default listed in columns, sorted vertically. For vdir, files are by default listed in long format.

OPTIONS

ls -a	list all files including hidden file starting with '.'
ls --color	colored list [=always/never/auto]
ls -d	list directories - with '*'
ls -F	add one char of */=>@ to enteries
ls -i	list file's inode index number
ls -l	list with long format - show permissions
ls -la	list long format including hidden files
ls -lh	list long format with readable file size
ls -ls	list with long format with file size
ls -r	list in reverse order
ls -R	list recursively directory tree
ls -s	list file size
ls -S	sort by file size
ls -t	sort by time & date
ls -X	sort by extension name

- **mv—Rename files**

SYNOPSIS

mv [options] source dest

mv [options] source... directory

DESCRIPTION

If the last argument names an existing directory, mv moves each other given file into a file with the same name in that directory. Otherwise, if only two files are given, it moves the first onto the second. It is an error if the last argument is not a directory and more than two files are given. It can move only regular files across filesystems. If a destination file is unwritable, the standard input is a tty, and the -f or --force option is not given, mv prompts the user for whether to overwrite the file. If the response does not begin with y or Y, the file is skipped.

OPTIONS

mv -f	force move by overwriting destination file without prompt
mv -i	interactive prompt before overwrite
mv -u	update - move when source is newer than destination
mv -v	verbose - print source and destination files
man mv	help manual

6. Algorithm:

- **Ls command**

```
#include <sys/types.h>
#include <sys/dir.h>
#include <sys/param.h>
#include <stdio.h>
#define FALSE 0
#define TRUE 1
extern int alphasort();
char pathname[MAXPATHLEN];
main() {
    int count,i;
    struct dirent **files;
    int file_select();
    if (getwd(pathname) == NULL )
    {
        printf("Error getting pathn");
        exit(0);
    }
    printf("Current Working Directory = %sn",pathname);
    count = scandir(pathname, &files, file_select, alphasort);
    if (count <= 0)
    {
        printf("No files in this directoryn");
        exit(0);
    }
    printf("Number of files = %dn",count);
    for (i=1;i<count 1; i)
        printf("%s \n",files[i-1]->d_name);
    }
    int file_select(struct direct *entry)
    {
        if ((strcmp(entry->d_name, ".") == 0) || (strcmp(entry->d_name, "..") == 0))
            return (FALSE);
        else
            return (TRUE);
    }
}
```

- **Mv command**

```
#include<sys/types.h>
#include<sys/stat.h>
#include<stdio.h>
#include<fcntl.h>
main( int argc,char *argv[] )
{
    int i,fd1,fd2;
    char *file1,*file2,buf[2];
```

```

file1=argv[1];
file2=argv[2];
printf("file1=%s file2=%s",file1,file2);
fd1=open(file1,O_RDONLY,0777);
fd2=creat(file2,0777);
while(i=read(fd1,buf,1)>0)
write(fd2,buf,1);
remove(file1);
close(fd1);
close(fd2);
}

```

7. **Conclusion and Discussion:** Thus, we have studied some of the basic Linux commands. We also implemented few commands using kernel API.

8. **Viva Questions:**

- What is Chmod, umask and whois command?
- How cp command works?
- What is pwd, cat command?
- What is grep, awk command?

9. **References:**

- William Stallings, Operating System: Internals and Design Principles, Prentice Hall, 8th Edition, 2014
- Abraham Silberschatz, Peter Baer Galvin and Greg Gagne, Operating System Concepts, John Wiley & Sons, Inc., 9th Edition, 2016.
- Andrew Tannenbaum, Operating System Design and Implementation, Pearson, 3rd Edition.
- D.M Dhamdhare, Operating Systems: A Concept Based Approach, Mc-Graw Hill
- Maurice J. Bach, "Design of UNIX Operating System", PHI
- Achyut Godbole and Atul Kahate, Operating Systems, Mc Graw Hill Education, 3rd Edition
- The Linux Kernel Book, Remy Card, Eric Dumas, Frank Mevel, Wiley Publications.

Operating System

Experiment No.: 4

Linux Process

Experiment No. 4

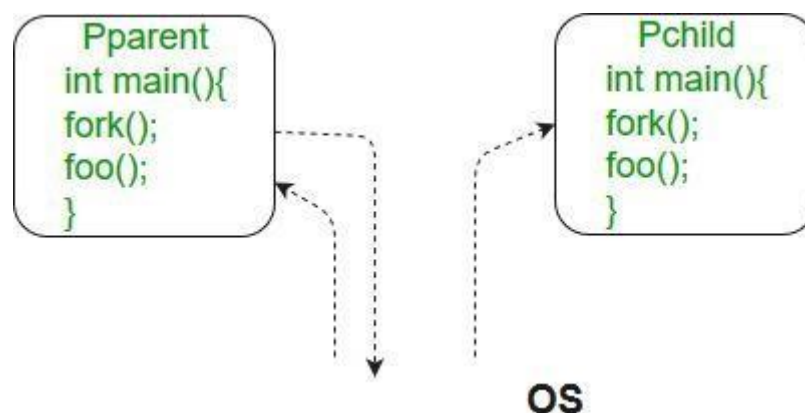
1. **Aim:** Create a child process in Linux using the fork system call. From the child process obtain the process ID of both child and parent by using getpid and getppid system call.
2. **Objective:** From this experiment, the student will be able to
 - Study how to create a process in UNIX using fork() system call.
 - Study how to use wait(), exec() system calls, zombie, daemon and orphan states.
 - Appreciate the role of operating system as System software
3. **Outcomes:** The learner will be able to
 - Understand basic concept of Operating System, its type, architecture.
 - Explore various Operating system commands, system calls and able to write shell scripts, shell commands using kernel APIs.
4. **Hardware / Software Required :** Linux operating system, Java/C
5. **Theory:**

fork():

It is a system call that creates a new process under the UNIX operating system. It takes no arguments. The purpose of fork() is to create a new process, which becomes the child process of the caller. After a new child process is created, both processes will execute the next instruction following the fork() system call. Therefore, we have to distinguish the parent from the child. This can be done by testing the returned value of fork():

- If fork() returns a negative value, the creation of a child process was unsuccessful.
- fork() returns a zero to the newly created child process.
- fork() returns a positive value, the process ID of the child process, to the parent. The returned process ID is of type pid_t defined in sys/types.h. Normally, the process ID is an integer. Moreover, a process can use function getpid() to retrieve the process ID assigned to this process.

Therefore, after the system call to fork(), a simple test can tell which process is the child. Note that Unix will make an exact copy of the parent's address space and give it to the child. Therefore, the parent and child processes have separate address spaces.



ps command:

The ps command shows the processes we're running, the process another user is running, or all the processes on the system. E.g.

```
$ ps -ef
```

By default, the `ps` program shows only processes that maintain a connection with a terminal, a console, a serial line, or a pseudo terminal. Other processes run without needing to communicate with a user on a terminal. These are typically system processes that Linux uses to manage shared resources. We can use `ps` to see all such processes using the `-e` option and to get “full” information with `-f`.

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. Within the `exec` family there are functions that vary slightly in their capabilities.

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. Some flags that indicate the completion status of the child process are passed back with the integer pointer.

One of the main purposes of `wait()` is to wait for completion of child processes.

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.

Zombie Process:

When a child process terminates, an association with its parent survives until the parent in turn either terminates normally or calls `wait`. The child process entry in the process table is therefore not freed up immediately. Although no longer active, the child process is still in the system because its exit code needs to be stored in case the parent subsequently calls `wait`. It becomes what is known as defunct, or a zombie process.

Orphan Process:

An orphan process is a computer process whose parent process has finished or terminated, though itself remains running. A process may also be intentionally orphaned so that it becomes detached from the user's session and left running in the background; usually to allow a long running job to complete without further user attention, or to start an indefinitely running service. Under UNIX, the latter kinds of processes are typically called daemon processes. The UNIX `nohup` command is one means to accomplish this.

Daemon Process:

It is a process that runs in the background, rather than under the direct control of a user; they are usually initiated as background processes.

6. A) Example of fork() system call:

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

    // make two process which run same
    // program after this instruction
```

```

        fork();

        printf("Hello world!\n");
        return 0;
    }

```

Output:

Hello world!

Hello world!

B) Example of fork() system call:

pids_after_fork.c

```

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

int main() {
    pid_t pid, mypid, myppid;
    pid = getpid();
    printf("Before fork: Process id is %d\n", pid);
    pid = fork();
    if (pid < 0) {
        perror("fork() failure\n");
        return 1;
    }
    // Child process
    if (pid == 0) {
        printf("This is child process\n");
        mypid = getpid();
        myppid = getppid();
        printf("Process id is %d and PPID is %d\n", mypid, myppid);
    } else { // Parent process
        sleep(2);
        printf("This is parent process\n");
        mypid = getpid();
        myppid = getppid();
        printf("Process id is %d and PPID is %d\n", mypid, myppid);
        printf("Newly created process id or child pid is %d\n", pid);
    }
    return 0;
}

```

Sample output :

Before fork: Process id is 166629

This is child process

Process id is 166630 and PPID is 166629

Before fork: Process id is 166629

This is parent process

Process id is 166629 and PPID is 166628

Newly created process id or child pid is 166630

7. Conclusion and Discussion:

We studied different system calls that are required to create parent and child process. We also studied zombie, daemon and orphan states.

8. Viva Questions:

- What is Zombie process in UNIX?
- How do you find Zombie process in UNIX?
- In a file word UNIX is appearing many times? How will you count number

9. References:

- William Stallings, Operating System: Internals and Design Principles, Prentice Hall, 8th Edition, 2014
- Abraham Silberschatz, Peter Baer Galvin and Greg Gagne, Operating System Concepts, John Wiley & Sons , Inc., 9th Edition, 2016.
- Andrew Tannenbaum, Operating System Design and Implementation, Pearson, 3rd Edition.
- D.M Dhamdhare, Operating Systems: A Concept Based Approach, Mc-Graw Hill
- Maurice J. Bach, “Design of UNIX Operating System”, PHI
- Achyut Godbole and Atul Kahate, Operating Systems, Mc Graw Hill Education, 3rd Edition
- The Linux Kernel Book, Remy Card, Eric Dumas, Frank Mevel, Wiley Publications.

Operating System

Experiment No.: 5

CPU scheduling algorithms

Experiment No. 5

Aim: Write a program to implement non-pre-emptive and Pre-emptive scheduling algorithms and demonstrate using CPU OS simulator.

1. **Objectives:** From this experiment, the student will be able to
 - To study CPU Scheduling algorithms such as FCFS, SJF, Priority and Round Robin.
 - Compare various algorithms of CPU scheduling.
 - To understand the use of CPU scheduling for process management.
2. **Outcomes:** The learner will be able to
 - Understand the concept of a process, thread and analyze performance of process scheduling algorithms. They should be able to evaluate process management techniques using simulator
3. **Hardware / Software Required:** Linux operating system, Java/C
4. **Theory:**

CPU scheduling is the task of selecting a waiting process from the ready queue and allocating the CPU to it. The CPU is allocated to the selected process by the dispatcher (It is the module that gives control of the CPU to the processes by short-term scheduler). Scheduling is a fundamental operating system function.

In a single processor system, only one process can run at a time; any other process must wait until the CPU is free and can be rescheduled. The objective of multiprogramming is to have some process running at all times, to maximize CPU utilization.

CPU scheduling decisions may take place under the following four circumstances:

- When a process switches from the running state to the waiting state
- When a process switches from the running state to the ready state.
- When a process switches from the waiting state to the ready state.
- When a process terminates.

Depending on the above circumstances the two types of scheduling are:

- **Non-Preemptive**
- **Preemptive**

Non-Preemptive: Under this scheduling, once the CPU has been allocated to a process, the process keeps the CPU until it releases the CPU either by terminating or by switching to the waiting state.

Preemptive: Under this scheduling, once the CPU has been allocated to a process, the process does not keep the CPU but can be utilized by some other process. This incurs a cost associated with access to shared data. It also affects the design of the operating system kernel.

Scheduling Criteria:

1. CPU utilization: It can range from 0-100%. In a real system, it ranges should range from 40-90%.
2. Throughput: Number of processes that are completed per unit time.
3. Turnaround time: How long a process takes to execute. It is the sum of the periods spent waiting to get into memory, waiting in the ready queue, executing on the CPU, and doing I/O
4. Waiting time: It is the sum of the periods spent waiting in the ready queue.
5. Response time: Time from the submission of a request until the first response is produced.

It is desirable to maximize CPU utilization and Throughput and minimize Turnaround time, waiting time and Response time.

Scheduling Algorithms:

- **FCFS (First-Come, First-Served):**
 - It is the simplest algorithm and NON-PREEMPTIVE.
 - The process that requests the CPU first is allocated the CPU first.
 - The implementation is easily managed by queue. When a process enters the ready queue, its PCB is linked onto the tail of the queue. When the CPU is free, it is allocated to the process at the head of the queue
 - The average waiting time, however, is long. It is not minimal and may vary substantially if the process's CPU burst time varies greatly.
 - This algorithm is particularly troublesome for time-sharing systems.
- **SJF (Shortest Job First):**
 - This algorithm associates with each process the length of the process's next CPU burst. When the CPU is available, it is assigned to the process that has the smallest next CPU burst. If the next CPU bursts of two processes are same, FCFS is used to break the tie.
 - It is also called shortest next CPU burst algorithm or shortest remaining time first scheduling.
 - It is provably optimal, in that it gives the minimum average waiting time for a given set of processes.
 - The real difficulty with SJF knows the length of the next CPU request.
 - It can be either PREEMPTIVE (SRTF- Shortest Remaining Time First) or NON-PREEMPTIVE.
- **PRIORITY SCHEDULING:**
 - The SJF is a special case of priority scheduling.
 - In priority scheduling algorithm, a priority is associated with each process, and the CPU is allocated to the process with the highest priority.
 - It can be either PREEMPTIVE or NON-PREEMPTIVE
 - A major problem with priority scheduling algorithms is indefinite blocking, or starvation.
 - A solution to starvation is AGING. It is a technique of gradually increasing the priority of process that wait in the system for long time.
- **ROUND ROBIN SCHEDULING:**
 - It is designed especially for time-sharing systems.
 - It is similar to FCFS, but preemption is added to switch between processes.
 - A time quantum is defined.
 - The CPU scheduler goes around the ready queue, allocating the CPU to each process for a time interval of up to 1 time quantum. If a process's CPU burst exceeds 1 time quantum, that process is preempted and is put back in the ready queue.

5. Algorithm:

FCFS Scheduling

1. Start
2. Accept no. of processes from user.
3. Accept burst time of each process i.e BT[i].
4. Initialize waiting time of P1=0 i.e. (WT[1])=0.
5. For(i=2;i<=n;i++)
6. WT[i]=WT[i-1]+BT[i-1]
7. End of for loop

8. Calculate average waiting time= (Total Waiting Time) / No. of Processes
9. End

SJF (Shortest Job First) Scheduling

1. Start
2. Accept no. of processes from user.
3. Accept burst time of each process i.e BT[i].
4. Sort the processes according to ascending order of burst time.
5. Initialize waiting time of first process = 0 i.e. (WT[1])=0.
6. For(i=2;i<=n;i++)
7. WT[i]=WT[i-1]+BT[i-1]
8. End of for loop
9. Calculate average waiting time= (Total Waiting Time) / No. of Processes
10. End

Priority Scheduling

1. Start
2. Accept no. of processes from user.
3. Accept burst time and priority of each process i.e BT[i].
4. Sort the processes and burst time according to ascending order of priority.
5. Initialize waiting time of first process = 0 i.e. (WT[1])=0.
6. For(i=2;i<=n;i++)
7. WT[i]=WT[i-1]+BT[i-1]
8. End of for loop
9. Calculate average waiting time= (Total Waiting Time) / No. of Processes
10. End

CPU-OS simulator:

The OS simulator is designed to support two main aspects of a computer system's resource management: process management and memory management. Image 3 shows the main user interface for this simulator. Once a compiled code is loaded in CPU memory, its image is also available to the OS simulator. It is then possible to create multiple instances of the program images as separate processes. The OS simulator displays the running processes, the ready processes and the waiting processes. Each process is assigned a separate process control block (PCB) which contains information on process state. This information is displayed in a separate window. The memory display demonstrates the dynamic nature of page allocations according to the currently selected placement policy. The OS maintains a separate page table for each process which can also be observed. The simulator demonstrates how data memory is relocated and the page tables are modified as the pages are moved in and out of the main memory illustrating virtual memory activity.

The process scheduler includes various selectable scheduling policies which includes priority based, pre-emptive and round robin scheduling with variable time quantum.

1. Algorithm:

Step 1: Entering source code into the compiler and compiling the code to an executable program.

Step 2: Loading the program into the CPU simulator's memory

Step 3: Create Processes from Programs in the OS Simulator

Step 4: Select different scheduling policies and run the processes in the OS simulator. Observe the differences between Pre-emptive and Non-Pre-emptive scheduling

Step 5: Locate the CPU register values in a process's PCB when it is in the ready queue. Explain how the CPU register values in PCB are used in Round Robin scheduling.

6. Conclusion and Discussion:

- There are different types of scheduling algorithms. Depending upon the average waiting time and turnaround time, above stated scheduling algorithms are compared.
- INPUT: Accept no. of processes, burst time and priority of each process from user.
- OUPUT: Display waiting time of each process and average waiting time of CPU.
- From results, we can conclude that SJF always results into a lowest AWT.
- The OS simulator is able to carry out context-switching which can be visually enhanced by slowing down or suspending the progress at some key stage to enable to study the states of CPU registers, stack, cache, pipeline and the PCB contents.

7. Viva Questions:

- What is process?
- Which module gives control of the CPU to the process selected by the short-term scheduler?
- What are short, long and medium-term scheduling?
- What are the different Process States?
- What is preemptive multitasking?

8. References:

- William Stallings, Operating System: Internals and Design Principles, Prentice Hall, 8th Edition, 2014
- Abraham Silberschatz, Peter Baer Galvin and Greg Gagne, Operating System Concepts, John Wiley & Sons , Inc., 9th Edition, 2016.
- Andrew Tannenbaum, Operating System Design and Implementation, Pearson, 3rd Edition.
- D.M Dhamdhare, Operating Systems: A Concept Based Approach, Mc-Graw Hill
- Maurice J. Bach, "Design of UNIX Operating System", PHI
- Achyut Godbole and Atul Kahate, Operating Systems, Mc Graw Hill Education, 3rd Edition
- The Linux Kernel Book, Remy Card, Eric Dumas, Frank Mevel, Wiley Publications.

Operating System

Experiment No.: 6

Process Management: Synchronization

Experiment No. 6

Aim: - Write a C program to implement solution of Producer consumer problem through Semaphore

1. **Objectives:** From this experiment, the student will be able to
 - Study the inter-process communication in operating system.
 - Analyse how to solve the problems occur during process synchronization
2. **Outcomes:** The learner will be able to
 - Apply and analyse the concepts of interprocess communication and solve the classical producer and consumer problems using semaphores concept.
 -
3. **Hardware / Software Required:** Linux operating system, Java/C

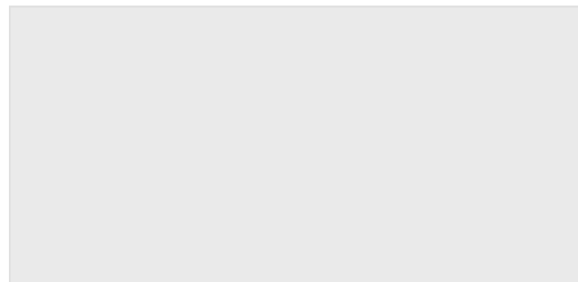
4. **Theory:**

The producer consumer problem is a synchronization problem. We have a buffer of fixed size. A producer can produce an item and can place in the buffer. A consumer can pick items and can consume them. We need to ensure that when a producer is placing an item in the buffer, then at the same time consumer should not consume any item. In this problem, buffer is the critical section. To solve this problem, we need two counting semaphores – Full and Empty. “Full” keeps track of number of items in the buffer at any given time and “Empty” keeps track of number of unoccupied slots.

Semaphore : A semaphore S is an integer variable that can be accessed only through two standard operations :

wait() - The wait() operation reduces the value of semaphore by 1

signal() - The signal() operation increases its value by 1.



Initialization of semaphores

mutex = 1

Full = 0 // Initially, all slots are empty. Thus full slots are 0

Empty = n // All slots are empty initially

Solution for Producer

```

do{
//produce an item
wait(empty);
wait(mutex);
//place in buffer
signal(mutex);
signal(full);
}while(true)

```

When producer produces an item then the value of “empty” is reduced by 1 because one slot will be filled now. The value of mutex is also reduced to prevent consumer to access the buffer. Now, the producer has placed the item and thus the value of “full” is increased by 1. The value of mutex is also increased by 1 because the task of producer has been completed and consumer can access the buffer.

Solution for Consumer

```

do{
wait(full);
wait(mutex);
// remove item from buffer
signal(mutex);
signal(empty);
// consumes item
}while(true)

```

As the consumer is removing an item from buffer, therefore the value of “full” is reduced by 1 and the value of mutex is also reduced so that the producer cannot access the buffer at this moment. Now, the consumer has consumed the item, thus increasing the value of “empty” by 1. The value of mutex is also increased so that producer can access the buffer now.

Program:

```

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

int mutex=1,full=0,empty=3,x=0;

int main()
{
    int n;
    void producer();
    void consumer();
    int wait(int);
    int signal(int);
    printf("\n1.Producer\n2.Consumer\n3.Exit");
    while(1)
    {
        printf("\nEnter your choice:");
        scanf("%d",&n);
        switch(n)
        {
            case 1:  if((mutex==1)&&(empty!=0))
                    producer();
                    else
                    printf("Buffer is full!!!");

```



```

        break;
    case 2:  if((mutex==1)&&(full!=0))
            consumer();
            else
                printf("Buffer is empty!!");
            break;
    case 3:
        exit(0);
        break;
    }
}

return 0;
}

int wait(int s)
{
    return (--s);
}

int signal(int s)
{
    return(++s);
}

void producer()
{
    mutex=wait(mutex);
    full=signal(full);
    empty=wait(empty);
    x++;
    printf("\nProducer produces the item %d",x);
    mutex=signal(mutex);
}

void consumer()
{
    mutex=wait(mutex);
    full=wait(full);
    empty=signal(empty);
    printf("\nConsumer consumes item %d",x);
    x--;
    mutex=signal(mutex);
}

```

9. Conclusion and Discussion:

- The Producer consumer problem is also known as bounded buffer problem is solve using the concepts semaphores.
- In this problem we have two processes, producer and consumer, who share a fixed size buffer. Producer work is to produce data or items and put in buffer. Consumer work is to remove data from buffer and consume it.
- Using the concept of semaphore we have to make sure that producer do not produce data when buffer is full and consumer do not remove data when buffer is empty.

10. Viva Questions:

- What is race condition ?
- What is multi-threading ? write advantages of multi-threading

References:

- William Stallings, Operating System: Internals and Design Principles, Prentice Hall, 8th Edition, 2014
- Abraham Silberschatz, Peter Baer Galvin and Greg Gagne, Operating System Concepts, John Wiley & Sons , Inc., 9th Edition, 2016.
- Andrew Tannenbaum, Operating System Design and Implementation, Pearson, 3rd Edition.
- D.M Dhamdhare, Operating Systems: A Concept Based Approach, Mc-Graw Hill
- Maurice J. Bach, “Design of UNIX Operating System”, PHI
- Achyut Godbole and Atul Kahate, Operating Systems, Mc Graw Hill Education, 3rd Edition
- The Linux Kernel Book, Remy Card, Eric Dumas, Frank Mevel, Wiley Publications.

Experiment No. 7

Process Management: Deadlock

Experiment No. 7

1. **Aim:** Write a program to demonstrate the concept of deadlock avoidance through Banker's Algorithm.
2. **Objectives:** From this experiment, the student will be able
 - To study deadlock situation in operating system.
 - To understand Banker's algorithm for deadlock avoidance and detection.
3. **Outcomes:** The learner will be able to
 - Implement and analyse concepts of synchronization and deadlocks
4. **Hardware / Software Required:** Linux Operating System, C

5. **Theory:**

An approach to solving the deadlock problem that differs subtly from deadlock prevention is deadlock avoidance. In deadlock prevention, we constrain resource requests to prevent at least one of the four conditions of deadlock. This is either done indirectly, by preventing one of the three necessary policy conditions (mutual exclusion, hold and wait, no pre-emption), or directly by preventing circular wait. This leads to inefficient use of resources and inefficient execution of processes. Deadlock avoidance, on the other hand, allows the three necessary conditions but makes judicious choices to assure that the deadlock point is never reached. As such, avoidance allows more concurrency than prevention. With deadlock avoidance, a decision is made dynamically whether the current resource allocation request will, if granted, potentially lead to a deadlock. Deadlock avoidance thus requires knowledge of future process resource requests.

We describe two approaches to deadlock avoidance:

- Do not start a process if its demands might lead to deadlock.
- Do not grant an incremental resource request to a process if this allocation might lead to deadlock.

Resource Allocation Denial:

Consider a system of n processes and m different types of resources. Let us define the following vectors and matrices:

[Resource = $\mathbf{R} = (R_1, R_2, \dots, R_m)$	total amount of each resource in the system
Available = $\mathbf{V} = (V_1, V_2, \dots, V_m)$	total amount of each resource not allocated to any process
Claim = $\mathbf{C} = \begin{bmatrix} C_{11} & C_{12} & \dots & C_{1m} \\ C_{21} & C_{22} & \dots & C_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ C_{n1} & C_{n2} & \dots & C_{nm} \end{bmatrix}$	C_{ij} = requirement of process i for resource j
Allocation = $\mathbf{A} = \begin{bmatrix} A_{11} & A_{12} & \dots & A_{1m} \\ A_{21} & A_{22} & \dots & A_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ A_{n1} & A_{n2} & \dots & A_{nm} \end{bmatrix}$	A_{ij} = current allocation to process i of resource j

The matrix Claim gives the maximum requirement of each process for each resource, with one row dedicated to each process. This information must be declared in advance by a process for deadlock avoidance to work. Similarly, the matrix Allocation gives the current allocation to each process. The following relationships hold:

$$R_j = V_j + \sum_{i=1}^n A_{ij}, \quad \text{for all } j \quad \text{All resources are either available or allocated.}$$

$$C_{ij} \leq R_j, \quad \text{for all } i, j \quad \text{No process can claim more than the total amount of resources in the system.}$$

$$A_{ij} \leq C_{ij}, \quad \text{for all } i, j \quad \text{No process is allocated more resources of any type than the process originally claimed to need.}$$

With these quantities defined, we can define a deadlock avoidance policy that refuses to start a new process if its resource requirements might lead to deadlock. Start a new process P_{n+1} only if

$$R_j \geq C_{(n+1)j} + \sum_{i=1}^n C_{ij} \quad \text{for all } j$$

That is, a process is only started if the maximum claim of all current processes plus those of the new process can be met. This strategy is hardly optimal, because it assumes the worst: that all processes will make their maximum claims together.

The strategy of resource allocation denial, referred to as the banker's algorithm, was first proposed by Edsger Dijkstra. Let us begin by defining the concepts of state and safe state. Consider a system with a fixed number of processes and a fixed number of resources. At any time a process may have zero or more resources allocated to it. The state of the system reflects the current allocation of resources to processes. Thus, the state consists of the two vectors, Resource and Available, and the two matrices, Claim and Allocation, defined earlier. A safe state is one in which there is at least one sequence of resource allocations to processes that does not result in a deadlock (i.e., all of the processes can be run to completion). An unsafe state is, of course, a state that is not safe.

ALGORITHM:

Safety algorithm:

1. Let Work and Finish be vectors of length m and n, respectively. Initialize Work=Available and Finish[i]=false for $i=0,1,\dots,n-1$.
2. Find an I such that both
 - a. Finish[i]==false
 - b. $Need_i \leq Work$
3. $Work = Work + Allocation_i$
Finish[i]=true
Go to step 2.
4. If Finish[i]==true for all i, then the system is in a safe state.

Resource-Request Algorithm:

Let Request[i] be the request vector for process P_i . If Request[i][j]==k, then process P_i wants k instances of resource type R_j . When a request for resources is made by process P_i , the following actions are taken:

1. If $Request_i \leq Need_i$, go to step 2. Otherwise raise an error condition, since the process has exceeded its maximum claim.
2. If $Request_i \leq Available$, go to step 3. Otherwise, P_i must wait, since the resources are not available.
3. Have the system pretend to have allocated the requested resources to process P_i by modifying the state as follows:
 $Available = Available - Request_i$;
 $Allocation_i = Request_i$;
 $Need_i = Need_i - Request_i$

Input of the algorithm should Accept No. of processes and need of each resource type for each process from user. Here we accept the processes according to sequence it is to be executed. Output of the algorithm should Display whether given sequence is safe or unsafe.

6. Conclusion and Discussion:

Bankers algorithm is one of the deadlock avoidance algorithm. It is used in a banking system to ensure that the bank never allocated its available cash in such a way that it could no longer satisfy the needs of all its customer.

7. Viva Questions:

- What is deadlock?
- What necessary conditions can lead to a deadlock situation in a system?
- What is Concurrency? Explain with example Deadlock and Starvation.
- What are your solution strategies for “Dining Philosophers Problem”?

- What is Bankers Algorithm?

8. References:

- William Stallings, Operating System: Internals and Design Principles, Prentice Hall, 8th Edition, 2014
- Abraham Silberschatz, Peter Baer Galvin and Greg Gagne, Operating System Concepts, John Wiley & Sons , Inc., 9th Edition, 2016.
- Andrew Tannenbaum, Operating System Design and Implementation, Pearson, 3rd Edition.
- D.M Dhamdhare, Operating Systems: A Concept Based Approach, Mc-Graw Hill
- Maurice J. Bach, “Design of UNIX Operating System”, PHI
- Achyut Godbole and Atul Kahate, Operating Systems, Mc Graw Hill Education, 3rd Edition
- The Linux Kernel Book, Remy Card, Eric Dumas, Frank Mevel, Wiley Publications.

Operating System

Experiment No.: 08

**Memory Management- Dynamic
Partitioning**

Experiment No. 8

1. **Aim:** Write a program to implement dynamic partitioning placement algorithms i.e Best Fit, FirstFit, Worst-Fit etc
2. **Objectives:** From this experiment, the student will be able to
 - Study memory allocation strategies of operating system.
 - Study memory placement algorithms of main memory.
 - Improve both the utilization of the CPU and the speed of its response to its users, the computer must keep several processes in memory.
3. **Outcomes:** The learner will be able to
 - Apply and analyze the concepts of memory management techniques and analyze the performance of memory allocation and replacement techniques.
4. **Hardware / Software Required :** Linux operating system, Java/C
5. **Theory:**

Memory compaction is time consuming; the operating system designer must be clever in deciding how to assign processes to memory (how to plug the holes). When it is time to load or swap a process into main memory, and if there is more than one free block of memory of sufficient size, then the operating system must decide which free block to allocate.

Four placement algorithms that might be considered are best-fit, first-fit, worst-fit and next-fit. All are limited to choosing among free blocks of main memory that are equal to or larger than the process to be brought in. Best-fit chooses the block that is closest in size to the request. First-fit begins to scan memory from the beginning and chooses the first available block that is large enough. Next-fit begins to scan memory from the location of the last placement, and chooses the next available block that is large enough. Worst-fit chooses largest empty block of all. Figure (a) below shows an example memory configuration after a number of placement and swapping-out operations. The last block that was used was a 22-Mbyte block from which a 14-Mbyte partition was created. Best-fit will search the entire list of available blocks and make use of the 18-Mbyte block, leaving a 2-Mbyte fragment. First-fit results in a 6-Mbyte fragment, and next-fit results in a 20-Mbyte fragment. Which of these approaches is best will depend on the exact sequence of process swapping that occurs and the size of those processes.

The first-fit algorithm is not only the simplest but usually the best and fastest as well. The next-fit algorithm tends to produce slightly worse results than the first-fit. The next-fit algorithm will more frequently lead to an allocation from a free block at the end of memory. The result is that the largest block of free memory, which usually appears at the end of the memory space, is quickly broken up into small fragments. Thus, compaction may be required more frequently with next-fit. On the other hand, the first-fit algorithm may litter the front end with small free partitions that need to be searched over on each subsequent first-fit pass. The best-fit algorithm, despite its name, is usually the worst performer. Because this algorithm looks for the smallest block that will satisfy the requirement, it guarantees that the fragment left behind is as small as possible. Although each memory request always wastes the smallest amount of memory, the result is that main memory is quickly littered by blocks too small to satisfy memory allocation requests. Thus, memory compaction must be done more frequently than with the other algorithms.

6. Algorithm:

First Fit Algorithm

1. Start
2. Accept no. of partition(n) and size of each partition(S[i]) and no. of processes(P) and size of each process(SP[i]) from user.
3. Declare i, j, flag[10]
4. for(i=0;i<n;i++)
5. Initialize flag[i]=0
6. End of for loop
7. for(i=0; i<n; i++)
8. for(j=0;j<p;j++)
9. if((s[i]>=sp[j]) && (flag[j]==0)) then
 - flag[j]=1;
 - printf("\nP%d",j);
 - printf(" %d\t\t%d",sp[j],s[i]);
 - break;
10. End of if
11. End of for loop
12. End of for loop
13. End

Best Fit Algorithm:

1. Start
2. Accept no. of partition(n) and size of each partition(S[i]) and no. of processes(P) and size of each process(SP[i]) from user.
3. Declare i, j, flag[10], temp
4. for(i=0;i<n;i++)
5. for(j=i+1;j<n;j++)
6. if(s[i]>=s[j]) then
 - temp=s[i]
 - s[i]=s[j]
 - s[j]=temp
7. End of if
8. End of for
9. End of for
10. for(i=0;i<n;i++)
11. flag[i]=0
12. for(i=0;i<n;i++)
13. for(j=0;j<p;j++)
14. if((s[i]>=sp[j]) && (flag[j]==0)) then
 - flag[j]=1;
 - printf("\nP%d",j);
 - printf(" %d\t\t%d",sp[j],s[i]);
 - break;
15. End of if
16. End of for
17. end of for
18. end

Worst Fit algorithm

1. Start
2. Accept no. of partition(n) and size of each partition(S[i]) and no. of processes(P) and size of each process(SP[i]) from user.
3. Declare i, j, flag[10], temp
4. for(i=0;i<n;i++)
5. for(j=i+1;j<n;j++)

```
6. if(s[i]<=s[j]) then
    temp=s[i];
    s[i]=s[j];
    s[j]=temp
7. End of if
8. End of for
9. End of for
10. End
```

7. Conclusion and Discussion:

There are contiguous and non-contiguous memory allocation methods. In this, we implemented contiguous memory allocation and also we calculated internal and external fragmentation

8. Viva Questions:

- What is Virtual Memory? How is it implemented?
- How does swapping result in better memory management?
- What is thrashing?
- Explain Memory allocation method.

9. References:

- William Stallings, Operating System: Internals and Design Principles, Prentice Hall, 8th Edition, 2014
- Abraham Silberschatz, Peter Baer Galvin and Greg Gagne, Operating System Concepts, John Wiley & Sons , Inc., 9th Edition, 2016.
- Andrew Tannenbaum, Operating System Design and Implementation, Pearson, 3rd Edition.
- D.M Dhamdhere, Operating Systems: A Concept Based Approach, Mc-Graw Hill
- Maurice J. Bach, "Design of UNIX Operating System", PHI
- Achyut Godbole and Atul Kahate, Operating Systems, Mc Graw Hill Education, 3rd Edition
- The Linux Kernel Book, Remy Card, Eric Dumas, Frank Mevel, Wiley Publications.

Operating System

Experiment No.: 9

Page replacement policies

Experiment No. 9

1. **Aim:** Write a program to implement various page replacement policies.
2. **Objectives:** From this experiment, the student will be able
 - To study page replacement algorithms for virtual memory of operating system.
 - Implement variations of page replacement memory management schemes.
 - Be able to compare and contrast the relative performance of different replacement schemes.
3. **Outcomes:** The learner will be able to
 - Apply and analyze the concepts of memory management techniques and analyze the performance of memory allocation and replacement techniques.
4. **Hardware / Software Required :** Linux Operating System, Java/C

5. Theory:

Regardless of the resident set management strategy, there are certain basic algorithms that are used for the selection of a page to replace in main memory. General page replacement algorithms include-

- Optimal
- Least recently used (LRU)
- First-in-first-out (FIFO)

The optimal policy:

It selects for replacement that page for which the time to the next reference is the longest. It can be shown that this policy results in the fewest number of page faults. Clearly, this policy is impossible to implement, because it would require the operating system to have perfect knowledge of future events. However, it does serve as a standard against which to judge real world algorithms. Figure below gives an example of the optimal policy. The example assumes a fixed frame allocation (fixed resident set size) for this process of three frames. The execution of the process requires reference to five distinct pages. The page address stream formed by executing the program is-

2 3 2 1 5 2 4 5 3 2 5 2

which means that the first page referenced is 2, the second page referenced is 3, and so on. The optimal policy produces three page faults after the frame allocation has been filled.

The least recently used (LRU) policy:

It replaces the page in memory that has not been referenced for the longest time. By the principle of locality, this should be the page least likely to be referenced in the near future. And, in fact, the LRU policy does nearly as well as the optimal policy. The problem with this approach is the difficulty in implementation. One approach would be to tag each page with the time of its last reference; this would have to be done at each memory reference, both instruction and data. Even if the hardware would support such a scheme, the overhead would be tremendous. Alternatively, one could maintain a stack of page references, again an expensive prospect.

Figure shows an example of the behavior of LRU, using the same page address stream as for the optimal policy example. In this example, there are four page faults.

The first-in-first-out (FIFO) policy:

It treats the page frames allocated to a process as a circular buffer, and pages are removed in round-robin style. All that is required is a pointer that circles through the page frames of the process. This is therefore one of the simplest page replacement policies to implement. The logic behind this choice,

other than its simplicity, is that one is replacing the page that has been in memory the longest: A page fetched into memory a long time ago may have now fallen out of use. This reasoning will often be wrong, because there will often be regions of program or data that are heavily used throughout the life of a program. Those pages will be repeatedly paged in and out by the FIFO algorithm. In the below example, FIFO policy results in six page faults. Note that LRU recognizes that pages 2 and 5 are referenced more frequently than other pages, whereas FIFO does not.

Example:

Page address stream	2	3	2	1	5	2	4	5	3	2	5	2																																				
OPT	<table><tr><td>2</td></tr><tr><td></td></tr><tr><td></td></tr></table>	2			<table><tr><td>2</td></tr><tr><td>3</td></tr><tr><td></td></tr></table>	2	3		<table><tr><td>2</td></tr><tr><td>3</td></tr><tr><td></td></tr></table>	2	3		<table><tr><td>2</td></tr><tr><td>3</td></tr><tr><td>1</td></tr></table>	2	3	1	<table><tr><td>2</td></tr><tr><td>3</td></tr><tr><td>5</td></tr></table>	2	3	5	<table><tr><td>2</td></tr><tr><td>3</td></tr><tr><td>5</td></tr></table>	2	3	5	<table><tr><td>4</td></tr><tr><td>3</td></tr><tr><td>5</td></tr></table>	4	3	5	<table><tr><td>4</td></tr><tr><td>3</td></tr><tr><td>5</td></tr></table>	4	3	5	<table><tr><td>4</td></tr><tr><td>3</td></tr><tr><td>5</td></tr></table>	4	3	5	<table><tr><td>2</td></tr><tr><td>3</td></tr><tr><td>5</td></tr></table>	2	3	5	<table><tr><td>2</td></tr><tr><td>3</td></tr><tr><td>5</td></tr></table>	2	3	5	<table><tr><td>2</td></tr><tr><td>3</td></tr><tr><td>5</td></tr></table>	2	3	5
2																																																
2																																																
3																																																
2																																																
3																																																
2																																																
3																																																
1																																																
2																																																
3																																																
5																																																
2																																																
3																																																
5																																																
4																																																
3																																																
5																																																
4																																																
3																																																
5																																																
4																																																
3																																																
5																																																
2																																																
3																																																
5																																																
2																																																
3																																																
5																																																
2																																																
3																																																
5																																																
	F	F	F		F		F			F																																						
LRU	<table><tr><td>2</td></tr><tr><td></td></tr><tr><td></td></tr></table>	2			<table><tr><td>2</td></tr><tr><td>3</td></tr><tr><td></td></tr></table>	2	3		<table><tr><td>2</td></tr><tr><td>3</td></tr><tr><td></td></tr></table>	2	3		<table><tr><td>2</td></tr><tr><td>3</td></tr><tr><td>1</td></tr></table>	2	3	1	<table><tr><td>2</td></tr><tr><td>5</td></tr><tr><td>1</td></tr></table>	2	5	1	<table><tr><td>2</td></tr><tr><td>5</td></tr><tr><td>1</td></tr></table>	2	5	1	<table><tr><td>2</td></tr><tr><td>5</td></tr><tr><td>4</td></tr></table>	2	5	4	<table><tr><td>2</td></tr><tr><td>5</td></tr><tr><td>4</td></tr></table>	2	5	4	<table><tr><td>3</td></tr><tr><td>5</td></tr><tr><td>4</td></tr></table>	3	5	4	<table><tr><td>3</td></tr><tr><td>5</td></tr><tr><td>2</td></tr></table>	3	5	2	<table><tr><td>3</td></tr><tr><td>5</td></tr><tr><td>2</td></tr></table>	3	5	2	<table><tr><td>3</td></tr><tr><td>5</td></tr><tr><td>2</td></tr></table>	3	5	2
2																																																
2																																																
3																																																
2																																																
3																																																
2																																																
3																																																
1																																																
2																																																
5																																																
1																																																
2																																																
5																																																
1																																																
2																																																
5																																																
4																																																
2																																																
5																																																
4																																																
3																																																
5																																																
4																																																
3																																																
5																																																
2																																																
3																																																
5																																																
2																																																
3																																																
5																																																
2																																																
	F	F	F		F		F		F	F																																						
FIFO	<table><tr><td>2</td></tr><tr><td></td></tr><tr><td></td></tr></table>	2			<table><tr><td>2</td></tr><tr><td>3</td></tr><tr><td></td></tr></table>	2	3		<table><tr><td>2</td></tr><tr><td>3</td></tr><tr><td></td></tr></table>	2	3		<table><tr><td>2</td></tr><tr><td>3</td></tr><tr><td>1</td></tr></table>	2	3	1	<table><tr><td>5</td></tr><tr><td>3</td></tr><tr><td>1</td></tr></table>	5	3	1	<table><tr><td>5</td></tr><tr><td>2</td></tr><tr><td>1</td></tr></table>	5	2	1	<table><tr><td>5</td></tr><tr><td>2</td></tr><tr><td>4</td></tr></table>	5	2	4	<table><tr><td>5</td></tr><tr><td>2</td></tr><tr><td>4</td></tr></table>	5	2	4	<table><tr><td>3</td></tr><tr><td>2</td></tr><tr><td>4</td></tr></table>	3	2	4	<table><tr><td>3</td></tr><tr><td>2</td></tr><tr><td>4</td></tr></table>	3	2	4	<table><tr><td>3</td></tr><tr><td>5</td></tr><tr><td>4</td></tr></table>	3	5	4	<table><tr><td>3</td></tr><tr><td>5</td></tr><tr><td>2</td></tr></table>	3	5	2
2																																																
2																																																
3																																																
2																																																
3																																																
2																																																
3																																																
1																																																
5																																																
3																																																
1																																																
5																																																
2																																																
1																																																
5																																																
2																																																
4																																																
5																																																
2																																																
4																																																
3																																																
2																																																
4																																																
3																																																
2																																																
4																																																
3																																																
5																																																
4																																																
3																																																
5																																																
2																																																
	F	F	F		F	F	F		F		F	F																																				

6. Algorithm:

FIFO – First in first out

1. Accept frame size(fno), Number of pages(count) and sequence of pages(arr[100]) from user
2. Create circular linked list whose size is equal to frame size
3. Initialize each node's data = -99
4. Call create function for CLL
5. When inserting data in a frame check whether
 - a. Page number is already present or not
 - b. There is empty location or not
6. If page is present then don't take any action, and read next page
7. But if page is not present then we have to insert page in LL
8. If Linked List have empty node then goto step 7
9. If Linked List doesn't have empty node then overwrite on the oldest node
10. stop

LRU - Least Recently Used

1. Start
2. Declare the size of frame
3. Get the number of pages to be inserted
4. Get the sequence of pages
5. Declare counter and stack
6. Select the least recently used page by counter value
7. Stack them according the selection.
8. Display the values
9. Stop

Optimal Algorithm

1. Start
2. Declare the required variables and initialize it.
3. Get the frame size and reference string from the user
4. Accommodate a new element look for the element that is not likely to be used in future replace.
5. Count the number of page fault and display the value
6. Stop

7. Conclusion and Discussion:

- We studied different page replacement algorithms, it's advantages and dis-advantages and implemented successfully. We studied concept of paging and segmentation.

8. Viva Questions:

- What is the best page size when designing an operating system?
- What DAT refers to in memory management by an operating system supporting virtual memory ?
- Which of the following page replacement algorithms suffers from Belady's Anomaly?
- A process refers to 5 pages, A, B, C, D, E in the order : A, B, C, D, A, B, E, A, B, C, D, E. If the page replacement algorithm is FIFO, the number of page transfers with an empty internal store of 3 frames is
- A memory page containing a heavily used variable that was initialized very early and is in constant use is removed, then the page replacement algorithm

9. References:

- William Stallings, Operating System: Internals and Design Principles, Prentice Hall, 8th Edition, 2014
- Abraham Silberschatz, Peter Baer Galvin and Greg Gagne, Operating System Concepts, John Wiley & Sons , Inc., 9th Edition, 2016.
- Andrew Tannenbaum, Operating System Design and Implementation, Pearson, 3rd Edition.
- D.M Dhamdhare, Operating Systems: A Concept Based Approach, Mc-Graw Hill
- Maurice J. Bach, "Design of UNIX Operating System", PHI
- Achyut Godbole and Atul Kahate, Operating Systems, Mc Graw Hill Education, 3rd Edition
- The Linux Kernel Book, Remy Card, Eric Dumas, Frank Mevel, Wiley Publications.

Operating System

Experiment No.: 10

Disk scheduling algorithms

Experiment No. 10

1. **Aim:** Write a program to implement any two Disk scheduling algorithms like FCFS, SSTF, SCAN etc.
2. **Objectives:** From this experiment, the student will be able
 - To get the concepts of Storage Management, Disk Management and disk scheduling
3. **Outcomes:** The learner will be able to
 - Apply and analyze different techniques of file and I/O management
4. **Hardware / Software Required:** Linux Operating System, JAVA/C

5. Theory:

A hard disk drive is a collection of plates called platters. The surface of each platter is divided into circular tracks. Furthermore, each track is divided into smaller pieces called sectors. Disk I/O is done sector by sector. A group of tracks that are positioned on top of each other form a cylinder. There is a head connected to an arm for each surface, which handles all I/O operations. For each I/O request, first head is selected. It is then moved over the destination track. The disk is then rotated to position the desired sector under the head= and finally, the read/write operation is performed.

There are two objectives for any disk scheduling algorithm:

1. Maximize the throughput - the average number of requests satisfied per time unit.
 2. Minimize the response time - the average time that a request must wait before it is satisfied.
- Some of the disk scheduling algorithms are explained below.

- **FCFS (First Come, First Served)**
 - perform operations in order requested
 - no reordering of work queue
 - no starvation: every request is serviced
 - poor performance
- **SSTF (Shortest Seek Time First)**
 - after a request, go to the closest request in the work queue, regardless of direction
 - reduces total seek time compared to FCFS
 - Disadvantages
 - starvation is possible; stay in one area of the disk if very busy
 - switching directions slows things down
- **SCAN**
 - go from the outside to the inside servicing requests and then back from the outside to the inside servicing requests.
 - repeats this over and over.
 - reduces variance compared to SSTF.
- **LOOK**
 - like SCAN but stops moving inwards (or outwards) when no more requests in that direction exist.
- **C-SCAN (circular scan)**
 - moves inwards servicing requests until it reaches the innermost cylinder; then jumps to the outside cylinder of the disk without servicing any requests.
 - repeats this over and over.

- variant: service requests from inside to outside, and then skip back to the innermost cylinder.
- **C-LOOK**
 - moves inwards servicing requests until there are no more requests in that direction, then it jumps to the outermost outstanding requests.
 - repeat this over and over.
 - variant: service requests from inside to outside, then skip back to the innermost request.

6. Algorithm:

Step 1: Start

Step 2: Read the number of processes and the requested tracks.

Step 3: In FCFS, the processes are scheduled according to the order in which they arrive.

Step 4: In SSTF, the next process to be scheduled is selected as the one requiring the minimum seek time from the current position of disk head.

Step 5: In SCAN, the processes are scheduled from track 0 to highest numbered track. After reaching highest numbered track, it schedules the processes from there to track 0 and so on.

Step 6: Circular SCAN always schedules the processes from track 0 to the highest numbered track.

Step 7: Stop.

7. Conclusion and Discussion:

- There are different types of disk scheduling algorithms. When selecting a Disk Scheduling algorithm, performance depends on the number and types of requests. SSTF is common and has a natural appeal.
- SCAN, C-SCAN for systems that place a heavy load on the disk, as they are less likely to cause starvation

8. Viva Questions:

- What data structure for a sector typically contain?
- What is the terminology used for the Defective sectors on disks?
- What happen if a process needs I/O to or from a disk, and if the drive or controller is busy?
- What is the terminology used for the time taken to move the disk arm to the desired cylinder?

9. References:

- William Stallings, Operating System: Internals and Design Principles, Prentice Hall, 8th Edition, 2014
- Abraham Silberschatz, Peter Baer Galvin and Greg Gagne, Operating System Concepts, John Wiley & Sons , Inc., 9th Edition, 2016.
- Andrew Tannenbaum, Operating System Design and Implementation, Pearson, 3rd Edition.
- D.M Dhamdhare, Operating Systems: A Concept Based Approach, Mc-Graw Hill
- Maurice J. Bach, "Design of UNIX Operating System", PHI
- Achyut Godbole and Atul Kahate, Operating Systems, Mc Graw Hill Education, 3rd Edition
- The Linux Kernel Book, Remy Card, Eric Dumas, Frank Mevel, Wiley Publications.